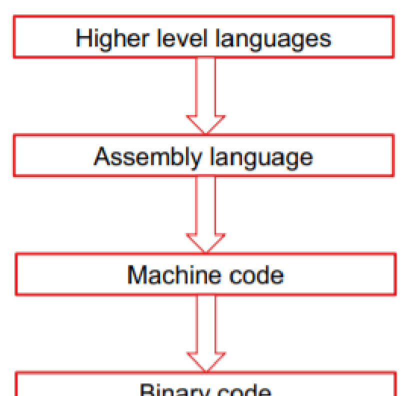
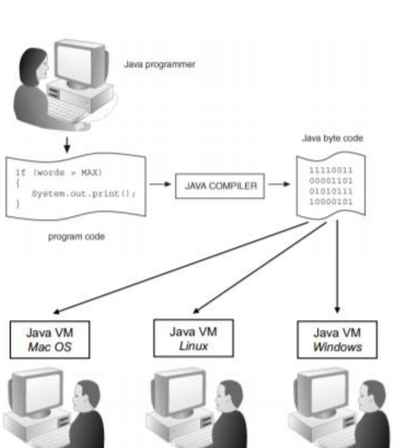
WEEK 1 – FIRST STEPS

SESSION 1: INTRODUCTION





- Programming is a way to tell computers to do a specific task
- Java is a general purpose, object-oriented programming language from 1995
- Software developer usually codes in higher level languages or in assembly
- Machine code can also be expressed in hexadecimal, binary code only in binary.

HIGHER-LEVEL PROGRAMMING LANGUAGES

- **C, C++, Haskell, Go…**
  - Compiled for a specific (hardware) platform *(the compiler will produce an object file, that is, a file that contains machine code, and such machine code may or may not have been based on a specific assembly language. In other words, it may has been translated directly from higher level language to machine code by the compiler)*
- **Java, C#, Scala…**
  - Compiled to an intermediate format, for java this is **bytecode** (these are the .class files), which are **executed** by a virtual machine (instead of a specific hardware). A **virtual machine** is a piece of software that translate your program to run on the hardware underneath it. *Which means that the same **bytecode** can be run on all hardware devices that support a Java virtual machine, thus not needing to compile it each time.*
- **Python, Ruby, PHP…**
  - These are not compiled, a special program (**interpreter**) directly executes them.

PROGRAMMING PARADIGMS IN CSE1100

- **Imperative programming**
  - List of **statements** (combination of **data** and **operations**)
- **Object-oriented programming**
  - Classes
  - Objects
  - Statements (composed of **objects**, **data** and **operations**)

DATA AND OPERATIONS

Data is stored in bytes (groups of 8 bits).

Primitive data types:

	Java type	Bits	Range
Booleans:	boolean		{ true, false }
Integers:	byte	8	-128 to 127
	short	16	-32786 to 32767
	int	32	-2147483648 to 2147483647
	long	64	-9223372036854775808 to 9223372036854775807
Real numbers:	float	32	+/- 1.4 * 10^{-45} to 3.4 * 10^{38}
	double	64	+/- 4.9 * 10^{-324} to 1.8 * 10^{308}
Characters:	char	16	Unicode character set

Reference data types:

String, Anything else…

The string type stores strings of characters.

Declaration:

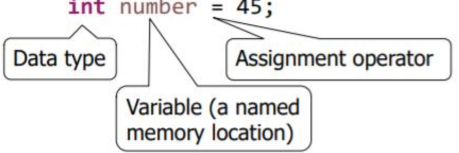
```
int number;
```

Assigning a value:

```
number = 45;
```

Combined (declaration with initialisation):

```
int number = 45;
```



Question:

What is the value of a local variable after its declaration?

```
int number;
```

Answer: it is *unassigned* and you will get a compiler error if you try to use it before initialising it*. (Class fields behave differently: see future lecture.)

Operators:

= Assignment

\+ - * / % Arithmetic

\+ string concatenation

```
int first = 4;              statement

first + 6                   expression (value 10)
first / 5                   expression (value 0)

int second = first - 2;     statement
```

You can mix integers and floating point numbers:

```
2 + 3.5 → 5.5               automatic type cast
2 + (double) 3 → 5.0        type cast
```

Note that + is also used for string concatenation:

```
"Hello, " + "World!" → "Hello,  World!"
```

The +, - and * operators work mostly as expected:

```
2 + 3 → 5              result of type int
2.5 * 3.0 →7.5         result of type double
```

Be aware of integer overflow though:

```
2147483647 + 1 → -2147483648
```

in binary:

```
01111111_11111111_11111111_11111111 + 1
→ 10000000_00000000_00000000_00000000
```

When both operands are integers, the / operator performs integer division, which rounds towards 0:

```
5 / 3 → 1
5 / (-3) → -1
3 / 4 → 0
```

If this is not what you want, you must force Java to use floating-point division by using at least one floating-point number:

```
3.0 / 2 → 1.5
```
or: `3 / 2.0 → 1.5`
or: `(double) 3 / 2 → 1.5`

Data type = the combination of the (set of) data elements and the (set of) allowed operators

Outputting data to the screen is done with the methods print() and println(). Which are part of the object System.out.

Compiling and running the program

Text editor (Notepad, Notepad++, Atom, Vim, ...):
Used to compose the source code

Compiler (`javac`):
Translates source code to bytecode

Virtual machine (`java`):
Executes bytecode

Call to compiler: `javac Hello.java`
Call to virtual machine: `java Hello`

Reading input data using `Scanner`

```
import java.util.*;

public class ScannerExample {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        int number = scanner.nextInt();
        double average = scanner.nextDouble();
        String name = scanner.next();
        // Do something with the input...
    }
}
```

3!

\

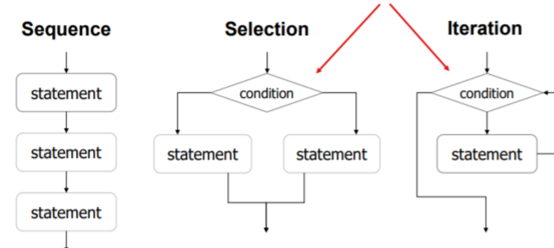
Structure in programs

A program is written once, but read many more times
→ Write readable code

1. **Indentation:** to show that certain parts belong together
2. **Separation:** use spaces, empty lines, capital/smaller letters
3. **Cohesion:** show parts that logically belong together
4. **Variable naming:** Use variable names that clearly express the role of the variable
5. **Comments:** use comments to explain

Rule number 1 in OOP. **Never copy-paste code**. Any code that can be reused shall be made into a callable method.
Reason: **more readable and easier to maintain**.

Control structures



Equality operators

== equal to
!= not equal to

- Numerical equality (`5 == 5`)
- Boolean equality (`true != false`)
- Reference equality (for comparing objects)

Should **not** be used to test the equality of `Strings`!

Also be careful when testing `floats` or `doubles` for equality.

Strings are a reference data type, which means that when the equality operators compare the memory contents of the strings, it will compare the addresses of the objects they refer to and not the actual contents of the objects themselves. That is why Strings come with a built-in equals() method. Should your created objects be compared, then you'd have to **override** (overwrite) an equals method in the class of your object.

Relational operators

<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to

Only used for numerical comparison (`5 < 7`, `'a' < 'b'`).

Boolean logical operators

& logical AND
| logical OR
^ logical XOR
! logical complement

a	b	a & b	a | b	a ^ b
false	false	false	false	false
false	true	false	true	true
true	false	false	true	true
true	true	true	true	false

Integer bitwise operators

& bitwise AND
| bitwise OR
^ bitwise XOR
~ bitwise complement

```
0b00001111 & 0b01010101 → 0b00000101
0b00001111 | 0b01010101 → 0b01011111
0b00001111 ^ 0b01010101 → 0b01011010
~0b...00000000 → 0b...11111111
```

(0b means "interpret this number as binary")

a	!a
false	true
true	false

Conditional operators

&& conditional AND
|| conditional OR

These operators are "lazy": the right-hand side is only evaluated when necessary.

a	b	a && b
false	<anything>	false
true	false	false
true	true	true

a	b	a || b
false	false	false
false	true	true
true	<anything>	true

Selection

```
if (<condition>)
 <statement>;

if (<condition>)
   <statement>;
else
   <statement>;
```

Example

```
if (x != 0)
    y = 1/x;

if (x > y)
    max = x;
else
    max = y;
```

Example, with blocks

```
if (x > y) {
    System.out.println("x is greater than y");
    System.out.println("y is smaller than x");
}
else {
    System.out.println("x is smaller than/equal to y");
    System.out.println("y is greater than/equal to x");
}
```

Switch-statement: table with limited number of values

```
String result;

switch (grade) {
   case 10:
      result = "Perfect";
      break;
   case 9:
   case 8:
   case 7:
   case 6:
      result = "Pass";
      break;
   default:
      result = "Fail";
}
```

- Type of condition: byte, short, char, int, Character, Byte, Short, Integer, String and enum types → `switch (grade) {`
- Stops any further case matching → `break;`
- Is assigned when result is 9, 8, 7 or 6 → `result = "Pass";`
- Catch all for when no cases match → `default:`

Repetition

What do we need to control the repetition:

- A **counter** to keep track of how many times a row has been printed
  `int row = 0;`
- A **(stopping) condition** to end the repetition.
  `row == 8`
- An **increment** of the counter after printing the row
  `row = row + 1;`

(This was an example to print the same text n rows)

For-statement

```
for (int row = 0; row < 8; row = row + 1)
    System.out.println("********");
```

The **condition for repetition**: `row != 8`

is the inverse of the **stopping condition**: `row == 8`

While statement: printing 8 stars

```
// 0 stars have been printed
int col = 0;

// number of stars is not (yet) equal
while (col != 8){
    System.out.print('*');
    // 1 new star has been printed
    col = col + 1;
}
// the number of stars equals 8
```

do … while

```
do {
    // statements
} while (x > 0);
```

This loop is always executed at least once. Only after this first execution, the condition is checked!

WEEK 1 – TUTORIAL

HELLO WORLD

```
package weblab;

class HelloWorld {

  public static void run() {
    System.out.println("Hello, World!");

  }

}

// notice that the name of the class has to be the same of the .class file, case sensitive!
```

REAL NUMBERS

```
package weblab;

class RealNumbers {

  public static void run(){

    int divInt = 1 / 3;
    float divFloat = 1/3;
    // first will do the int 1 / int 3 division (which yields 0) and then it will cast the ou
tcome to float
    double divDouble = 1.0 / 3.0;

    System.out.println(
      "The value of the int is: " + divInt +
      "\nThe value of the float is: " + divFloat +
      "\nThe value of the double is: " + divDouble
    );

  }

}
```

```
Status: Done

The value of the int is: 0

The value of the float is: 0.0

The value of the double is: 0.3333333333333333


Test score: 1/1
```

ITERATION

```java
class Iteration {

  public static void run(){

    printNumbersUpToFor(5);

    printNumbersUpToWhile(5);

  }

  /** Prints the numbers from 1 up to N
   *  @param n - an integer value
   */
  public static void printNumbersUpToFor(int n){
    for (int i=1; i<=n; i++){
    System.out.println(i);
    }
  }

  /** Prints the numbers from 1 up to N
   *  @param n - an integer value
   */
  public static void printNumbersUpToWhile(int n){
    int i=1;
    while (i<=n){
      System.out.println(i);
      i++;
    }
  }

}
```

```
Status: Done

1

2

3

4

5

1

2

3

4

5


Test score: 1/1
```

WEEK 2 – METHODS AND CLASSES

METHODS

Any piece of code that will be run more than once must not be copy pasted but made into a callable method.
Methods are like a mathematical function, they may take parameters and return or do something. They may also not.

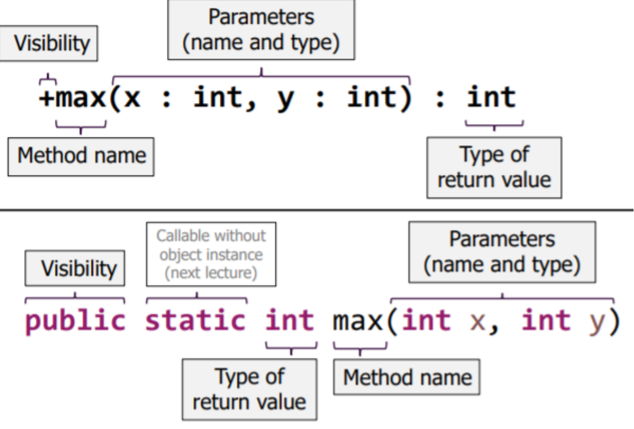
```
 /** Takes two arguments with type double, and returns the largest. If equal returns it in pa
ramater order.
   * @param x - first double
   * @param y - second double
   * @param z - third double
   */

  public static double max(double x, double y) {
    if (x>=y) {
      return x;
    } else {
      return y;
    }
  }
```

The comments /* inside this */ above the method is "JavaDoc" Java Documentation. Each method needs to be documented. The practice says that if the method explanation takes more than 1 line to explain, it is probably a complex enough method to split it into more methods..

Note: **return** is a keyword that says that what comes after it needs to be returned in this case as double, because the method is declared with return type int, and it also marks the end of the method execution. Any other lines of code after it, regardless of their indentation, will be skipped altogether.

A **void** method does not return something. (It does execute something though). You can still use the return keyword to "branch out" from the method call, but return is not obligatory for void methods.

Pre: (precondition) defines the initial state before the calculation of the method.
Post: (postcondition) defines the end state after the calculation in the method.



Some interesting methods:

```
public static boolean isEven(int i){
    return (i%2==0);
}
public static boolean isPrime(int n) {
    for (int i = 2; i < n; i++) {
        if (n % i == 0)
            return false;
    }
    return true;
}
public static int factorial(int n){
        if (n==0) {
      return 1;
    }
        return (n * factorial(--n));
} //this one is recursive
```

Note that the visibility of something is public when + and private when -

PASS BY VALUE

Consider:

```
public class TwoMethods {
    public static int max(int x, int y) {
        int res = x;
        if (y > res)
            res = y;
        return res;
    }

    public static void main(String[] args) {
        int a = 4;
        int b = 12;
        int res = max(a, b);
        System.out.println(res);
    }
}
```

main()	
a	4
b	12
res	

max()	
x	4
y	12
res	

x and **y** are the **formal parameters** of the implementation,
a and **b** are the **actual parameters** at the call of the method

Actual paramaters = original variable at a specific method/class
Formal paramaters = pass by value "copied" from the actual one

Lifetime of variables

A variable declared within a block, will get cleaned (garbage collected) when control leaves the block.
→ garbage collection is freeing up unused memory

```
{
    int x = 12; System.out.print(x + " ");
    {
        int y = 34; System.out.print(y + " ");
    }
    System.out.println(x + " " + y);
}
```

Compiler reports this error:

```
Unresolved compilation problem:
    y cannot be resolved to a variable
```

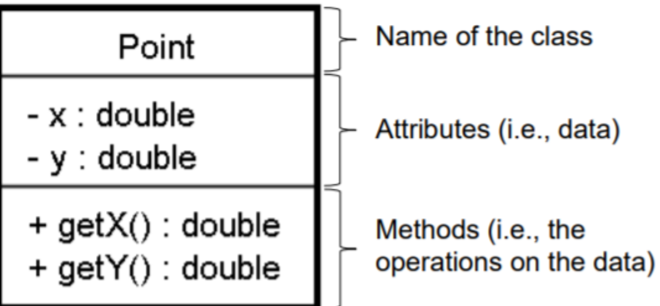
CLASSES

A class is a software module that is composed of attributes and methods.

```
Math
+PI = 3.14159
+E  = 2.71818
+abs(val : int) : int
post: returns absolute value of val

+sin(rad: double) : double
post: returns sinus of rad

+sqrt(val: double) : double
post: returns square root of val
```

Usage: `ClassName.method(), ClassName.CONSTANT`

As you can observe, the Pi and Euler **constants** are **public attributes**, which can be retrieved by Class.CONSTANT. Note that **constants cannot be changed** and that are **represented in capital letters**. This is done by using **final**.

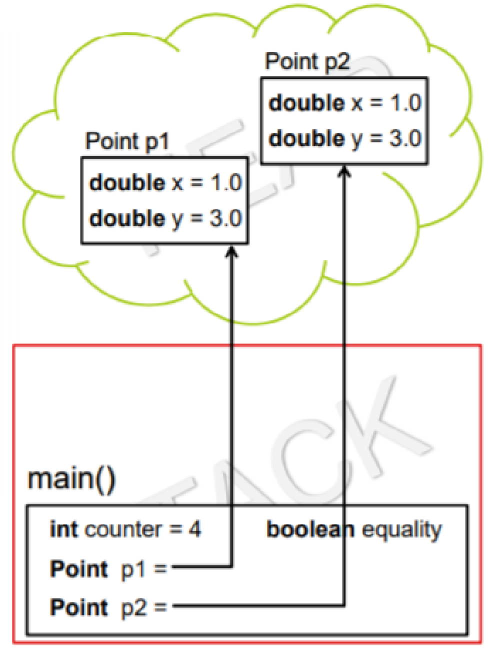


Classes generally have **private attributes** (except for constants, which generally are public because they can't be changed anyhow). The attributes are private so that **they can't be changed by anything from the outside**. **(encapsulation:** hide implementation and prevent errors**)**

To operate with those private attributes classes generally include "**getters**" and "**setters**", as well as "**constructor"** methods. The constructor method must have the same name as the class and is used when an object of that class is being declared and initialized.

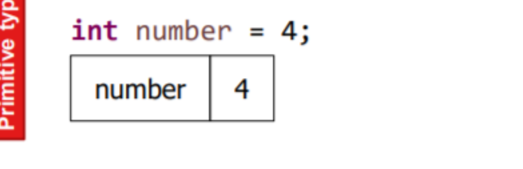
MEMORY & EQUALITY

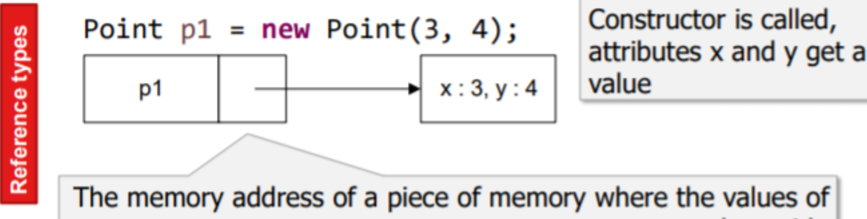
Absolutely **all variables** (data types) of all types (primitive and reference) **point to a single memory location in the stack**. **In primitives, that memory location contains the initialized/assigned values of the primitive**. For instance, the memory location of an int (32 bits) points to the lower end byte of the 4 memory bytes that contain the 32 bits of the integer, which contain all the contents of our variable. **However, reference types such us objects usually contain many attributes that cannot be fitted into a few addressable memory stack bytes**. **Therefore the stack memory located to a reference type variable is a value that points to an address in the heat memory space that contains the actual contents of the reference type.**



Is for that reason that the equals operator doesn't work well with reference types and it is the responsibility of the newly created class to include an equals method.

The constructor method will declare (and assign) primitives and other reference type variables which at the end will calculate how much memory needs to be allocated for the created object.





In stack terms, it would be like:
Number = stack address 1160 (which contains "4")
P1 = stack address 2320 (which contains "heap address 1992")
So when you compare number A with Number B you will be compare two integers, which is a legit comparison with ==, whereas when you compare two reference types you are just comparing two heap addresses, not a legit comparison.

EQUALS METHOD

```
  /**Compares this Student to another object and assesses whether they are
   * both students with equal values.
   * @param other - an Object that is to be compared to this.
   * @return a boolean value which is true when both objects are of type Student
   * with the same name and studentNumber values, and false otherwise.
   */
public boolean equals(Object other) {
    if (other == null){
      return false;
    }
    if (other instanceof Student) {
      Student that = (Student) other;
      return (this.name.equals(that.name) && this.studentNumber == that.studentNumber);
    } else {
      return false;
    }
  }
```

Observe that we first get rid of possible null pointer errors. Then we check if the object is actually of the same class and if so we may use its casted contents to create an equivalent variable but officially of the relevant class. By default java doesn't know the class of the object being passed on and it would be very cumbersome to cast each pass by value each time, that is why a separate temporary object is used instead (local, lives only during the equal method).

Class versus object

- A **class** is a blueprint for objects; it defines:
  - Attributes
  - Methods
- An **object** is an instance of a class, with its own values for the attributes

```java
public static int numberOfDigits(String s) {
    // returns number of digits in String s

    int res = 0;
    for (int i = 0; i < s.length(); i++) {
        char c = s.charAt(i);
        if (Character.isDigit(c))
            res = res + 1;
    }
    return res;
}
```

A static method does not work on an object! This implies:
- no reference to "this"
- no access to attributes (it can access static attributes)

Static methods are class methods rather than object specific methods.
Similarly a static attribute is an attribute common to all objects of a class

What about a static attribute?

- Is common to all objects of a class

```java
public class Account {
    private static int nextFreeNumber = 0;
    private int accountNumber;

    public Account(String name) {
        accountNumber = nextFreeNumber;
        nextFreeNumber++;
    }
}
```

Each Account object gets a unique accountNumber

A callable static method from outside the class could be any mathematical method called from the outside by using Class.method(), i.e. Math.sine() instead of MathObject.sine()

Static methods are important because the main method, the first one that runs in the execution of the java program, is called from the outside.

How to use static?

```java
String.valueOf(true);

String s = new String();
s.trim();

s.valueOf(false);
```

A static method is called through the **class**

Non-static method call through object

Static method call through object → will work, but compiler warning

Java pass by value (objects)

```java
// method in class Point
public static void swap(Point point) {
    double temp = point.x;
    point.x = point.y;
    point.y = temp;
}

public static void main(String[] args) {
    Point p = new Point(10, 12);
    System.out.println(p.getX() + "," + p.getY());
    Point.swap(p);
    System.out.println(p.getX() + "," + p.getY());
}
```

```
Output:    10,12
           12,10
```

In terms of primitives, Java you cannot change the actual parameters from inside the method for those outside the method. A method can only return a primitive at most. However, **in terms of reference type**, if the attributes are public or if the operations are being made within the same class**, it is possible to change the actual parameters of pass by object attributes from within a method**. See swap example on the left.

This is because **although the memory address of the passed by value "point" cannot change** (heap address doesn't change) **you can change the contents inside such heap address**.

Example uses of char variable

```java
char c = 65;
System.out.println(c);              // => A
System.out.println(c+1);            // => 66
System.out.println((char)(c + 1));  // => B
```

Class **Character** contains many useful methods, such as **Character.isLowerCase(char c)**.

char: primitive type
Character: class

Or others like **isUpperCase()**, **isDigit()**, **isLetter()**, **isLetterOrDigit()**, **isSpace()** and **isWhiteSpace()**.

The char datatype has has 65536 different elements.
These are the first few of them:

0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
16	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
32		!	"	#	$	%	&	`	(	)	*	+	,	-	.	/
48	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
64	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
96	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
112	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	del

WEEK 2 – TUTORIAL

POINT (CONSTRUCTOR AND GETTERS)

In this assignment we want to define the `class Point`. Point has two attributes: the variables `x` and `y`. Point also has several methods: `Point(double x, double y)`, `getX()` & `getY()`.

The method `Point(double x, double y)` is special here *(note that it's written with a capital P!)*, it is a **constructor**. We call this method when we want to allocate memory to create an instance of a `Point`.

Create the `class Point`, the constructor and the two getters. `getX()` should return the `x` coordinate of the `Point`, and `getY()` should return the `y` coordinate

SOLUTION

```java
// Your class here.
class Point {

  private double x;
  private double y;

  public Point(double x, double y){
    this.x = x;
    this.y = y;
  }

  public double getX(){
    return this.x;
  }
  public double getY(){
    return this.y;
  }
}
```

TEST

```java
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class UTest {

  @Test
  public void constructorTest() {
    Point p = new Point(2.0, 4.0);
    assertNotNull(p, "Expected Point p to be not null, but it was null.");
  }

  @Test
  public void getXTest() {
    Point p = new Point(2.0, 4.0);
    assertEquals(2.0, p.getX(), 0.0001, "Expected the x coordinate to be 2.0, but it was not.");
  }

  @Test
  public void getYTest() {
    Point p = new Point(2.0, 4.0);
    assertEquals(4.0, p.getY(), 0.0001, "Expected the y coordinate to be 4.0, but it was not.");
  }
}
```

Generally we want to do (at least) 1 test for each method. You can see that the assertEquals tet method has a 3rd parameter, it is the absolute difference that is willing to ignore between the compared floats. This is necessary due to the natural imprecision of the floating numbers.

PARKING LOTS CAPACITY

In this assignment we are going to add some functionality to the class ParkingGarage. A parking garage has a size (total amount of cars that can park there) and an actual count of cars that is currently parked in the garage. There are *getters* for both properties. Now you are asked to add several methods to this class:

public String parkCar()
Post: Adds one to the amount of parkedCars if this amount is smaller than the size of the garage, and then returns "*Car parked successfully!*". Otherwise returns "*The garage is full!*"
public int freeSpaces()
Post: Returns the number of spaces still available by subtracting the amount of parked cars from the size of the garage

SOLUTION

```
class ParkingGarage {
  private int size;
  private int parkedCars;

  /**Constructor: creates a new ParkingGarage with size n, and 0 parked cars.
   * @param size - An int value representing the total amount of parking spaces in the garage
   */
  public ParkingGarage(int size) {
    this.size = size;
    parkedCars = 0;
  }

  /**Getter for the size of the ParkingGarage.
   * @return the value of size.
   */
  public int getSize() {
    return size;
  }

  /**Getter for the amount of cars parked in the ParkingGarage.
   * @return the value of parkedCars.
   */
  public int getParkedCars() {
    return parkedCars;
  }

  public String parkCar(){
    if (this.getParkedCars() < this.getSize()) {
      this.parkedCars++;
      return "Car parked successfully!";
    } else {
      return "The garage is full!";
    }
  }

  public int freeSpaces(){
    return (this.size - this.parkedCars);
  }
}
```

TEST

```java
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class UTest {

    @Test
    public void constructorTest() {
      ParkingGarage p = new ParkingGarage(500);
      assertNotNull(p,  "Expected ParkingGarage p to be not null, but it was null");
    }

    @Test
    public void getSizeTest() {
      ParkingGarage p = new ParkingGarage(500);
      assertEquals(500, p.getSize(), "Expected the size of the garage to be 500, it wasn’t");
    }

    @Test
    public void getParkedCarsTest() {
      ParkingGarage p = new ParkingGarage(500);
      assertEquals(0, p.getParkedCars(), "Expected the number of parked cars in the garage to be 0, but it was not");
    }

    @Test
    public void parkCarTest() {
      ParkingGarage p = new ParkingGarage(500);
      p.parkCar();
      p.parkCar();
      assertEquals(2, p.getParkedCars(), "Expected the number of parked cars in the garage to be 2, but it was not");
    }

    @Test
    public void parkCarSuccessTest() {
      ParkingGarage p = new ParkingGarage(1);
      assertEquals("Car parked successfully!", p.parkCar(), "Expected a success message but got something else.");
    }

    @Test
    public void parkCarFullTest() {
      ParkingGarage p = new ParkingGarage(1);
      p.parkCar();
      assertEquals("The garage is full!", p.parkCar(), "Expected the garage to be full, but got something else.");
    }

    @Test
    public void freeSpacesTest() {
      ParkingGarage p = new ParkingGarage(500);
      p.parkCar();
      p.parkCar();
      assertEquals(498, p.freeSpaces(), "Expected the number of free spaces in the garage to be 498, but it was not");
    }
}
```

EQUALS

The equals method returns a boolean value that is `true` when the `Object other` is a `Student` with the same `name` and `studentNumber` as `this`, otherwise it returns `false`. This means that two different Student objects with the same `name` and `studentNumber` will be seen as equal even though they have different memory addresses.

Write the implementation for the equals method of `Student`. Make sure your implementation checks all the properties above.

SOLUTION

```
class Student {
  private String name;
  private int studentNumber;

  /**Constructor: creates a new Student with a name and a studentNumber.
   * @param name - A String value representing the name coordinate of the student
   * @param studentNumber - An int value representing the studentNumber coordinate of the stu
dent
   */
  public Student(String name, int studentNumber) {
    this.name = name;
    this.studentNumber = studentNumber;
  }

  /**Getter for the name coordinate of the Student.
   * @return the value of name.
   */
  public String getName() {
    return this.name;
  }

  /**Getter for the studentNumber of the Student.
   * @return the value of studentNumber.
   */
  public int getStudentNumber() {
    return this.studentNumber;
  }

  /**Compares this Student to another object and assesses whether they are
   * both students with equal values.
   * @param other - an Object that is to be compared to this.
   * @return a boolean value which is true when both objects are of type Student
   * with the same name and studentNumber values, and false otherwise.
   */
  public boolean equals(Object other) {
    if (other == null){
      return false;
    }
    if (other instanceof Student) {
      Student that = (Student) other;
      return (this.name.equals(that.name) && this.studentNumber == that.studentNumber);
    } else {
      return false;
    }
  }
}
```

TEST

```java
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class UTest {

  @Test
  public void constructorTest() {
    Student s = new Student("Henk", 12345);
    assertNotNull(s,  "Expected Student s to be not null, but it was null");
  }

  @Test
  public void getNameTest() {
    Student s = new Student("Henk", 12345);
    assertEquals("Henk", s.getName(), "Expected the student name to be 'Henk', but it was not");
  }

  @Test
  public void getStudentNumberTest() {
    Student s = new Student("Henk", 12345);
    assertEquals(12345, s.getStudentNumber(), "Expected the student number to be 12345, but it was not");
  }

  @Test
  public void testEqualsSelf(){
    Student student = new Student("Henk", 12345);
    assertEquals(student, student, "A Student should be equal to itself.");
  }

  @Test
  public void testEqualsObject(){
    Student student = new Student("Henk", 12345);
    Object other = new Object();
    assertNotEquals(student, other, "A Student not be equals to an object.");
  }

  @Test
  public void testEqualsOther(){
    Student student = new Student("Henk", 12345);
    Student otherStudent = new Student("Henk", 12345);
    assertEquals(student, otherStudent, "Two Student objects with the same properties should be equal.");
  }

  /* If you fail this test you are most likely comparing two String objects by using "==".*/
   @Test
  public void testStringComparisonEquals(){
    Student student = new Student(new String("Henk"), 12345);
    Student otherStudent = new Student(new String("Henk"), 12345);
    assertEquals(student, otherStudent, "Two Student objects with different string memory locations should be equal.");
  }

  @Test
  public void testNotEqualsOtherName(){
    Student student = new Student("Henk", 12345);
    Student otherStudent = new Student("Karel", 12345);
    assertNotEquals(student, otherStudent, "Two Student objects with different student names should not be equal.");
  }

  @Test
  public void testNotEqualsOtherStudentNumber(){
    Student student = new Student("Henk", 12345);
    Student otherStudent = new Student("Henk", 12346);
    assertNotEquals(student, otherStudent, "Two Student objects with different student numbers should not be equal.");
  }

}
```

ASSIGNMENT 1 (CIRCLE)

A CLASS WITH A CONSTRUCTOR

Now you will define the first class yourself. Create a new class with name `Point` (make sure you don't use the keyword `public`).
This class has 2 `private` fields with type `double`: `x` and `y`.
The fields are initialised by calling the constructor with the `x` and `y` values supplied, for example:

`Point p = new Point(2.0, 5.0)`.

CREATING GETTERS

Also define two so called 'getters' for respectively `x` and `y`.

```
public double getX()
public double getY()
```

For our example point `p` defined above:
`p.getX()` should return `2.0`
`p.getY()` should return `5.0`

OVERRIDING TOSTRING()

Override the default `public String toString()` method to produce the following output:
`<Point(X_VALUE, Y_VALUE)>`

For our example point `p.toString()` should return `<Point(2.0, 5.0)>`

ADDING MORE METHODS

Now implement several more methods. Note that you can make use of the methods you implemented for the previous assignment by, for instance, calling: `Assignment1_7.max(YOUR_X, YOUR_Y)` to avoid code duplication.

`public void translate(double dx, double dy)`
Modify the values of the fields in the point by incrementing with respectively dx and dy.

`public double distance(Point other)`
Returns the distance between this point and the other point.

`public boolean equals(Object other)`
If other is also a Point and has equivalent coordinates, return `true`. Else return `false`.

```
package weblab;

class Point {
  private double x;
  private double y;

  public Point (double x, double y){
    this.x = x;
    this.y = y;
  }

  public double getX(){
    return this.x;
  }

  public double getY(){
    return this.y;
  }

  public String toString(){
    String str = "<Point(" + getX() + ", " + getY() + ")>";
    return str;
  }

  public void translate(double dx, double dy){
    this.x = this.x + dx;
    this.y = this.y + dy;
  }

  public double distance(Point other){
    if (this == other) {
      return 0;
    }
    Point that = (Point) other;
    return Assignment1_7.distance(this.x, this.y, that.x, that.y);
  }

  public boolean equals(Object other){
    if (this == other) {
      return true;
    }
    if (other instanceof Point) {
      Point that = (Point) other;
      return this.x == that.x && this.y == that.y;
    }
    return false;
  }
}
```

```
package weblab;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;

import java.util.concurrent.atomic.AtomicBoolean;

class Assignment1PointTest {

    private static final double ONE_X = 3.0;

    private static final double ONE_Y = 5.0;

    private static final double OTHER_X = 7.0;

    private static final double OTHER_Y = 9.0;

    private Point onePoint;

    private Point otherPoint;

    private Point samePoint;

    @BeforeEach
    void setUp() {
        onePoint = new Point(ONE_X, ONE_Y);
        otherPoint = new Point(OTHER_X, OTHER_Y);
        samePoint = new Point(ONE_X, ONE_Y);
    }

    @Test
    void getX() {
        assertEquals(onePoint.getX(), ONE_X, "getX
 should return the x coordinate");
    }

    @Test
    void getY() {
        assertEquals(onePoint.getY(), ONE_Y, "getY
 should return the y coordinate");
    }

    @Test
    void equals_same_point_succesful() {
        assertEquals(onePoint, onePoint, "An objec
t should be equal to itself");
    }

    @Test
    void equals_other_point_failure() {
        assertNotEquals(onePoint, otherPoint, "Dif
ferent objects with different attributes should no
t be equal");
    }

    @Test
    void equals_other_point_succesful() {
        assertEquals(onePoint, (Object) samePoint,
 "Different objects with same attributes should be
 equal");
    }

    @Test
    void equals_null_failure() {
        assertNotEquals(onePoint, null, "Null shou
ld not be equal to an object");
    }

    @Test
    void toString_returns_correct_format() {
        assertEquals("<Point(" + ONE_X + ", " + ON
E_Y + ")>", onePoint.toString(), "toString should
return the correct format");
    }

    @Test
    void equals_String_failure() {
        assertNotEquals("<Point(" + ONE_X + ", " +
 ONE_Y + ")>", onePoint, "Two objects of different
 classes should not be equal");
    }

    @Test
    void translate() {
        onePoint.translate(OTHER_X - ONE_X, OTHER_
Y - ONE_Y);
        assertAll(
            () -
> {assertEquals(onePoint.getX(), OTHER_X, "The x c
oordinate of the point is not translated correctly
");},
            () -
> {assertEquals(onePoint.getY(), OTHER_Y, "The y c
oordinate of the point is not translated correctly
");}
        );
    }

    @Test
    void distance() {
        assertEquals(5.65685424949238, onePoint.di
stance(otherPoint), 1E-
9, "Distance of two points is incorrect");
    }

    @Test
    void equals_does_not_use_tostring() {
        final AtomicBoolean called = new AtomicBoo
lean(false);

        class PointMock extends Point {

            private PointMock(double x, double y)
{
                super(x, y);
            }

            @Override
            public String toString() {
                called.set(true);
                return super.toString();
            }

        }

        new PointMock(ONE_X, ONE_Y).equals(new Poi
ntMock(ONE_X, ONE_Y));
        assertFalse(called.get(), "The equals meth
od should not be using toString for comparison");
    }

}
```

Define a class called `Circle`.
A circle has two `private` fields: `Point center` and `double radius`.
In a similar fashion, define two getters for the fields, a constructor with equivalent arguments and override the toString method.

```
Circle c = new Circle(p, 4.0)
c.toString() → <Circle(<Point(2, 5)>, 4.0)>
```

Note: If the radius provided to the constructor is negative, set the radius to zero.

Note: Correct (hidden) implementations of `Point` and `Assignment1_7` are provided as library for this exercise, the implementation of these methods is correct, regardless of your submission for the previous assignments. Make use of the methods defined in `Point` & `Assignment1_7` when needed to prevent code duplication/copy-pasting.

Lastly, define the following methods:

`public boolean equals(Object other)`
If other is also a Circle and has equivalent Points and radius, return true. Else return false.

`public double circumference()`
Returns the length of the outer periphery of the Circle. Hint: Use `Math.PI`.

`public double surface()`
Returns the total surface of the Circle.

`public void translate(double dx, double dy)`
Translate the center by dx and dy.

`public boolean overlappingWith(Circle other)`
If the other circle overlaps with this circle, return true. Else return false. Hint: draw circles and use method(s) defined in `Assignment_1.7`

```java
package weblab;
class Circle{
  private Point center;
  private double radius;

  public Circle(Point c, double r){
    this.center = c;
    if (r>=0) {
      this.radius = r;
    } else {
      this.radius = 0;
    }
  }

  public Point getCenter(){
    return this.center;
  }

  public double getRadius(){
    return this.radius;
  }

  public String toString(){
    String str = "<Circle(" + this.center.toString() + ", " + this.getRadius() + ")>";
    return str;
  }

  public boolean equals(Object other){
    if (this == other){
      return true;
    }

    if (other instanceof Circle) {
      Circle that = (Circle) other;

      Point p1 = (Point) this.getCenter();
      Point p2 = (Point) that.getCenter();

      if (p1.equals(p2) && this.getRadius() == that.getRadius()){
        return true;
      }
    }
    return false;
  }

  public double circumference(){
    return (2 * Math.PI * this.getRadius());
  }

  public double surface(){
    return (Math.PI * Assignment1_7.squared(this.getRadius()));
  }

  public void translate(double dx, double dy){
    this.center.translate(dx,dy);
  }

  public boolean overlappingWith(Circle other){
    Circle that = (Circle) other;
    if (this == other){
      return true;
    }

    double cd = Assignment1_7.distance(this.center.getX(), this.center.getY(), that.center.getX(), that.center.getY());

    double rs = this.getRadius() + that.getRadius();

    double d = cd - rs;

    if (d<=0){
      return true;
    } else {
      return false;
    }
  }
}
```

```
package weblab;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

import java.util.concurrent.atomic.AtomicBoolean;


@SuppressWarnings("SimplifiableJUnitAssertion")
class UTest {

    private static final double ONE_X = 3.0;

    private static final double ONE_Y = 5.0;

    private static final double OTHER_X = 7.0;

    private static final double OTHER_Y = 9.0;

    private static final double ONE_RADIUS = 3.0;

    private static final double OTHER_RADIUS = 1.5
;

    private static final double EPSILON = 1E-9;

    private Point onePoint;

    private Circle oneCircle;

    private Circle otherCircle;

    private Circle sameCircle;

    @BeforeEach
    void setUp() {
        onePoint = new Point(ONE_X, ONE_Y);
        oneCircle = new Circle(onePoint, ONE_RADIU
S);
        otherCircle = new Circle(new Point(OTHER_X
, OTHER_Y), OTHER_RADIUS);
        sameCircle = new Circle(onePoint, ONE_RADI
US);
    }

    @Test
    void get_center() {
        assertEquals(onePoint, oneCircle.getCenter
(), "getCenter should return the center");
    }

    @Test
    void get_radius() {
        assertEquals(ONE_RADIUS, oneCircle.getRadi
us(), EPSILON, "getRadius should return the radius
");

    }

    @Test
    void get_surface() {
        assertEquals(28.274333882308138, oneCircle
.surface(), EPSILON, "surface should return the ca
lculated surface");
    }

    @Test
    void translate() {
        oneCircle.translate(OTHER_X - ONE_X, OTHER
_Y - ONE_Y);
        assertEquals(OTHER_X, oneCircle.getCenter(
).getX(), EPSILON, "The x coordinate of the point
is not translated correctly");
        assertEquals(OTHER_Y, oneCircle.getCenter(
).getY(), EPSILON, "The y coordinate of the point
is not translated correctly");
    }

    @Test
    void radius_negative_to_zero() {
        Circle circle = new Circle(onePoint, -5);
        assertEquals(0.0, circle.getRadius(), EPSI
LON, "Radius should default to zero if a negative
radius is provided");
    }

    @Test
    void translate_delegates_to_Point_translate()
{
        final AtomicBoolean called = new AtomicBoo
lean(false);

        class PointMock extends Point {

            private PointMock(double x, double y)
{
                super(x, y);
            }

            @Override
            public void translate(double dx, doubl
e dy) {
                called.set(true);
                super.translate(dx, dy);
            }
        }

        new Circle(new PointMock(ONE_X, ONE_Y), ON
E_RADIUS).translate(ONE_X, ONE_Y);
        assertTrue(called.get(), "The translate me
thod of cirlce should delegate to the translate me
thod of point");
    }

    @Test
    void periphery() {
        assertEquals(18.84955592153876, oneCircle.
circumference(), EPSILON, "periphery should return
 the calculated periphery");
    }

    @Test
    void overlappingWith_same_Circle() {
        assertTrue(oneCircle.overlappingWith(oneCi
rcle), "overlappingWith same Circle should be true
");
    }

    @Test
    void overlappingWith_other_failure() {
        assertFalse(oneCircle.overlappingWith(othe
rCircle), "overlappingWith other Circle should be
false");
    }
```

```
    @Test
    void overlappingWith_other_success() {
        oneCircle.translate(OTHER_X - ONE_X - ONE_
RADIUS, OTHER_Y - ONE_Y - ONE_RADIUS);
        assertTrue(oneCircle.overlappingWith(other
Circle), "overlappingWith this other Circle should
 be true");
    }

    @Test
    void toString_returns_correct_format() {
        assertEquals(
                "<Circle(<Point(" + ONE_X + ", " +
 ONE_Y + ")>, " + ONE_RADIUS + ")>",
                oneCircle.toString(),
                "toString should return the correc
t format"
        );
    }

    @Test
    void equals_same_Circle_succesful() {
        assertEquals(oneCircle, oneCircle, "An obj
ect should be equal to itself");
    }

    @Test
    void equals_other_Circle_failure() {
        assertNotEquals(oneCircle, otherCircle, "D
ifferent circles with different points should not
be equal");
    }

    @Test
    void equals_other_Circle_succesful() {
        assertEquals(oneCircle, (Object) sameCircl
e, "Different circles with same attributes should
be equal");
    }

    @Test
    void equals_other_radius_failure() {
        Circle biggerCircle = new Circle(onePoint,
 OTHER_RADIUS);
        assertNotEquals(oneCircle, biggerCircle, "
Different circles with different radius should not
 be equal");
    }

    @Test
    void equals_null_failure() {
        assertNotEquals(oneCircle, null, "Null sho
uld not be equal to an object");
    }

    @Test
    void equals_String_failure() {
        assertNotEquals(
                oneCircle,
                "<Circle(<Point(" + ONE_X + ", " +
 ONE_Y + ")>, " + ONE_RADIUS + ")>",
                "Two objects of different classes
should not be equal"
        );
    }

    @Test
    void equals_does_not_use_tostring() {
        final AtomicBoolean called = new AtomicBoo
lean(false);

        class CircleMock extends Circle {

            private CircleMock(Point point, double
 radius) {
                super(point, radius);
            }

            @Override
            public String toString() {
                called.set(true);
                return super.toString();
            }

        }

        new CircleMock(new Point(ONE_X, ONE_Y), ON
E_RADIUS).equals(new CircleMock(new Point(ONE_X, O
NE_Y), ONE_RADIUS));
        assertFalse(called.get(), "The equals meth
od should not be using toString for comparison");
    }

    @Test
    void equals_delegates_to_equals() {
        final AtomicBoolean called = new AtomicBoo
lean(false);

        class PointMock extends Point {

            private PointMock(double x, double y)
{
                super(x, y);
            }

            @Override
            public boolean equals(Object other) {
                called.set(true);
                return super.equals(other);
            }
        }

        new Circle(new PointMock(ONE_X, ONE_Y), ON
E_RADIUS).equals(new Circle(new PointMock(ONE_X, O
NE_Y), ONE_RADIUS));
        assertTrue(called.get(), "The equals metho
d should delegate equals on attributes");
    }
}
```

WEEK 3 – ARRAYS & UNIT TESTING

ARRAYS

Multiple elements of the **same** data type stored in a single reference.

Declaration: int[] row;
Memory allocation: new int[4]
Binding: int[] row = new int[4];

Must be done together though, can't just declare without allocating memory. You may then populate the contents one by one or in a loop. Note that we always start indexing at 0 and that the length of an array is not retreived from .length()

```
for (int i = 0; i < row.length; i++){
        row[i] = ++i;
}
```

Note that I used the ++ before the i so that the i gets incremented when assigned to row row[i]. Otherwise i++ would assign the current i, and after assigning the current i it would increase it.

faster way: int[4] = {1,2,3,4,5};

```
public static void swap(int x, int y) {
    int temp = x;
    x = y;
    y = temp;
}

public static void main(String[] args) {
    int a = 10;
    int b = 12;
    System.out.println(a + "," + b);
    swap(a, b);
    System.out.println(a + "," + b);
}
```

Remember this slide (again)?

Output: 10,12
10,12

```
public static void swap(int[] row) {
    int temp = row[0];
    row[0] = row[1];
    row[1] = temp;
}

public static void main(String[] args) {
    int[] row = {10, 12};
    System.out.println(row[0] + "," + row[1]);
    swap(row);
    System.out.println(row[0] + "," + row[1]);
}
```

Output: 10,12
12,10

Arrays are also reference type, therefore as contrary to the primitives, its contents CAN be changed within a method. Because although the passby value (heap address) remains the same its contents can be re-assigned (changed).

Useufl array methods:

```
public static String toString(int[] row)
{
    String result = "";
    int n = row.length;

    for (int i = 0; i < n; i = i + 1)
        result = result + row[i] + " ";

    return result;
}
```

This one will return the max or Integer.MIN_VALUE if all contents are null

```
public static int max(int[] row) {
    int n = row.length;
    int max = Integer.MIN_VALUE;

    for (int i = 0; i < n; i = i + 1)
        if (row[i] > max)
            max = row[i];

    return max;
}
```

```
public static int index(int[] row, int x) {
    int n = row.length;
    int i = 0;

    while (i != n && row[i] != x)
        i = i + 1;

    if (i == n)
        return -1;
    else
        return i;
}
```

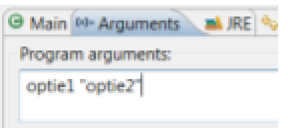
Recursion: Like in assembly, a method calling itself will create a separate stack frame for itself. Recursion can be dangerous because it can lead to a stack overflow if the stop condition is wrong or rarely acheived. Factorial recursion:

```
public static int factorial(int n){
        if (n==0 || n==1) {
        return 1; //basecase
    }
        return (n * factorial(--n)); //recursion i.e. 3 * f(2), f(2) hasn't been processed yet so it gets called.
} //once the basecase is reached, the program counter will go back to complete each unfinished multiplication.
```

Recap …

```
public static void main(String[] args)
```

- `public`: accessible to all
- `static`: works at the level of a class, not the object
- `void`: no return value
- `main`: entry point of the program
- `String[] args`: an array of Strings
  → but where does it come from?
  → user input that user enters when starting the program



Implementation "1 to many relation" (1)

```
public class Person {

    private String name;
    private Car[] cars;
    private int numberOfCars;

    public Person(String name)
    {
        this.cars = new Car[10];
        this.name = name;
        this.numberOfCars = 0;
    }

    public void addCar(Car car)
    {
        if(numberOfCars < cars.length)
        {
            cars[numberOfCars] = car;
            numberOfCars++;
        }
    }
    //...
}
```

Person 1 —— 1..* Car

Person
- name : String
+ Person(name : String)
+ addCar(car : Car) : void

Person 1 —— - car 1..* → Car

Car
- licenseplate : String
+ Car(license : String)

7

Implementation "1 to 1" relation

```
public class Person {

    private String name;
    private Car car;

    public Person(String name, Car car) {
        this.name = name;
        setCar(car);
    }

    public void setCar(Car car) {
        this.car = car;
    }

    //..
}
```

Perso n —— -this 1 —— -that 1 → Car

Summary of class ArrayList

ArrayList
+ArrayList<T>() post: an empty ArrayList was constructed with a capacity of 10 **+add(element: T)** post: has added an element at the end of the list **+get(index: int) : T** post: returns element at position index **+indexOf(element: T): int** post: returns 1st position at which element is found, if the element is not present -1 is returned **+size(): int** post: returns the number of elements in the list

An ArrayList is a **container** class that *contains* an array. This Java built-in container class comes with handy methods that allows us to add (push) elements without explicitly mentioning the index nor the array length, it has methods to get the index of an element, the size of the array, etc. It also allows you to add more elements once the size of the array has been reached by automatically extending the size of the array.

Implementation with ArrayList

```
import java.util.ArrayList;

public class Person {

    private String name;
    private ArrayList<Car> cars;

    public Person(String name)
    {
        this.cars = new ArrayList<Car>();
        this.name = name;
    }

    public void addCar(Car car)
    {
        this.cars.add(car);
    }

    public Car get(int i)
    {
        return this.cars.get(i);
    }

    //...
}
```

Perso n 1 → 1 ArrayList<Car> 1 → 0..* Car

11

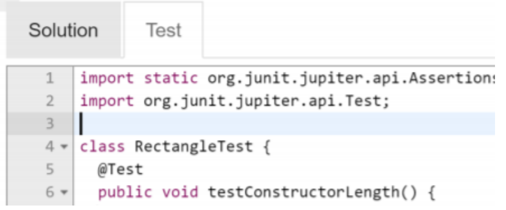
See how to add a car you just need to call the private attribute via this.cars and then type .add(car) to use the built-in ArrayList method. That's an easy way to add objects in an Array(List) and will most likely prevent you from (NullPointer) errors if you would have created the addCar method with basic arrays from scratch.

ArrayLists are expensive because it will often check the size of the array and copy the values to newly sized arrays but it is a constant cost.

UNIT TESTING

Theory question: Rocket crashed because of float conversion to integer (many bits of accuracy lost).
Good practice: Test Driven Development. Test each time you develop a new method instead of just testing at the end.
Unit Test: small test that tests the functionality of 1 class. Thumbrule, 1 test per method.

Testing with Weblab



JavaDoc Assert: https://junit.org/junit5/docs/current/api/index.html
Required for CSE1100:

1. Equals
2. NotEquals
3. True
4. False
5. Same
6. NotSame
7. Null
8. NotNull

Equals will first use the default or overriden **object.equals()** method whereas **Same** will use the **== operator** (useful if you want to check if they point to the same heap address). Each of these can include a 3rd string custom message and have an array version. Can also use primitives:
assertEquals(Object/primitive **expected**, Object/primitive **actual**):

```
assertArrayEquals(Object[] expected, Object[] actual)
assertArrayEquals(Object[] expected, Object[] actual, String message)
assertEquals(Object expected, Object actual)
assertEquals(Object expected, Object actual, String message)
assertFalse(boolean condition)
assertNotEquals(Object unexpected, Object actual)
assertNotSame(Object unexpected, Object actual)
assertNull(Object actual)
assertSame(Object expected, Object actual)
assertTrue(boolean condition)
```

```
@Test
public void testConstructorLength() {
  Rectangle rect = new Rectangle (4, 5);
  assertEquals(4, rect.getLength());
}
```

When the equals method has not been overriden it will use the default == operator.

Override = completetly overwrites the previously stablished method with that name and arguments.
Overload = add a new compatible verison of the same method that takes more and/or different arguments.

Integration tests: for interacting classes.

Rule of thumb: 1 assert per test method.; **Rule of thumb:** throw new IllegalArgumentException(); when pre: not met

Testing the unexpected

- Exceptions (see later) represent unlikely/unwanted scenarios. Also test whether you thought of implementing these.
- Example:

```
public static void swap(int[] seq){
 if (seq.length >= 2)    {
     // Do your swapping around
 }
 else {
    throw new IllegalArgumentException();
 }
}
```

Testing the unexpected (2)

```
@Test
public void swap_empty() {
    assertThrows(IllegalArgumentException.class,
() -> {
        Assignment2_1.swap(new int[]{});
    });
}
```

class

You're testing whether IllegalArgumentException() is thrown or not.

TUTORIAL 3

ARRAYS: FILL IN, MIN, CONTAINS

```
class ArraysTutorial {

  public static void run() {
    // Declare, initialize and bind an array with five spaces.
    int[] numbers = new int[5];

    // Fill the array with the numbers 1 through 5.
    for(int i = 0; i < numbers.length; i++) {
      numbers[i] = i + 1;
    }

    // Print the number in the 4th position.
    System.out.println(numbers[3]);

  }

  /**Find and return the lowest number in the input array.
   * @param numbers - an array that contains int values
   * @return the lowest number in the input array, or Integer.MAX_VALUE if the array has leng
th 0.
   */
  public static int min(int[] numbers) {
    int min = Integer.MAX_VALUE;
    for (int i = 0; i<numbers.length;i++){
      if (numbers[i]<min){
        min = numbers[i];
      }
    }
    return min;
  }

  /**Indicates whether a specific int value is present in the input array.
   * @param numbers - an array that contains int values
   * @param n - an int value
   * @return a boolean value that is true when n is present in numbers, else otherwise.
   */
  public static boolean contains(int[] numbers, int n) {
    for (int i = 0; i<numbers.length;i++){
      if (numbers[i]==n){
        return true;
      }
    }
    return false;
  }

}
```

ARRAYS: COUNT POSITIVES, FILTER POSITIVES

```java
class Filtering {

  /** Returns the number of positive int numbers in the array.
   * A number is positive when it is greater than 0.
   * @param numbers - the input array with numbers.
   * @return the number of positive numbers in numbers.
   */
  public static int countPositives(int[] numbers){
    int count = 0;
    for (int i = 0; i < numbers.length; i++){
      if (numbers[i]>0){
        count++;
      }
    }
    return count;
  }

  /** Filters the input array and returns an array that exactly contains the
   * positive numbers in the input array (without additional empty spaces).
   * @param numbers - the input array with numbers.
   * @return An array that contains only the positive numbers in numbers.
   */
  public static int[] filterPositives(int[] numbers){
    int n = 0;
    int[] temp = new int[numbers.length];
    for (int i = 0; i<numbers.length;i++){
      if (numbers[i]>0){
        temp[n]=numbers[i];
        n++;
      }
    }
    int[] result = new int[n];
    for (int i = 0; i<n;i++){
      result[i] = temp[i];
    }
    return result;
  }
}
```

TESTING: CONSTRUCTOR AND GETTERS

```java
package weblab;

class Car {

  private String licencePlate;
  private String brand;

  /**Initialize a new Car with licencePla
te `licencePlate` and brand `brand`
   * @param licencePlate - The licencePla
te of the car
   * @param brand - The brand of the car.
   */
  public Car(String licencePlate, String
brand) {
    this.licencePlate = licencePlate;
    this.brand = brand;
  }

  /**Getter for the licencePlate of the c
ar
   * @return the value of licencePlate
   */
  public String getLicencePlate() {
    return this.licencePlate;
  }

  /**Getter for the brand of the car
   * @return the value of brand
   */
  public String getBrand() {
    return this.brand;
  }

}
```

```java
package weblab;

// Notice the imports for the junit @Test annot
ation, and the asserts.
// The word "jupiter" indicates that we're deal
ing with JUnit 5.
// If your imports do not have this keyword, yo
u're probably using JUnit 4!
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.
*;

public class UTest {

  // @Test annotation to indicate the following
 method is a JUnit test case.
  @Test
  // Standard test definition "public void test
Name()"
  public void constructorTest() {
    // Setup the thing you want to test.
    // Here we're just calling the constructor
to create a new Car.
    Car car = new Car("12-AB-34", "Volvo");
    // Test whether the car is not "Null".
    // This doesn't tell us anything about the
fields that should have
    // been set by the constructor but, only th
at it did not crash.
    // For more information we need getters to
test (with)!
    assertNotNull(car);
  }

  @Test
  public void getLicencePlate() {
    Car car = new Car("12-AB-34", "Volvo");
    assertTrue(car.getLicencePlate().equals("12
-AB-34"));
  }

  @Test
  public void getBrand() {
    Car car = new Car("12-AB-34", "Volvo");
    assertTrue(car.getBrand().equals("Volvo"));
  }

}
```

```
package weblab;

// Notice the imports for the junit @Test annotation, and the asserts.
// The word "jupiter" indicates that we're dealing with JUnit 5.
// If your imports do not have this keyword, you're probably using JUnit 4!
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class UTest {
  // @Test annotation to indicate the following method is a JUnit test case.
  @Test
  // Standard test definition "public void testName()"
  public void constructorTest() {
    // Setup the thing you want to test.
    // Here we're just calling the constructor to create a new Car.
    Car car = new Car("12-AB-34", "Volvo");
    // Test whether the car is not "Null".
    // This doesn't tell us anything about the fields that should have
    // been set by the constructor but, only that it did not crash.
    // For more information we need getters to test (with)!
    assertNotNull(car);
  }

  @Test
  public void getLicencePlate() {
    Car car = new Car("12-AB-34", "Volvo");
    assertTrue(car.getLicencePlate().equals("12-AB-34"));
  }

  @Test
  public void getBrand() {
    Car car = new Car("12-AB-34", "Volvo");
    assertTrue(car.getBrand().equals("Volvo"));
  }

  @Test
  public void equals() {
    Car car1 = new Car("12-AB-34", "Volvo");
    Car car2 = new Car("12-AB-34", "Volvo");
    Car car3 = new Car("Not", "Same");
    Object car4 = new String ("");

    assertEquals(car1,car2);
    assertFalse(car1.equals(car3));
    assertFalse(car1.equals(car4));

  }

}
```

WEEK 4 – CONTAINER CLASSES & INHERITANCE

List = elements can be repeated, set = contents must be unique. But they both have almost identical same methods.

CONTAINER CLASS

Java comes with some default container classes, such us the ArrayList<*Ref Type*>. *Which exists for each data type*. It creates an automatically expanding array with built-in methods such as add, set, get, indexOf, contains, equals, toString.

Container classes are not restricted to just Lists and Sets of a single datatype. Container classes can be used as adhoc ***container*** classes to allow the creation of objects (such as Strings, object lists, object arrays) that contain a specific set of attributes and methods and it is not restricted to just primitives or to a single data or referefence type. Think of the "device" object that can be used in the different iPhone and iPad classes (without necessarily being a parent).

In the examples below an Int***List*** container class is made from scratch (although it already exists as ArrayList<Int>):

```
IntList
-----------------------------------------------
-elements: int[]
-number: int
-----------------------------------------------
+IntList(n: int)
pre: n > 0
post: an IntList object is created with number = 0 and
capacity = n

+add(el: int)
pre: number < capacity
post: if pre holds, el is placed at the first empty spot
in the Intlist, number = number + 1
```

Implementation of class IntList

```java
public class IntList {
    private int[] elements;
    private int number;

    public IntList(int n) {
        elements = new int[n];
        number = 0;
    }
}
```

Implementation of method add()

```java
public void add(int el) {
    if (number < elements.length) {
        elements[number] = el;
        number = number + 1;
    }
}
```

As you can see the main difference is that IntList requires an explicit list length and it is bound to that capacity constraint.

```
+get(i: int): int
pre: 0 <= i < number
post: if pre holds, returns elements[i]; otherwise
Integer.MIN_VALUE is returned

+index(el: int): int
post: returns the 1st position at which el is found,
returns -1 if el is not found

+contains(el: int): boolean
post: returns true if el is present in IntList
```

Implementation of method get()

```java
public int get(int i) {
    if (0 <= i && i < number)
        return elements[i];
    else
        return Integer.MIN_VALUE;
}
```

Conversely, instead of Integer.MIN_VALUE you could also throw an IllegalArgumentExceptionError()

Implementation of method index()

```java
public int index(int el) {
    int i = 0;
    while (i < number && elements[i] != el)
        i = i + 1;

    if (i == number)
        return -1;
    else
        return i;
}
```

Implementation of contains()

```java
public boolean contains(int el) {
    return index(el) != -1;
}
```

```
+getSize(): int
post: returns number

+set(i: int, el: int)
pre: 0 <= i < number
post: if pre holds, elements[i] is set to el

+toString() : String
post: returns a String representation of the IntList

+equals(other: Object) : boolean
post: returns true if other is of type IntList, this and
other contain an equal number of elements and the
respective elements of this and other are equal
```

Implementation of toString()

```java
public String toString() {
    String result = "<IntList[";
    for (int i = 0; i < number; i++) {
        result = result + elements[i];
        if (i < number - 1) {
            result = result + ",";
        }
    }

    result = result + "]>";
    return result;
}
```

→ example result: `<IntList[4,88,2,14]>`

Implementation of getSize() and set()

```java
public int getSize() {
    return number;
}

public void set(int i, int el) {
    if (0 <= i && i < number)
        elements[i] = el;
}
```

Implementation of equals()

```java
public boolean equals(Object other) {
    boolean result = false;
    if (other instanceof IntList) {
        IntList that = (IntList) other;
        if (this.number == that.number) {
            int i = 0;
            while (i < number
                    && this.elements[i] == that.elements[i]) {
                i = i + 1;
            }
            result = (i == number);
        }
    }
    return result;
}
```

Tip: start typing 'public' on intellij and you can auto fill a boolean equals method!

Mathematical sets

In a set, elements are distinct (no duplicates).
The order of the elements is irrelevant.
Some operations on sets:

Constructor empty set:	`{}`
Constructor set with 1 element:	`{el}`
Union of 2 sets:	`{el1} ∪ {el2}`
Intersection of 2 sets:	`{el1,el2} ∩ {el2}`
Membership:	`el1 ε {el1,el2}`

```
+union(that: IntSet): IntSet
post: returns an IntSet that contains the elements of this
and that

+intersection(that: IntSet): IntSet
post: returns an IntSet with elements that are contained
in both this and that

+equals(other: Object) : boolean
post: returns true if other is of type IntSet, this and
other contain an equal number of elements and the
respective elements of this are equal to those in other
```

```
IntSet
-elements: int[]
-number: int
+IntSet(n: int)
pre: n > 0
post: an IntSet object is created with number = 0 and
capacity = n

+add(el: int)
pre: number < capacity, el is not present in IntSet
post: if pre holds, el is placed at the first empty spot
in the IntSet, number = number + 1
```

```java
public class IntSet {
    private int[] elements;
    private int number;

    public IntSet(int n) {
        elements = new int[n];
        number = 0;
    }
}
```

```java
public void add(int el) {
    if (!contains(el))
        if (number < elements.length) {
            elements[number] = el;
            number = number + 1;
        }
}
```

(Exactly same constructor). Can only add new elements. Add, and equals are thus slightly different.
union and intersection are set specific methods. There's no Java built-in container class for sets.

Implementation of method union()

```
public IntSet union(IntSet that) {
    int n = this.number + that.number;
    IntSet result = new IntSet(n);

    for (int i = 0; i < this.number; i++)
        result.add(this.elements[i]);

    for (int i = 0; i < that.number; i++)
        result.add(that.elements[i]);
    return result;
}
```

Can we actually access a private member? Yes, because type (IntSet) is the same.

Method intersection()

```
public IntSet intersection(IntSet that) {
    int n = Math.min(this.number, that.number);
    IntSet result = new IntSet(n);

    for (int i = 0; i < this.number; i++)
        if (that.contains(this.elements[i]))
            result.add(this.elements[i]);

    return result;
}
```

(equals for sets)

Implementation of equals()

```
public boolean equals(Object other) {
    boolean res = false;
    if (other instanceof IntSet) {
        IntSet that = (IntSet) other;

        if (this.number == that.number) {
            IntSet temp = this.intersection(that);
            res = (temp.number == this.number);
        }
    }
    return res;
}
```

CONTENTS OF AN ARRAYLIST

Going back to ArrayList<Ref Type> Ref type stands for an object of reference type. ArrayList allows the use of int type via the Integer object (so-called "autoboxing"). The mayor **difference between an int and an Integer** is that although both are used to store *integers (as in its mathematical definition)* of 32 bits, signed, two's compelment, **int is a primitive so it holds the stack addres that contains the value** of such int **whereas Integer is a wrapper class which wraps a primitive type int into an object**.

INHERITANCE

What is the problem?

- A **teacher** has a name and an address
- A **student** also has a name and an address

→ Both are special cases of a **person**

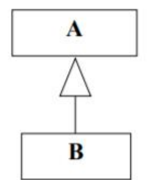
- Inheritance makes it possible to:
  1. Define relationships between concepts (classes)
  2. Enables **reuse**
  3. You can create variants of a concept and still approach/use them in the same way

So what is inheritance?

- A relationship between two classes A and B:
  – B inherits from A
  – B is a subclass of A
  – B is derived from A
  – A is a superclass of B
- This relationship is transitive
- This relationship does not allow cycles
- It is a hierarchy

Inheritance in UML

- Class B inherits from class A
- Class A is the parent of B
- Class B is the child of A
- Class A is the superclass of B
- Class B is the subclass of A



* UML = Unified Modelling Language

What does inheritance mean?

- An object of a subclass has all the properties of the superclass
- On such an object, you can call all (public) methods of the superclass.
- (**Liskov Substitution Principle**) If A is a superclass of B, then at each location in the program where an object of type A is requested, an object of type B is also acceptable.

→ `list.addPerson(new Student());`
→ `list.addPerson(new Person());`

Construction

```
class Person{
   private String name;

   public Person(String nm){
      name = nm;
   }
   ...
}
class Student extends Person{
   private int studentnumber;

   public Student(int number, String nm){
      ...
   }
   ...
}
```

Access Control Rules

- public access is global.
- private access is only within the class
  - Subclasses have no rights!
- package access is the whole package.
  - This is used when you don't specify!
- protected access is package plus subclasses.

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

Usually you don't want to let the Parent class to use the more specialized methods that the childclass has since most likely the "Person" is not necesasrily "enrolled" to a course as most persons are not students. On the other hand, a student shall have all the properties of a person, therefore children ***inherit*** the attributes and methods of the parent.

The parent must however explicitly declare which methods and attributes are ***inheritable***, namely those with the modifiers **"public" and "protected"**, only the latter being exclusive to the class, package and subclasses (and not the whole world).

By default *(no modifier)* if you "pack" compatible classes into a same package it is allowed for **classes within a package to be able to borrow methods from one another** when these have no access control keywords (*modifiers*). Because these havent been explcitly declared as "inerhitable", these won't be inherited to the children classes.

private keyword makes it **exclusive to the class**.

A parent will never be able to use a children method or attribute unless these are set to public.

Packages (in other programming languages: namespaces)

- Declare as:
  - `package nl.tudelft.andyzaidman;`
- Use as:
  - `import nl.tudelft.andyzaidman.*;` → import entire package
  - `import nl.tudelft.andyzaidman.myClass;` → import 1 class
- Why?
  - Grouping pieces of code (classes) that are related
  - Prevent "name-collides", you want to implement a "special" class Date, while Java already has a standard Date implemented…

Construction

```
class Person {
   public Person(String nm){
      name = nm;
   }
   ...
}

class Student extends Person{
   public Student(int number, String nm){
      super(nm);
      studentnumber = number;
   }
   ...
}
```

super() is a call to the constructor of the superclass

```
class Person {

   public Person() {}

   public Person(String nm){
      name = nm;
   }
   ...
}

class Student extends Person{
   public Student(int number, String nm){
      studentnumber = number;
   }
   ...
}
```

Will compile…

In case you forgot or you're likely to rename the parent, you can always call a method of the parent by using the super prefix. Just using super(); will call the constructor. If you don't call it at all, it will automatically do so. If you call it, it must be the first statement inside the child constructor. Be sure to either provide a default constructor in the parent that takes no arguments or to explicitly call the non-default parent constructor from the child constructor.

Default construction order

1. Call to the constructor of the super class
2. Initialisation of the attributes
3. Body of the constructor of the derived class

Typechecker reference types

- Safe conversion (coercion)
- Unsafe conversion (cast)
- Impossible conversion
- Declared/actual type
- What does the compiler want?

Suppose: SavingsAccount is a subclass of Account

```
Account a1 = new Account(…);                  //allowed
Account a2 = new SavingsAccount(…);           //allowed, conversion

SavingsAccount s = new Account();             //not allowed

SavingsAccount s = (SavingsAccount)  a2;  //allowed
SavingsAccount s = (SavingsAccount)  a1;  //ClassCastException

Button b = (Button) a1;                       //impossible!
```

Parent: Account <|- -------]
Children: Savings Account |

Liskov principle: **Anywhere a parent is expected, a children is accepted, but not they otherway around**.

Declared vs. actual type

- A variable has a **declared (or static)** type
- An object has an **actual (or dynamic)** type

Conversion (dynamic) (Liskov coercion): You can declare an object a1 as a Parent type and assign it to a children memory location during initalisation. Account a1 = new SavingsAccount();

(Declared) Actual type: you can ofcourse declare an object a1 as a Parent and assign it to a parent memory location. Account a1 = new Account();

Unsafe (dynamic) conversion (cast): Assign declared object to a different object casted as the same. The compiler will check if the different object and the declared object have any sort of (parent-child) relation. If so it'll give it a try and let it check at run time if the different object was converted at some point to the same object before the cast.

Impossible (dynamic) conversion (ClassCastException): Same as above, you'll manage to fool the compiler to trust you that the different object will be converted to an acceptable form but at runtime such casted object turned out to not have been converted to a "castable" same object.

```
Object str = new String ( original: "Text");

String s1 = str;

System.out.println(s1);
```

```
C:\Users\sergi\OneDrive - Delft University of Technology\CSE\Y1\Q1\CSE1100 Object Or
java: incompatible types: java.lang.Object cannot be converted to java.lang.String
```

Type conversion

- Primitive types:
  - Change, convert
- Reference types
  - "interpret as"

String s1 = (String) str; //would work

Object **instanceof** Class: expression that will return a boolean true if the object is an instance of the class.
Thumbrule: dont create a "getType()" method and use the built-in instance of operator.
Thumbrule: It is OK to redefine methods, use dynamic binding (conversion, casting), to use super. **But don't redefine attributes of the parent class.** They are not inteded to be overriden.

Polymorphism

The concept that different classes can be used with the same interface. Each of these classes can provide its own implementation of the interface.

```
Animals[] zoo = new Animals[10];
zoo[0] = new Cat();
zoo[1] = new Dog();

zoo[0].makeSomeNoise();      // "miauw"
zoo[1].makeSomeNoise();      // "woef"
```

Same interface, different implementation!

Redefining methods

"Method overriding"

- Methods have same signature
  - Are located in another class of the inheritance-hierarchy
  - Have a different "behaviour"
- … if not, "**method overloading**"!!!
  - Same method name, different parameter list

That is why even if we create our own custom lists and set, it is good practice to keep using the same standard methods and its standard names (add, contains, set, get, equals, etc.) In fact, UnitTesting will rely on the object.equals() method.

```
public class SimpleCode {
   public char encode (char token) {
      return (char) (token + 1);
   }
}

public class SmartCode extends SimpleCode {
   public char encode (char token) {
      return encode(token, 3);
   }

   public char encode(char token, int key) {
      return (char) (token + key);
   }
}
```

overriding

overloading

Dynamic binding

- Binding: what does this word mean?
  - Association of a method definition to the method call
- Static binding: the compiler decides
  - Some information is not available at the time of compilation (see slide What can the compiler know?)
- Dynamic binding: the virtual machine (JVM) decides

Example binding (2)

- `SimpleCode code = new SimpleCode();`
  `char c = code.encode('B');`
  - Accepted by compiler, no errors, value: 'C'
- `SimpleCode code = new SmartCode();`
  `char c = code.encode('C');`
  - Accepted by compiler, no errors, value: 'F'
- `SimpleCode code = new SmartCode();`
  `char c = code.encode('D', 4);`
  - Not accepted, why?
  - `char c = ((SmartCode) code).encode('D', 4)` would give 'H'
- The compiler looks: does the static type of code (SimpleCode) have a method `encode(char token, int key)`?
- No… and Java doesn't know for sure whether code's dynamic type is SimpleCode or SmartCode… no risks taken, so not possible!

- `SimpleCode code = new SimpleCode();`
  `char c = code.encode('B');`
  - Static and dynamic type of code are the same, namely SimpleCode
  - `encode(char token)` of SimpleCode is called
- `SimpleCode code = new SmartCode();`
  `char c = code.encode('C');`
  - Static type code: SimpleCode
  - Dynamic type code: SmartCode
    - Liskov Substitution Principle: everywhere where an instance of a base class is expected, an instance of a subclass is also acceptable
  - Does SmartCode have an implementation of `encode(char c)`?
    - If yes, use that one
    - If no, rely on `encode(char c)` of SimpleCode

48

Every class is an Object

- Even if you don't make it explicit, each class you create is a subclass of Object

  (have a look at the Javadoc of a self-created class without explicit superclass)

java.lang
Class String

java.lang.Object
└ java.lang.String

All Implemented Interfaces:
Serializable, CharSequence, Comparable<String>

The equals method in Object

```java
public boolean equals(Object obj){
  return (this == obj);
}
```

SO: if you create a class without an equals(), Java will rely on the implementation of the superclass… and that could very well be the equals() of Object

→ The equals() of Object compares memory addresses!

Prevent/forbid redefinition

- Keyword: `final`
  - We used it earlier for attributes… (they became constants)
- `public final class String`
  - You cannot create a subclass of String!
- `public final void addElement(Object obj)`
  - You cannot redefine addElement in a subclass!!

The abstract class Shape

Abstract class = you cannot create an object from this class

```java
public abstract class Shape {
  public abstract double volume();
  // value: the volume of the specific shape

  public abstract String toString();
  // value: a specific string-representation
  // of the  specific shape

}
```

Abstract method = if you extend from this class, you **are obliged** to implement the method

56

Abstract class / abstract method

- An abstract class cannot be instantiated
- If a class contains an abstract method, the class should be abstract
- An abstract class can also contain non-abstract methods (e.g. for shared functionality that is useful for the child classes)
- An abstract class can contain attributes

- An abstract method has no body
- An abstract method *must* be implemented by its children (unless they are abstract too)

Shape as interface

```java
interface Shape {
   double volume();
   String toString();
}
```

```java
public abstract class Shape {
   public abstract double volume();
   public abstract String toString();
}
```

The abstract method of an abstract superclass *must* be implemented in the (immediate) subclass!

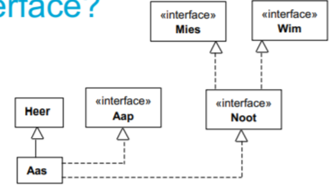
Syntax and semantics

```java
public class Sphere implements Shape {
    ...
}
```

means:

the class Sphere *is obligated* to follow the contract of Shape (i.e. implementing `volume()` and `toString()`)

Why interface?



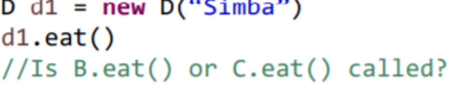
Java does not allow **multiple inheritance**
- You can inherit from 1 class
- And implement multiple interfaces

Why doesn't Java allow multiple inheritance?

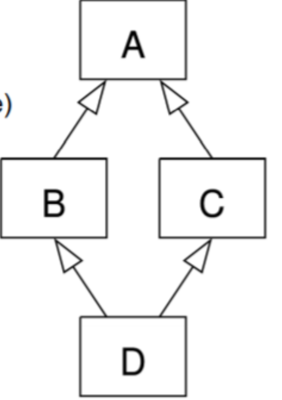
A children "extends" the parent an interface children "implements" the abstract parent

Issues with multiple inheritance

- A is Animal
- B is a Mammal
- C is a Carnivore
- D is a Lion (both mammal and carnivore)



```
D d1 = new D("Simba")
d1.eat()
//Is B.eat() or C.eat() called?
```



Interface since Java 8 (1)

As of Java 8, interfaces can have:

- A default method

```
public interface DoIt {
    default boolean didItWork() {
        // Method body
    }
}
```

→ You can provide implementation!

(Main goal: make sure you can add something to an interface while it has already been implemented by many other classes, while not forcing the other classes to add an implementation)

The keyword **static**

- Attributes and methods can be **static**
- What does the static keyword mean?
  - It is a class-attribute or class-method
  - Only 1 such attribute exists, for all instances (objects) of the class
  - A static method works on such an attribute. A static method can be called through the class (instead of through an instance → still possible, but Java will warn!)

Static: how to use?

```
public void someMethod() {
  Savingsaccount s1 = new SavingsAccount("Andy", …)

  SavingsAccount.setInterest(3.10);
  double interest = SavingsAccount.getInterest();

  interest = s1.getInterest();
```

Will work, but will lead to compiler warning.

Rule of thumb: Java doesn't like to guess. If there's a situation where Java will have to guess, it will most likely throw an error.

Abstract class / interface → differences

- An abstract class *can* contain attributes
- An abstract class *can* contain implementations for methods
  - Useful to put an implementation there that is valid for all/some subclasses

- An interface cannot contain attributes
- An interface cannot contain method implementations

In a sense an interface is even more abstract as in restricted

Interface since Java 8 (1)

As of Java 8, interfaces can have:

- Static methods with implementation
  → similar to static methods in classes
- Static final attributes, i.e., constants at "class level"

```
public interface DoIt {
    int k = 1;
}
```

→ You could read **int k = 1** as a **public static final int k = 1**

Static: how to define?

```
public class SavingsAccount {
    static private double fInterest;
    private String fName;

    public SavingsAccount(…) {
      // initialisation
    }
    public static void setInterest(double interest){
      fInterest = interest;
    }

    public static double getInterest() {
      return fInterest;
    }
}
```

TUTORIAL 4 – CONTAINER CLASS (CARS SHOWROOM WITH CARS[] ARRAY AND AMOUNT)

In this assignment we're going to create a container class for `Car`: were going to create `Showroom` a class that keeps track of a list of cars in a showroom.
The implementation for the class `Car` is provided. You can see the class by clicking on the *library* tab in this assignment!

- **Implement the following:**

A `class Showroom` (not public) with two private fields:

- An array of Cars `cars`.
- An `int amount` that keeps track of the number of cars in `cars`.

`public Showroom(int size)`
Post: Created a showroom with a list of cars with size `size`, and set amount to `0`, as the showroom is empty.

`public int getShowroomSize()`
Post: returns the length of `cars`.

`public Car getCar(int i)`
Pre: `i < amount`. If this evaluates to `false` the position is empty and the method should throw an `IllegalArgumentException`.
Post: returns the `car` on position `i` of `cars`.

`public int getAmount()`
Post: returns `amount`.

`public void addCar(Car car)`
Pre: `amount < cars.length`. If this evaluates to `false` the showroom is full and the method should throw an `IllegalArgumentException`.
Post: `car` is added to `cars` on the first free position and `amount` is incremented by one.

LIBRARY CONTENTS:

```java
package weblab;

class Car {

  private String licencePlate;
  private String brand;

  public Car(String licencePlate, String br
and) {
    this.licencePlate = licencePlate;
    this.brand = brand;
  }

  public String getLicencePlate() {
    return this.licencePlate;
  }

  public String getBrand() {
    return this.brand;
  }

  public boolean equals(Object other) {
    if(other instanceof Car) {
      Car that = (Car) other;

      return this.licencePlate.equals(that.
licencePlate) &&
        this.brand.equals(that.brand);
    }
    return false;
  }

}
```

SOLUTION

```
package weblab;

class Showroom {

  private Car[] cars;
  private int amount;

  public Showroom(int size){
      cars = new Car[size];
      amount = 0;
  }

  public int getShowroomSize(){
      return cars.length;
  }

  public Car getCar(int i){
      if (i>= amount) throw new IllegalArgumentException();
      return cars[i];
  }

  public int getAmount(){
      return amount;
  }

  public void addCar(Car car){
      if (!(car instanceof Car)) return;
      if (amount >= cars.length) throw new IllegalArgumentException();
      cars[amount] = car;
      amount++;
  }

}
```

TEST

```
package weblab;

import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.BeforeEach;

public class UTest {

    private Showroom myShowroom;
    private Car realCar1;
    private Car realCar2;

    @BeforeEach
    void setUp() {
    myShowroom = new Showroom(5);
    realCar1 = new Car("LicensePlate123","Mercedes");
    realCar2 = new Car("LicensePlate124","Ford");
    }


    @Test
    void getShowroomSize(){
        assertEquals(5,myShowroom.getShowroomSize(), "getShowroomSize()");
    }

    @Test
    void getAmount(){
        assertEquals(0,myShowroom.getAmount(), "getAmount()");
    }

    @Test
    void addCar(){

        myShowroom.addCar(realCar1);
        myShowroom.addCar(realCar2);
        assertEquals(2,myShowroom.getAmount(),"addCar(Car car)");
    }

    @Test
    void getCar(){
        myShowroom.addCar(realCar1);
        myShowroom.addCar(realCar2);
        assertEquals(realCar2,myShowroom.getCar(1), "getCar(int i)");
    }

}
```

TUTORIAL 4 - ARRAYLIST

Up to this point you've been implementing *lists* in Java using Arrays in Java. As you may have noticed this is not always very convenient, as you need to know the exact size of the Array ahead of time, or else you may hit the limit. Keeping track of the elements in an Array, and creating a container class for every object type is not trivial either.

Luckily Java has something that can help: `ArrayList`. An `ArrayList` is essentially an all purpose wrapper for an array.
You can create an `ArrayList` by like so:

```
ArrayList<String> myList = new ArrayList<>();
```

This `ArrayList` can hold objects of type `String`, but we can replace `String` with any other reference type we want. This is the power of *generics* (this will be discussed more extensively in a later lecture).

PUTTING IT TO USE

Now, to use an `ArrayList` we need to import it. We can do this by defining the import:

```
import java.util.ArrayList;
```

`ArrayList` has several benefits over an array from a user perspective: it grows automatically, and there are many methods available you can use out of the box.

Go to the Java documentation here and find the documentation for ArrayList. Now implement the class for this assignment and the methods avaible in the `ArrayList` class to make this as easy as possible.

IMPLEMENT THE FOLLOWING

A `class CarCatalog` that has one field: an `ArrayList` of `Car` named catalog.
The implementation for the class `Car` is provided. You can see the class by clicking on the *library* tab in this assignment!

`public CarCatalog()`
Post: Created a car catalog with an `ArrayList` of cars.

`public int getCatalogSize()`
Post: returns the size of `catalog`.

`public ArrayList<Car> getCatalog()`
Post: returns the `catalog`.

`public void addCar(Car car)`
Post: added `car` to `catalog`.

`public boolean contains(Car car)`
Post: returns `true` if `catalog` contains `car`, returns `false` otherwise.

TESTING

Once you're done, don't forget to write at least one test for each method you've implemented!

SOLUTION

```java
package weblab;

import java.util.ArrayList;

class CarCatalog {

    private ArrayList<Car> catalog;

  /** Constructor for CarCatalog, initializes a new CarCatalog.
   */
  public CarCatalog() {
    catalog = new ArrayList<Car>();
  }

  /**Getter for catalog.
   * @return the catalog ArrayList.
   */
  public ArrayList<Car> getCatalog() {
    return catalog;
  }

  /**Getter for the size of the catalog.
   * @return the size of the catalog ArrayList.
   */
  public int getCatalogSize() {
    return catalog.size();
  }

  /**Adds a car to the catalog, duplicates are allowed.
   * @param car - the car that needs to be added to the catalog.
   */
  public void addCar(Car car) {
    catalog.add(car);
  }

  /**Checks whether the catalog contains a specific car.
   * @param car - the car that needs to be in the catalog.
   * @return "true" if car is contained in the catalog,
   * returns "false" otherwise.
   */
  public boolean contains(Car car) {
    return catalog.contains(car);
  }
}
```

TEST

```
package weblab;

import java.util.ArrayList;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.BeforeEach;

public class UTest {

  private CarCatalog catalog1;
  private CarCatalog nullCatalog;
  private Car car;

  @BeforeEach
  void setUp(){
      catalog1 = new CarCatalog();
      car = new Car("License123","Mercedes");
  }

  @Test //1st Constructor
  void CarCatalog(){
      assertNotNull(catalog1, "CarCatalog()"); //I like to explictly mention the exact name of the method
      assertTrue(catalog1 instanceof CarCatalog, "CarCatalog()"); //easier to find if test has different name
  }

  @Test //2nd Getters
  void getCatalog(){
      assertTrue(catalog1.getCatalog() instanceof ArrayList, "getCatalog()"); //make sure to
import java.util.ArrayList;
  }

  @Test //3rd counters
  void getCatalogSize(){
      assertEquals(0,catalog1.getCatalogSize(),"getCatalogSize()");
  }

  @Test //4th adders/setters/changers
  void addCar(){
      catalog1.addCar(car);
      catalog1.addCar(car);
      catalog1.addCar(car);
      assertEquals(3,catalog1.getCatalogSize(),"addCar(Car car)");
  }

  @Test //5th checkers
  void contains(){
      catalog1.addCar(car);
      assertTrue(catalog1.contains(car),"contains(Car car)");
  }

}
```

```java
package weblab;

class Showroom {

  private Car[] cars;
  private int amount;

  /**Initialize a new CarDatabase with size `size`.
  *  Set the initial amount of cars in the database to 0.
  *  @param size - The size of the CarDatabase.
  */
  public Showroom(int size) {
    this.cars = new Car[size];
    this.amount = 0;
  }

  /**If the cardatabase is not full add a car to the database and increase the amount of cars
  *  If the cardatabase is full throw an exception.
  *  @param c - A car to add to the database
  */
  public void addCar(Car c) {
    if( amount < cars.length ){
        cars[amount] = c;
        amount++;
    } else {
      throw new IllegalArgumentException("The showroom is full!");
    }
  }

  /**Count the number of cars in the database that match the provided brand
  *  @param brand - the brand that should be compared to that of the cars in the database
  *  @return the number of cars that have the same brand as the provided brand
  */
  public int brandCount(String brand) {
    int count = 0;
    for(int i = 0; i < amount; i++) {
      if(cars[i].getBrand().equals(brand)){
        count++;
      }
    }
    return count;
  }

  /**Gets the amount of cars in the showroom
   * @return the amount of cars in the showroom
   */
  public int getAmount() {
    return amount;
  }

  /**Gets the size of the cars array
   * @return the number of cars that can fit in the showroom

  public int getCarsSize() {
    return cars.length;
  }
}
```

TUTORIAL 4 – CAR SHOWROOM TESTS

```
package weblab;

import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.BeforeEach;

public class UTest {

    private Showroom myShowroom;
    private int showroomSize;
    private Car Mercedes;

    @BeforeEach
    void setUp(){
        showroomSize = 3;
        myShowroom = new Showroom(showroomSize);
        Mercedes = new Car("License123","Mercedes");
    }
    @Test //1 Constructor
    void Showroom(){
        assertNotNull(myShowroom,"Showroom()");
        assertTrue(myShowroom instanceof Showroom);
    }
    @Test //2 Getters
    void getAmount(){
        assertEquals(0,myShowroom.getAmount(),"getAmount()");
    }
    @Test //3 Counters
    void getCarsSize(){
        assertEquals(showroomSize,myShowroom.getCarsSize(),"getCarsSize()");
    }
    @Test //4 Changers
    void addCar(){
        myShowroom.addCar(Mercedes);
        assertEquals(1,myShowroom.getAmount(),"addCar()");
    }
    @Test
    void addCarException(){
        myShowroom.addCar(Mercedes);//1
        myShowroom.addCar(Mercedes);//2
        myShowroom.addCar(Mercedes);//3
        assertThrows(IllegalArgumentException.class,
        () -> {
            myShowroom.addCar(Mercedes);// 4 > 3
        }
        ,"addCar()");
    }
    @Test //5 Checkers
    void brandCount(){
        myShowroom.addCar(Mercedes);
        myShowroom.addCar(Mercedes);
        assertEquals(2,myShowroom.brandCount("Mercedes"),"brandCount(String brand)");
    }
}
```

ASSIGNMENT 2.1: ARRAYS

In this assignment, define multiple static methods according to the specifications below. Make sure to reuse methods wherever you can (i.e. avoid implementing the same thing twice). **Remember: if a precondition referring to a parameter fails you should work with an `IllegalArgumentException`.**

Keep your code neatly formatted, and make sure you write comments!

`public static void println(int[] seq)`
Post: Print the values in `seq` on 1 line, separated by a space. After that the cursor should be on the next line

`println(new int[]{1, 2})` → `1 2`

`public static void swap(int[] seq)`
Pre: `seq` has 2 or more values.
Post: The values at position 0 and 1 are swapped.

`swap(new int[]{1, 2})` → `2 1`

`public static int[] copy(int[] seq)`
Post: Returns a `result` with the same values as `seq`.

`result != seq` → `true` (this means that the memory address of `result` and `seq` should not be the same, but the values in the arrays should be the same!)

`public static void rotate(int[] seq)`
Pre: `seq` has 1 or more values.
Post: The values are rotated one to the right; the last element is at position 0.

`rotate(new int[]{1, 2, 3})` → `3 1 2`

`public static void rotate(int[] seq, int n)`
Pre: `seq` has 1 or more values and `n > 0`.
Post: The values are rotated `n` times to the right.
`rotate(new int[]{1, 2, 3}, 3)` → `1 2 3`

SOLUTION

```
package weblab;

class Assignment2_1 {

    public static void println(int[] seq) {
      int i = 1;
      String list = String.valueOf(seq[0]);
      do {
        list += " " + seq[i];
        i++;
      } while (i<seq.length);
      System.out.println(list);
    }

    static void swap(int[] seq) {
      if (seq.length >=2){
      int temp = seq[0];
      seq [0] = seq[1];
      seq [1] = temp;
      } else {
        throw new IllegalArgumentException();
      }
    }

    static int[] copy(int[] seq) {
        int[] result = new int[seq.length];
        for (int i = 0; i<seq.length; i++){
          result[i] = seq[i];
        }
        return result;
    }

    public static void rotate(int[] seq) {
      if(seq.length<1){
        throw new IllegalArgumentException();
      }
      int[] temp = new int[seq.length];
      temp[0] = seq[(seq.length - 1)];

      for (int i=1;i<seq.length;i++){
        temp[i] = seq[(i-1)];
      }

      for (int i=0;i<seq.length;i++){
        seq[i] = temp[i];
      }
    }

    public static void rotate(int[] seq, int n) {
      if (seq.length<1 || n<=0){
        throw new IllegalArgumentException();
      }

      int[] temp = new int[seq.length];
      for (int i=0; i<n; i++){
        temp[i] = seq[(seq.length-n+i)];
      }

      for (int i=n;i<seq.length;i++){
        temp[i] = seq[(i-n)];
      }

      for (int i=0;i<seq.length;i++){
        seq[i] = temp[i];
      }
    }
}
```

TEST

```java
package weblab;

import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.ass
ertArrayEquals;
import static org.junit.jupiter.api.Assertions.ass
ertNotEquals;
import static org.junit.jupiter.api.Assertions.ass
ertThrows;

class Assignment2_1Test {

    @Test
    void swapEmpty() {
        assertThrows(IllegalArgumentException.clas
s, () -> {
            Assignment2_1.swap(new int[]{});
        });
    }

    @Test
    void swapTwoElements() {
        int[] array = new int[]{1, 2};
        Assignment2_1.swap(array);
        assertArrayEquals(new int[]{2, 1}, array,
"First two elements must be swapped");
    }

    @Test
    void swapManyElements() {
        int[] array = new int[]{5, 27, 91, 36, 2};
        Assignment2_1.swap(array);
        assertArrayEquals(new int[]{27, 5, 91, 36,
 2}, array, "First two elements must be swapped");
    }

    @Test
    void copyShouldReturnDifferentArray() {
        int[] array = new int[]{};
        assertNotEquals(array, Assignment2_1.copy(
array), "Array should be copied");
    }

    @Test
    void copyEmpty() {
        int[] array = new int[]{};
        assertArrayEquals(new int[]{}, Assignment2
_1.copy(array), "Array should be empty");
    }

    @Test
    void copyManyElements() {
        int[] array = new int[]{6, 3, 7, 10, 52};
        assertArrayEquals(new int[]{6, 3, 7, 10, 5
2}, Assignment2_1.copy(array), "Values in array sh
ould be copied");
    }

    @Test
    void rotateEmpty() {
        assertThrows(IllegalArgumentException.clas
s, () -> {
            Assignment2_1.rotate(new int[]{});
        });
    }

    @Test
    void rotateOnce() {
        int[] array = new int[]{3, 6, 3, 9, 1, 0};
        Assignment2_1.rotate(array);
        assertArrayEquals(new int[]{0, 3, 6, 3, 9,
 1}, array, "Values should be shifted one place to
 the right");
    }

    @Test
    void rotateTwice() {
        int[] array = new int[]{3, 6, 3, 9, 1, 0};
        Assignment2_1.rotate(array);
        Assignment2_1.rotate(array);
        assertArrayEquals(new int[]{1, 0, 3, 6, 3,
 9}, array, "Values should be shifted two places t
o the right");
    }

    @Test
    void rotateOnceWithN() {
        int[] array = new int[]{3, 6, 3, 9, 1, 0};
        Assignment2_1.rotate(array, 1);
        assertArrayEquals(new int[]{0, 3, 6, 3, 9,
 1}, array, "Values should be shifted one place to
 the right");
    }

    @Test
    void rotateThriceWithN() {
        int[] array = new int[]{3, 6, 3, 9, 1, 0};
        Assignment2_1.rotate(array, 3);
        assertArrayEquals(new int[]{9, 1, 0, 3, 6,
 3}, array, "Values should be shifted three places
 to the right");
    }

    @Test
    void rotateMinusOneWithN() {
        int[] array = new int[]{3, 6, 3, 9, 1, 0};

        assertThrows(IllegalArgumentException.clas
s, () -> {
            Assignment2_1.rotate(array, -1);
        });
    }

    @Test
    void rotateEmptyWithN() {
        assertThrows(IllegalArgumentException.clas
s, () -> {
            Assignment2_1.rotate(new int[]{}, 2);
        });
    }

}
```

ASSIGNMENT 2.2: PRIME NUMBERS

In this assignment, define multiple static methods according to the specifications below. Make sure to reuse methods wherever you can (i.e. avoid implementing the same thing twice). **Remember: if a precondition referring to a parameter fails you should work with an `IllegalArgumentException`.**

`public static int max(int[] seq)`
Post: Returns the maximum number in `seq`, or `Integer.MIN_VALUE` if `seq` has 0 values.

`max(new int[]{1, 2})` → `2`

`public static int index(int[] seq, int el)`
Post: Returns the first index which has the same value as `el`, or `-1` if no such value exists.

`index(new int[]{3, 1, 2}, 2)` → `2`

`public static boolean contains(int[] seq, int el)`
Post: If the value `el` is in `seq` return `true`, else `false`.

`contains(new int[]{1, 2}, 2)` → `true`

`public static boolean isPrime(int el)`
Pre: `el > 1`
Post: If `el` is a prime return `true`, else `false`.

`isPrime(11)` → `true`

`public static int countPrimes(int[] seq)`
Post: Returns the number of primes in `seq`.

`countPrimes(new int[]{4, 11, 11})` → `2`

`public static int[] primesIn(int[] seq)`
Post: Returns a new array with all primes in `seq`.

`primesIn(new int[]{4, 11, 11})` → `[11, 11]`

`public static int[] primesUpTo(int n)`
Post: Returns an array with all primes up to, but excluding, `n`.

`primesUpto(14)` → `[2, 3, 5, 7, 11, 13]`

SOLUTION

```
package weblab;
class Assignment2_2 {
    public static int max(int[] seq) {
      int max = Integer.MIN_VALUE;
      for (int i=0; i<seq.length;i++){
        if (seq[i]>max)
          max = seq[i];
      }
      return max;
    }

    public static int index(int[] seq, int el) {
      for (int i=0;i<seq.length;i++){
        if (seq[i]==el)
          return i;
      }
      return -1;
    }

    public static boolean contains(int[] seq, int el) {
        return (index(seq,el) != -1);
    }

    public static boolean isPrime(int el) {
        if (el>1){
          for (int i=2;i<el;i++){
            if (el%i==0)
              return false;
          }
          return true;

        } else {
          throw new IllegalArgumentException();
        }
    }

    public static int countPrimes(int[] seq) {
      int count = 0;
      for (int i=0; i<seq.length;i++){
        if(isPrime(seq[i]))
          count++;
      }
      return count;
    }

    public static int[] primesIn(int[] seq) {
      int[] result = new int[countPrimes(seq)];
      int j = 0;
      for (int i=0; i<seq.length;i++){
        if(isPrime(seq[i])){
          result[j] = seq[i];
          j++;
        }
      }
      return result;
    }

    public static int[] primesUpTo(int n) {
      int[] temp = new int[(n-2)];
      //cascade array from 2, to n-1.
      for (int i=0;i<(n-2);i++){
        temp[i] = (i+2);
      }
      // count primes in casacade array and build result array with only primes of temp
      int[] result = new int[countPrimes(temp)];
      int j = 0;
      for (int i=0;i<temp.length;i++){
        if(isPrime(temp[i])){
          result[j] = temp[i];
          j++;
        }
      }
      return result;
    }
}
```

TEST

```
package weblab;

import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertion
s.*;

class UTest {

    @Test
    void maxEmpty() {
        assertEquals(Integer.MIN_VALUE, Assig
nment2_2.max(new int[]{}), "Empty array must
have a max of MINVALUE");
    }

    @Test
    void maxOneElement() {
        assertEquals(5, Assignment2_2.max(new
 int[]{5}), "Single value in array must be th
e maximum");
    }

    @Test
    void maxManyElements() {
        assertEquals(16, Assignment2_2.max(ne
w int[]{3, 6, 16, -10, -
1}), "Maximum from many values must be correc
t");
    }

    @Test
    void indexEmpty() {
        assertEquals(-
1, Assignment2_2.index(new int[]{}, 5), "Empy
 array should return index -1");
    }

    @Test
    void indexFound() {
        assertEquals(2, Assignment2_2.index(n
ew int[]{3, 6, 5, 1}, 5), "Should return inde
x of value in array");
    }

    @Test
    void indexFirstPlaceFound() {
        assertEquals(0, Assignment2_2.index(n
ew int[]{3, 6, 5, 1}, 3), "Should return inde
x of value in array");
    }

    @Test
    void indexOffByOneCheck() {
        assertEquals(3, Assignment2_2.index(n
ew int[]{3, 6, 5, 1}, 1), "Should return inde
x of value in array");
    }

    @Test
    void indexFirstValue() {
        assertEquals(3, Assignment2_2.index(n
ew int[]{3, 6, 1, 5, 3, 5}, 5), "Should retur
n index of the first value in array");
    }

    @Test
    void indexNotFound() {
        assertEquals(-
1, Assignment2_2.index(new int[]{3, 6, 1, 5,
3, 5}, 7), "Should return -
1 if value is not found in array");
    }

    @Test
    void containsEmpty() {
        assertFalse(Assignment2_2.contains(ne
w int[]{}, 5), "Empty array should not contai
n value");
    }

    @Test
    void containsFirstPlaceFound() {
        assertTrue(Assignment2_2.contains(new
 int[]{3, 6, 5, 1}, 3), "Should contain value
 in array");
    }

    @Test
    void containsFound() {
        assertTrue(Assignment2_2.contains(new
 int[]{3, 6, 5, 1}, 5), "Should contain value
 in array");
    }

    @Test
    void containsDuplicateValue() {
        assertTrue(Assignment2_2.contains(new
 int[]{3, 6, 1, 5, 3, 5}, 5), "Should contain
 value in array with duplicates");
    }

    @Test
    void containsFirstValue() {
        assertTrue(Assignment2_2.contains(new
 int[]{3, 6, 1, 5, 4, 5}, 3), "Should contain
 value in array with duplicates");
    }

    @Test
    void containsOffByOneCheckValue() {
        assertTrue(Assignment2_2.contains(new
 int[]{3, 6, 1, 5, 4, 7}, 7), "Should contain
 value in array with duplicates");
    }

    @Test
    void containsNotFound() {
        assertFalse(Assignment2_2.contains(ne
w int[]{3, 6, 1, 5, 3, 5}, 7), "Should not co
ntain value in array");
```

```
    }

    @SuppressWarnings("ResultOfMethodCallIgno
red")
    @Test
    void isPrimeNegativeNumber() {
        assertThrows(IllegalArgumentException
.class, () -> {
            Assignment2_2.isPrime(-1);
        });
    }

    @SuppressWarnings("ResultOfMethodCallIgno
red")
    @Test
    void isPrimeLowNumber() {
        assertThrows(IllegalArgumentException
.class, () -> {
            Assignment2_2.isPrime(1);
        });
    }

    @Test
    void isPrimeSmallNumberSuccess() {
        assertTrue(Assignment2_2.isPrime(5),
"5 is a prime");
    }

    @Test
    void isPrimeLargeNumberSuccess() {
        assertTrue(Assignment2_2.isPrime(379)
, "379 is a prime");
    }

    @Test
    void isPrimeSmallNumberFailure() {
        assertFalse(Assignment2_2.isPrime(9),
 "9 is not a prime");
    }

    @Test
    void isPrimeLargeNumberFailure() {
        assertFalse(Assignment2_2.isPrime(380
), "380 is not a prime");
    }

    @Test
    void countPrimesEmpty() {
        assertEquals(0, Assignment2_2.countPr
imes(new int[]{}), "Empty array has no primes
");
    }

    @Test
    void countPrimes_manyElements() {
        assertEquals(7, Assignment2_2.countPr
imes(new int[]{167, 173, 179, 181, 191, 31, 3
6, 76, 89}), "Big array must have some primes
");
    }

    @Test
    void primesUpToSmallNumber() {
        assertArrayEquals(new int[]{2, 3, 5,
7}, Assignment2_2.primesUpTo(8), "There are s
ome primes below 8");
    }

    @Test
    void primesUpTo_bigNumber() {
        assertArrayEquals(new int[]{2, 3, 5,
7, 11, 13}, Assignment2_2.primesUpTo(17), "Th
ere are some primes below 17");
    }

    @Test
    void primesInEmpty() {
        assertArrayEquals(new int[]{}, Assign
ment2_2.primesIn(new int[]{}), "Empty array h
as no primes");
    }

    @Test
    void primesInSmall() {
        assertArrayEquals(
                new int[]{167, 173, 179, 191,
 31, 89},
                Assignment2_2.primesIn(new in
t[]{167, 173, 179, 12, 191, 31, 36, 76, 89}),
                "There are some primes in thi
s array"
        );
    }

}
```

ASSIGNMENT 2.3: STRINGLIST

Define the following methods. Remember: if a precondition referring to a parameter fails you should work with an `IllegalArgumentException`.

`public StringList(int n)`
Post: Initialize `amount` with 0, and `elements` with an array of length `n`. If `n` is less than 0, `elements` must have length 0.

`public void add(String el)`
Pre: `amount < elements.length`.
Post: On the first empty place in `elements`, the value `el` is inserted and `amount` is incremented by 1.

`public String get(int i)`
Post: If `0 <= i < amount` return the value on the `i`th place in `elements`, else return `null`.

`public void set(int i, String el)`
Pre: `0 <= i < amount`
Post: The value on the `i`th place in `elements` is updated to `el`.

`public int index(String el)`
Post: Returns the first index which has the same value as `el`, or `-1` if no such value exists.

`public boolean contains(String el)`
Post: If the value `el` is in elements return `true`, else `false`.

`public int getSize()`
Post: Returns `amount`.

`public boolean equals(Object other)`
Post: If `other` is also a `StringList` and has equivalent size, capacity and values on the same indices in elements, return `true` . Else return`false`.

`public String toString()`
Post: Returns a human-friendly `String` representation of this object.

SOLUTION

```
package weblab;

class StringList {

    private String[] elements;
    private int amount;

    public StringList(int n) {
      amount = 0;
      if (n<0)
        n=0;
      elements = new String[n];
    }

    public void add(String el) {
      if (amount<elements.length){
        elements[amount]=el;
        amount++;
      } else {
      throw new IllegalArgumentException();
      }
    }

    public String get(int i) {
      if (0<=i && i<amount){
        return elements[i];
      }
      return null;
    }

    public void set(int i, String el) {
      if (0<=i && i<amount){
        elements[i]=el;
      } else {
      throw new IllegalArgumentException();
      }
    }

    public int index(String el) {
        for (int i=0;i<amount;i++) {
          if (get(i).equals(el)) {
             return i;
          }
        }
        return -1;
    }

    public boolean contains(String el) {
    return (this.index(el) > -1);
    }

    public int getSize() {
        return amount;
    }

    public int getArraySize() {
      return elements.length;
    }

    public int max(int x, int y){
      if (y>x)
        return y;
      return x;
    }

    public boolean equals(Object other) {

       if (other instanceof StringList) {

          StringList that = (StringList) ot
her;
          if (this.getSize() == that.getSiz
e() && this.getArraySize() == that.getArray
Size()) {

            for (int i=0;i<this.getArraySiz
e();i++){
              if (!String.valueOf(this.get(
i)).equals(String.valueOf(that.get(i))))
                return false;
            }

            return true;
          }

          return false;

      } else {
        return false;
      }

     }

     public String toString() {
       String list = "";
       for (int i=0; i<elements.length;i++)
{
         list += elements[i] + " ";
       } list += amount;
      return list;
     }
}
```

TEST

```
package weblab;

import org.junit.jupiter.api.Test;

import java.util.concurrent.atomic.AtomicBoolean;

import static org.junit.jupiter.api.Assertions.*;

@SuppressWarnings({"RedundantStringConstructorCall"})
class StringListTest {

    @Test
    void initializeAmountTest() {
        assertEquals(0, new StringList(1).getSize(), "Amo
unt should initially be zero");
    }

    @Test
    void emptyListTest() {
        assertThrows(IllegalArgumentException.class, () -
> {
            new StringList(0).add("New String");
        });
    }

    @Test
    void illegalDimensionSetsEmptyListTest() {
        assertThrows(IllegalArgumentException.class, () -
> {
            new StringList(-5).add("New String");
        });
    }

    @Test
    void fullListTest() {
        StringList list = new StringList(2);
        list.add("String 1");
        list.add("String 2");
        assertThrows(IllegalArgumentException.class, () -
> {
            list.add("String 3");
        });

    }

    @Test
    void addAndGetTest() {
        StringList list = new StringList(2);
        list.add("String 1");
        assertEquals("String 1", list.get(0), "The first
item should be String 1");
    }

    @Test
    void addAndGetMultipleValuesTest() {
        StringList list = new StringList(3);
        list.add("String 1");
        list.add("String 2");
        assertEquals("String 2", list.get(1), "The second
 item should be String 2");
    }

    @Test
    void negativeGetTest() {
        assertNull(new StringList(2).get(-
5), "Invalid get should return null");
    }

    @Test
    void outsideRangeGetTest() {
        assertNull(new StringList(2).get(5), "Outside ran
ge get should return null");
    }

    @Test
    void addSetAndGetTest() {
        StringList list = new StringList(3);
        list.add("String 1");
        list.set(0, "String 2");
        assertEquals("String 2", list.get(0), "The first
item should be String 2");
    }

    @Test
    void negativeSetTest() {
        assertThrows(IllegalArgumentException.class, () -
> {
            new StringList(5).set(-2, "String -2");
        });
    }

    @Test
    void outsideRangeSetTest() {
        assertThrows(IllegalArgumentException.class, () -
> {
            new StringList(5).set(6, "String 6");
        });
    }

    @Test
    void notLessThanEqualAmountSetTest() {
        assertThrows(IllegalArgumentException.class, () -
> {
            new StringList(3).set(2, "String 2");
        });
    }

    @Test
    void indexTest() {
        StringList list = new StringList(3);
        list.add("String 1");
        assertEquals(0, list.index(new String("String 1")
), "The first item should be String 2");
    }

    @Test
    void indexDuplicateValuesTest() {
        StringList list = new StringList(3);
        list.add("String 1");
        list.add("String 1");
        assertEquals(0, list.index(new String("String 1")
), "The first item should be String 2");
    }

    @Test
    void indexNonExistingTest() {
        assertEquals(-
1, new StringList(2).index(new String("String 1")), "Empt
y list can not have any items");
    }

    @Test
    void containsTest() {
        StringList list = new StringList(3);
        list.add("String 1");
        assertTrue(list.contains(new String("String 1")),
 "The first item should be String 2");
    }

    @Test
    void containsDuplicateValuesTest() {
        StringList list = new StringList(3);
        list.add("String 1");
        list.add("String 1");
        assertTrue(list.contains(new String("String 1")),
 "The first item should be String 2");
    }

    @Test
    void containsNonExistingTest() {
```

```
        assertFalse(new StringList(2).contains(new String
("String 1")), "Empty list can not have any items");
    }

    @SuppressWarnings("EqualsWithItself")
    @Test
    void equalsSameListSuccesfulTest() {
        StringList list = new StringList(0);
        assertEquals(list, list, "An object should be equ
al to itself");
    }

    @Test
    void equalsOtherListAmountFailureTest() {
        StringList one = new StringList(0);
        StringList other = new StringList(1);
        assertNotEquals(one, other, "Different lists with
 different amount should not be equal");
    }

    @Test
    void equalsOtherListSuccesfulTest() {
        StringList one = new StringList(1);
        StringList other = new StringList(1);
        one.add(new String("String 1"));
        other.add(new String("String 1"));
        assertEquals(one, other, "Different lists with sa
me elements should be equal");
    }

    @Test
    void equalsWithDifferentLengthSameAmountTest() {
        StringList one = new StringList(2);
        StringList other = new StringList(3);
        one.add(new String("String 1"));
        one.add(null);
        other.add(new String("String 1"));
        other.add(null);
        assertNotEquals(one, other, "Different lists of d
ifferent length should not be equal");
    }

    @Test
    void equalsOtherListElementsFailure() {
        StringList one = new StringList(1);
        StringList other = new StringList(1);
        one.add(new String("String 1"));
        other.add(new String("String 2"));
        assertNotEquals(one, other, "Different list with
different elements should not be equal");
    }

    @SuppressWarnings({"ConstantConditions", "ObjectEqual
sNull", "SimplifiableJUnitAssertion"})
    @Test
    void equalsNullFailure() {
        assertNotEquals(new StringList(1), null, "Null sh
ould not be equal to an object");
    }

    @Test
    void equalsWithNullValue() {
        StringList one = new StringList(3);
        StringList other = new StringList(3);
        one.add(null);
        one.add("a");
        other.add(null);
        other.add("a");
        assertEquals(one, other, "StringLists should work
 with null values");
    }

    @Test
    void equalsAmount() {
        StringList one = new StringList(3);
        StringList other = new StringList(3);
        one.add("a");
        other.add("a");
        other.add(null);
        assertNotEquals(one, other, "StringLists with a n
ull value added should be different");
    }

    @Test
    void equalsAmountNotSame() {
        StringList one = new StringList(3);
        StringList other = new StringList(3);
        one.add(null);
        assertNotEquals(one, other, "StringLists with a d
ifferent amount of items should not be the same");
    }

    @Test
    void equalsDoesNotUseTostring() {
        final AtomicBoolean called = new AtomicBoolean(fa
lse);

        class StringListMock extends StringList {

            private StringListMock(int n) {
                super(n);
            }

            @Override
            public String toString() {
                called.set(true);
                return super.toString();
            }

        }

        StringList list1 = new StringListMock(1);
        StringList list2 = new StringListMock(1);
        list1.add(new String("String 1"));
        list2.add(new String("String 1"));
        assertEquals(list1, list2, "Same Stringlists shou
ld be equal");
        assertFalse(called.get(), "The equals method shou
ld not be using toString for comparison");
    }

}
```

ASSIGNMENT 3.1: DATE

In this assignment, define the class `Date` (make sure you do not use the `public` keyword) and write JUnit 5 tests in a test-suite in the test tab.

- Create at least one test for every method you implement.
- Additionally, write appropriate Javadoc for the methods you implement.

`Date` has 1 field: `String date`.

Define the following methods:

`public Date(String date)`
Post: Initializes the field `date` with the value of the argument `date`

`public String getDate()`
Post: Returns the date `String`.

`public String toString()`
Post: Returns a `String` representation of this object following the pattern below:
`Date("01-01-2015")` → `<Date: 01-01-2015>`

`public boolean equals(Object other)`
Post: If `other` is also a `Date` and has an equivalent date, return `true`. Else return `false`.

SOLUTION

```
class Date {
    private String date;

    /**Constructor: creates a new Date object that contains a String date.
    * @param date - A date string value in "dd-mm-yyyy".
    */
    public Date(String date){
        this.date = date;
    }

    /**Getter for the date field.
    * @return String with the date value in "dd-mm-yyyy".
    */
    public String getDate(){
        return this.date;
    }

    /**toString method.
    * @return a String representation of this object "<Date: dd-mm-yyyy>"
    */
    public String toString(){
        return("<Date: " + this.date + ">");
    }

    /**Compares this Date object with another object and assess whether they are the same dat
e.
    * @param other - an Object that is compared to this.
    * @return a boolean value which is true when both objects are of the same instance and co
ntain same date"
    */
    public boolean equals(Object other){
        if(other == this) return true;
        if(!(other instanceof Date)) return false;
        Date that = (Date) other;
        return this.getDate().equals(that.getDate());
    }
}
```

TEST

```
package weblab;

import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.BeforeEach;

class UTest {
    /*********
    //Declare the test fields/attributes
    private Date testDate;
    private Date testDateDiff;
    private Date testDateSame;
    private String stringDate;


    @BeforeEach
    void setUp(){
        //Initialise the test fields/attributes
        stringDate ="01-01-2015";
        testDate = new Date(stringDate);
    }

    @Test
    void Date() {
        //Make an assertion that is tested.
        assertNotNull(testDate,"Date(String date)");
    }

    @Test
    void getDate() {
        //Make an assertion that is tested.
        assertEquals(testDate.getDate(),stringDate,"getDate()");
    }

    @Test
    void toStringTest() {
        //Make an assertion that is tested.
        assertEquals("<Date: " + stringDate + ">",testDate.toString(),"toString()");
    }

    @Test
    void equals() {
        //Initialise the test fields/attributes
        testDateDiff = new Date("01-01-1999");
        testDateSame = new Date(stringDate);
        //Make an assertion that is tested.
        assertNotEquals(testDate,testDateDiff,"equals()");
        assertEquals(testDate,testDateSame,"equals()");
    }
}
```

ASSIGNMENT 3.2: DATESET

In this assignment, define the class `DateSet` and write JUnit tests in the test_suite.

- Create at least one test for every method you implement.
- Additionally, write appropriate Javadoc for the methods you implement.
- Remember: if a precondition referring to a parameter fails you should work with an `IllegalArgumentException`.

`DateSet` has 1 (private) field: `List<Date> dates`.

Now that you're going to use more functionality, you are going to need to import packages and classes. In this case you need the `import` statement: `import java.util.List;` in order to use `List`. You will also need to make use of the class `ArrayList`.
`import` statements go near the top of the file, before the class declaration.
If you ever need information about classes or packages that you can import you can look at the Java documentation: https://docs.oracle.com/en/java/javase/11/docs/api/java.base/module-summary.html

Define the following methods:
`public DateSet()`
Post: Initializes the field `dates` with an empty `ArrayList`. *Mind you, the type of the field is different from the actual type that you instantiate the field with! It is important to understand why this is possible!*

`public int getDatesSize()`

Post: Return the size of `dates`.

`public Date getDate(int i)`

Pre: There is a Date on position `i`.
Post: Return the date on position `i` of the list `dates`.

`public void add(Date date)`
Pre: `dates` does not contain `date`.
Post: Adds `date` to the list of `dates`.

`public boolean contains(Date date)`
Post: If the `date` is in `dates` return `true`, else `false`.

`public DateSet intersection(DateSet other)`
Post: Returns a new `DateSet` with only those dates that are in this object as well as in `other`.

`public String toString()`
Post: Returns a `String` representation of this object following the pattern below:
`DateSet([ Date("01-01-2015"), Date("01-01-2016")])` → `"<DateSet: [<Date: 01-01-2015>, <Date: 01-01-2016>]>"`

`public boolean equals(Object other)`
Post: If `other` is also a `DateSet` and the lists of `this` and `other` are equal, return `true`. Else return `false`.

SOLUTION

```
package weblab;
import java.util.List;
import java.util.ArrayList;

class DateSet{

    private List<Date> dates;

    /**Constructor - creates a new DateSet object by initalising a child of date's static typ
e List
    * dates is allocated to an ArrayList because List is abstract and can't be initalised.
    */
    public DateSet(){
        dates = new ArrayList<Date>();
    }

    /**Getter for dates ArrayList size
    * @return an int with the size of the ArrayList.
    */
    public int getDatesSize(){
        return dates.size();
    }

    /**Getter: for the Date object
    * @param - int: points to the ArrayList index
    * @return the Date object at index int
    */
    public Date getDate(int i){
        if (i >= getDatesSize()) throw new IllegalArgumentException();
        return dates.get(i);
    }

    /**Adder: adds a Date object to the ArrayList
    * @param - date: Date object that will be added at the end of the ArrayList
    */
    public void add(Date date){
        if (dates.contains(date)) throw new IllegalArgumentException();
        dates.add(date);
    }

    /**Contains: checks if a Date exists in the ArrayList
    * @param - date: Date to be searched in the ArrayList
    * @return - boolean: true if found, false if not found
    */
    public boolean contains(Date date){
        return dates.contains(date);
    }
```

```
    /**Intersection: compares this to other and makes a set of only intersecting elements
    * @param - other: DateSet object that contains an ArrayList that will be compared to this
    * @return a DateSet object with the intersection ArrayList.
    */
    public DateSet intersection(DateSet other){
        DateSet temp = new DateSet();
        for (int i=0; i<this.getDatesSize(); i++){
            if (other.contains(this.getDate(i))) temp.add(this.getDate(i));
        }
        return temp;
    }

    /**toString: Returns a String representation of this object following the pattern below
    /* @return String of comma seperated Dates with format <DateSet: [<Date: dd-mm-
yyyy>, <Date: dd-mm-yyyy>]>
    */
    public String toString(){
        String result ="<DateSet: [";
        for (int i=0; i<this.getDatesSize()-1;i++){
            result += this.getDate(i) + ", ";
        }
        result += this.getDate(this.getDatesSize()-1) + "]>";
        return result;
    }

    /**Equals: checks if the other object is also an instance of this and checks if the list
contents are the same
    * @param - other: Object other to be compared to this
    * @return a boolean true if equal false if not equal
    */
    public boolean equals(Object other){

        if(this == other) return true;
        if(!(other instanceof DateSet)) return false;

        DateSet that = (DateSet) other;
        if(this.getDatesSize() != that.getDatesSize()) return false;

        DateSet temp = new DateSet();
        temp = this.intersection(that);
        if (temp.getDatesSize() == this.getDatesSize()) return true;
        return false;
    }

}
```

TEST

```java
package weblab;

import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.BeforeEach;

class UTest {

  private DateSet myDateSet;
  private DateSet myOtherDateSet;
  private Date myDate;

  @BeforeEach
  void setUp(){
      myDate = new Date("1-1-1999");
      myDateSet = new DateSet();
  }
  @Test
  void DateSet(){
      assertNotNull(myDateSet,"DateSet()");
  }
  @Test
  void getDatesSize(){
      assertEquals(0,myDateSet.getDatesSize(),"getDatesSize()");
  }
  @Test
  void add(){
      myDateSet.add(myDate);
      assertEquals(myDate,myDateSet.getDate(0),"add()");
  }
  @Test
  void getDateNull(){
      assertThrows(IllegalArgumentException.class,
      () -> {
          myDateSet.getDate(0);
      }
      ,"getDate(int i)");
  }
  @Test
  void getDateOutsideIndex(){
      assertThrows(IllegalArgumentException.class,
      () -> {
          myDateSet.getDate(100);
      }
      ,"getDate(int i)");
  }
```

```
  @Test
  void contains(){
      myDateSet.add(myDate);
      assertTrue(myDateSet.contains(myDate),"contains()");
  }

  @Test
  void intersection(){
      myDateSet.add(new Date("1-1-2001"));
      myDateSet.add(new Date("1-1-2002"));
      myDateSet.add(new Date("1-1-2003"));
      myDateSet.add(new Date("1-1-2004"));
      myOtherDateSet = new DateSet();
      myOtherDateSet.add(new Date("1-1-2001"));
      myOtherDateSet.add(new Date("1-1-2004"));
      assertEquals(2,myDateSet.intersection(myOtherDateSet).getDatesSize(),"intersection()");
  }

  @Test
  void toStringTest(){
      myDateSet.add(new Date("01-01-2015"));
      myDateSet.add(new Date("01-01-2016"));
      assertEquals("<DateSet: [<Date: 01-01-2015>, <Date: 01-01-
2016>]>",myDateSet.toString(),"toString()");
  }

  @Test
  void equals(){
      myOtherDateSet = new DateSet();
      myDateSet.add(new Date("01-01-2015"));
      myOtherDateSet.add(new Date("01-01-2015"));
      assertTrue(myDateSet.equals(myOtherDateSet));
  }

  @Test
  void equalsNull(){
      myDateSet.add(new Date("01-01-2015"));
      assertFalse(myDateSet.equals(myOtherDateSet));
  }
}
```

ASSIGNMENT 3.3: PERSON

In this assignment, define the class `Person` and write JUnit tests in the test_suite.

- Create at least one test for every method you implement.
- Additionally, write appropriate Javadoc for the methods you implement.
- Remember: if a precondition referring to a parameter fails you should work with an `IllegalArgumentException`.

`Person` has 2 fields: `String name` and `DateSet dates`.

Define the following addtional methods:

`public Person(String name)`
Post: Initializes the field `name` with the value of the argument `name`. Initializes the field `dates` with an empty `DateSet`.

`public String getName()`
Post: Returns the name of the `Person`.

`public DateSet getDates()`
Post: Returns the `DateSet` of the `Person`.

`public void addDate(Date date)`
Pre: `dates` does not contain `date`.
Post: Adds `date` to the list of `dates`.
Hint: Try not to reimplement the pre-condition, but use existing methods.

`public boolean equals(Object other)`
Post: If `other` is also a `Person` and has equivalent `name` and `dates`, return `true`. Else return `false`.
Hint: Re-use code written previously for comparing the dates

`public String toString()`
Post: Returns a `String` representation of this object following the pattern below:
Example output: `"<Person: <Name: Thomas>, <DateSet: [<Date: 01-01-2015>, <Date: 01-01-2016>]>>"`

SOLUTION

```
package weblab;

class Person{

    private String name;
    private DateSet dates;

    /**Constructor for Person object
    * @param - name: String with the name of the person
    */
    public Person(String name){
        this.name = name;
        dates = new DateSet();
    }

    /**Getter for person name
    * @return a String with the person name
    */
    public String getName(){
        return this.name;
    }

    /**Getter for person dates
    * @return a DateSet object with person dates
    */
    public DateSet getDates(){
        return this.dates;
    }

    /**Adder for person dates
    * @param - date: Date object to be added
    */
    public void addDate(Date date){
        this.dates.add(date);
    }

    /**Equals for Person objects. Check if both objects are Persons and if they have same nam
e and dates.
    * @param - other: Object of type Person to be compared to this
    * @return a boolean, true if equal false if not
    */
    public boolean equals(Object other){
        if(this == other) return true;
        if(!(other instanceof Person)) return false;
        Person that = (Person) other;
        return(this.getName() == that.getName() && this.dates.equals(that.dates));
    }

    /**toString for Persons, transforms Persons contents to following String pattern:
    * "<Person: <Name: Thomas>, <DateSet: [<Date: 01-01-2015>, <Date: 01-01-2016>]>>"
    * @retunr a String with the above pattern
    */
    public String toString(){
        return "<Person: <Name: " + this.getName() + ">, " + this.dates.toString() + ">";
    }
}
```

TEST

```
package weblab;

import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.BeforeEach;

class UTest {

  private Person person1;

  @BeforeEach
  void setUp(){
      person1 = new Person("Sergio");
  }

  @Test
  void Person(){
      assertNotNull(person1,"Person()");
  }

  @Test
  void getName(){
      assertEquals("Sergio",person1.getName(),"getName()");
  }

  @Test
  void getDates(){
      assertNotNull(person1.getDates(),"getDates()");
  }

   @Test
  void addDate(){
      person1.addDate(new Date("1-1-1999"));
      assertEquals("<DateSet: [<Date: 1-1-
1999>]>",person1.getDates().toString(),"addDate(Date date)");
  }

  @Test
  void equals(){
      person1.addDate(new Date("1-1-1999"));
      Person person2 = new Person("Sergio");
      person2.addDate(new Date("1-1-1999"));
      assertEquals(person1,person2,"equals()");
  }

  @Test
  void toStringTest(){
      person1.addDate(new Date("1-1-1999"));
      assertEquals("<Person: <Name: Sergio>, <DateSet: [<Date: 1-1-
1999>]>>",person1.toString());
  }

}
```

ASSIGNMENT 3.4: DATEPICKER

In this assignment, define the class DatePicker and write JUnit tests in the test_suite.

- Create at least one test for every method you implement.
- Additionally, write appropriate Javadoc for the methods you implement.
- Remember: if a precondition referring to a parameter fails you should work with an IllegalArgumentException.

DatePicker has one field: List<Person> persons.

Define the following methods:

public DatePicker()
Post: Initializes the field persons with an empty ArrayList.

public int getPersonsSize()
Post: Returns the size of persons.

public Person getPerson(int i)
Pre: There should be a Person on position i of persons.
Post: Returns the Person on the ith position of persons.

public void addPerson(Person person)
Pre: persons does not contain person.
Post: Adds person to the list of persons.

public DateSet commonDates()
Pre: persons contains at least two Person objects.
Post: Returns a DateSet which contains all dates that all persons have in common in their DateSet (in other words the intersection of the DateSet of all people).

public String toString()
Post: Returns a String representation of this object following the pattern below:

```
"<DatePicker: [<Person: <Name: Henk>, <DateSet: [<Date: 1-1-2015>, <Date: 12-12-2016>]>>, <Person: <Name: Jan>, <DateSet: [<Date: 1-3-2015>, <Date: 12-12-2016>]>>]>"
```

SOLUTION

```
package weblab;
import java.util.List;
import java.util.ArrayList;

class DatePicker{

    private List<Person> persons;

    /** Constructor for DatePicker an empty Person ArrayList is initalised
    */
    public DatePicker(){
        persons = new ArrayList<Person>();
    }

    /**Getter for number of persons in the ArrayList
    * @return an inter with the number of persons
    */
    public int getPersonsSize(){
        return persons.size();
    }

    /**Getter for Person object on the parameter position
    * @paramater - i: index of the ArrayList
    * @return the Person on the ith position
    */
    public Person getPerson(int i){
        if(i>=this.getPersonsSize() || i<0 ) throw new IllegalArgumentException();
        return this.persons.get(i);
    }

    /**Adder for DatePicker, it adds a Person object to the ArrayList
    * @param - person: Person object to add to the end of the ArrayList
    */
    public void addPerson(Person person){
        if(this.persons.contains(person)) throw new IllegalArgumentException();
        this.persons.add(person);
    }

    /**Intersection in DatePicker between all of its persons dates
    * @return a DateSet with all common dates
    */
    public DateSet commonDates(){
        if(this.getPersonsSize()<2) throw new IllegalArgumentException();
        DateSet result = new DateSet();
        result = this.getPerson(0).getDates().intersection(this.getPerson(1).getDates());

        for (int i=2; i<this.getPersonsSize(); i++){
            result = this.getPerson(i).getDates().intersection(result);
        }
        return result;
    }

    /**toString for the Date picker represents all dates
    */
    public String toString(){

        String result = "<DatePicker: [";
        int n = this.getPersonsSize()-1;
        if (n == -1) return (result + "]>");

        for (int i=0; i<n; i++) {
            result += this.getPerson(i).toString() + ", ";
        }
        result += this.getPerson(n).toString() + "]>";
        return result;
    }
}
```

TEST

```
package weblab;

import static org.junit.jupiter.api.Assertion
s.*;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.BeforeEach;


class UTest {

  private DatePicker myDatePicker;
  private Person person1;
  private Person person2;
  private Person person3;
  private Person person4;
  private Person personWithoutDates;

  @BeforeEach
  void setup(){
      myDatePicker = new DatePicker();
      person1 = new Person("Juan");
      person2 = new Person("Lou");
      person3 = new Person("Lee");
      personWithoutDates = new Person("Sergio
");
  }

  @Test
  void DatePicker(){
      assertNotNull(myDatePicker,"DatePicker(
)");
  }

  @Test
  void getPersonsSize(){
      assertEquals(0,myDatePicker.getPersonsS
ize(),"getPersonsSize()");
  }

  @Test
  void getPerson(){
      assertThrows(IllegalArgumentException.c
lass,
      () -> {
          myDatePicker.getPerson(0);
      },"getPerson()");
  }

  @Test
  void addPerson(){
      myDatePicker.addPerson(person1);
      assertEquals(person1,myDatePicker.getPe
rson(0),"addPerson()");
  }

   @Test
  void addPersonSame(){
      myDatePicker.addPerson(person1);
      assertThrows(IllegalArgumentException.c
lass,
      () -> {
          myDatePicker.addPerson(person1);
      },"addPerson()");
  }

  @Test
  void commonDates(){
      myDatePicker.addPerson(person1);
      myDatePicker.addPerson(person2);

      person1.addDate(new Date("8-10-2020"));
      person1.addDate(new Date("9-10-2020"));

      person2.addDate(new Date("9-10-2020"));
      person2.addDate(new Date("10-10-
2020"));

      assertEquals(1,myDatePicker.commonDates
().getDatesSize(),"commonDates()");
  }

  @Test
  void commonDatesTwo(){
      myDatePicker.addPerson(person1);
      myDatePicker.addPerson(person2);

      person1.addDate(new Date("8-10-2020"));
      person1.addDate(new Date("9-10-2020"));
      person1.addDate(new Date("10-10-
2020"));

      person2.addDate(new Date("9-10-2020"));
      person2.addDate(new Date("10-10-
2020"));
      person2.addDate(new Date("11-10-
2020"));

      assertEquals(2,myDatePicker.commonDates
().getDatesSize(),"commonDates()");
  }

    @Test
  void commonDatesThree(){
      myDatePicker.addPerson(person1);
      myDatePicker.addPerson(person2);
      myDatePicker.addPerson(person3);

      person1.addDate(new Date("8-10-2020"));
      person1.addDate(new Date("9-10-2020"));
      person1.addDate(new Date("10-10-
2020"));

      person2.addDate(new Date("9-10-2020"));
      person2.addDate(new Date("10-10-
2020"));
      person2.addDate(new Date("11-10-
2020"));

      person3.addDate(new Date("7-10-2020"));
      person3.addDate(new Date("8-10-2020"));
```

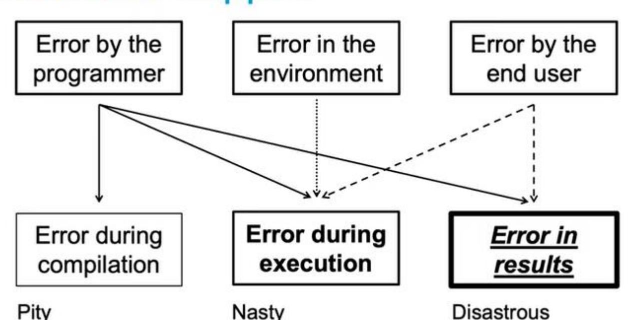
```
      person3.addDate(new Date("9-10-2020"));

      assertEquals(1,myDatePicker.commonDates
().getDatesSize(),"commonDates()");
  }

   @Test
  void commonDatesZero(){
      myDatePicker.addPerson(person1);
      myDatePicker.addPerson(person2);
      myDatePicker.addPerson(person3);

      person1.addDate(new Date("8-10-2020"));
      person1.addDate(new Date("9-10-2020"));
      person1.addDate(new Date("10-10-
2020"));

      person2.addDate(new Date("9-10-2020"));
      person2.addDate(new Date("10-10-
2020"));
      person2.addDate(new Date("11-10-
2020"));

      person3.addDate(new Date("7-10-2020"));
      person3.addDate(new Date("8-10-2020"));

      assertEquals(0,myDatePicker.commonDates
().getDatesSize(),"commonDates()");
  }

    @Test
  void commonDatesAlone(){
      myDatePicker.addPerson(person1);
      assertThrows(IllegalArgumentException.c
lass,
      () -> {
          myDatePicker.commonDates();
      },"commonDates()");
  }

  @Test
  void commonDatesNull(){
      myDatePicker.addPerson(person1);
      myDatePicker.addPerson(person2);
      myDatePicker.addPerson(person3);

      person2.addDate(new Date("7-10-2020"));
      person3.addDate(new Date("7-10-2020"));


      assertEquals(0,myDatePicker.commonDates
().getDatesSize(),"commonDates()");
  }

  @Test
  void toStringTestTwo(){
      myDatePicker.addPerson(person1);
      myDatePicker.addPerson(person2);
```

```
      person1.addDate(new Date("7-10-2020"));
      person1.addDate(new Date("8-10-2020"));

      person2.addDate(new Date("8-10-2020"));

      assertEquals("<DatePicker: [<Person: <N
ame: Juan>, <DateSet: [<Date: 7-10-
2020>, <Date: 8-10-
2020>]>>, <Person: <Name: Lou>, <DateSet: [<D
ate: 8-10-2020>]>>]>"
      ,myDatePicker.toString(),"toString()");

  }

  @Test
  void toStringTestOne(){
      myDatePicker.addPerson(person1);
      //myDatePicker.addPerson(person2);

      person1.addDate(new Date("7-10-2020"));
      person1.addDate(new Date("8-10-2020"));

      person2.addDate(new Date("8-10-2020"));

      assertEquals("<DatePicker: [<Person: <N
ame: Juan>, <DateSet: [<Date: 7-10-
2020>, <Date: 8-10-2020>]>>]>"
      ,myDatePicker.toString(),"toString()");

  }

  @Test
  void toStringTestOneHalf(){
      myDatePicker.addPerson(person1);
      myDatePicker.addPerson(person2);

      person1.addDate(new Date("7-10-2020"));
      person1.addDate(new Date("8-10-2020"));

      assertEquals("<DatePicker: [<Person: <N
ame: Juan>, <DateSet: [<Date: 7-10-
2020>, <Date: 8-10-
2020>]>>, <Person: <Name: Lou>, <DateSet:]>>]
>"
      ,myDatePicker.toString(),"toString()");

  }

  @Test
  void toStringZero(){
      assertEquals("<DatePicker: []>",myDateP
icker.toString(),"toString()");
  }

}
```

WEEK 6 – DEBUGGING & IO

EXCEPTIONS (FOCUS ON DURING EXECUTION)

Mistakes happen



Input of a number can lead to…

- NumberFormatException
- Integer overflow (not an exception!)
- Floating-point overflow (not an exception!)
- Floating-point division by zero (not an exception!)
  - It is a NaN → what is a NaN?
- Integer division by zero (ArithmeticExeption)

And those are just some errors related to numbers. Many more things can go wrong…

Errors during memory allocation

- OutOfMemoryError

```
double[][] a = new double[2000][2000]; //32 MB
double[][] b = new double[2000][2000]; //32 MB
```

- StackOverflowError

```
private void addElement(String s){
    if (s.length() > 0)
      addElement(s);
}
```

Recursion that doesn't stop…

This is a 2 dimensional array[][]

Stack & Heap

Stack	Heap
Runs out with many stacked method calls	*Runs out with a lot of data e.g., huge 2D Arrays*
• Primitive types • References to things on the heap	• Objects • Arrays

Exceptions with Arrays and Strings

- ArrayIndexOutOfBoundsException

```
private int[] a = new int[3];
a[3] = 0;
```

- StringIndexOutOfBoundsException

```
private String s = "abc";
char c = s.charAt(3);
```

- NegativeArraySizeException

```
private int size = -1;
int[] a = new int[size];
```

- ArrayStoreException

```
private Object[] x = new String[3];
x[1] = new Integer(0);
```

Exceptions with Objects

- NullPointerException

```
Label s;
s.setText("a");
→ Forgot to: Label s = new Label()!!!
```

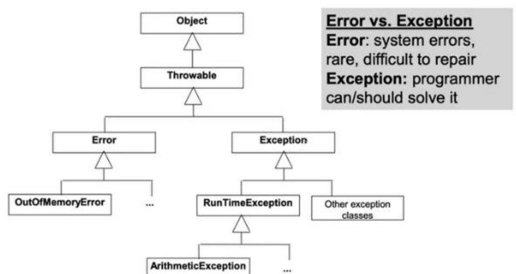
- ClassCastException

```
private Object i = new Integer(0);
String s = (String) i;
```

Array store exception when adding an element of different type.

NullpointerException: when we forget to create (allocate memory in the heap) an instance for a reference type.

Exceptions and Errors



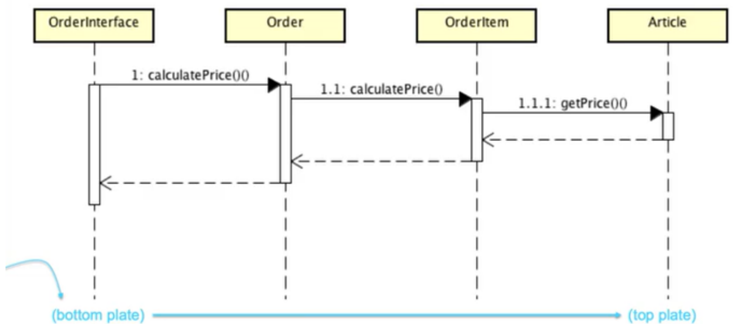
Errors are often beyond the control of the programmer.

- Should generally not be "caught"
- They represent irrecoverable sitautions (such us out of memory)

Exceptions must be solved by the programmer

- **Unchecked Runtime Exceptions**
  - Should be caught. Can be declared
  - Represent unexpected but recoverable situations (null, out of bounds)
- **Checked Runtime Exceptions (Java wont compile)**
  - Java will **force** you to catch or pass along. **Must** be declared
  - Represent recoverable situations (wrong input, file not present,etc.).

CallStack (1)



Assume: something goes wrong in class `Article`, method `getPrice()`
→ We can return "error codes", e.g. -1
→ We can work with exceptions

16

public double article.getPrice()

With return codes

```java
public double getPrice() {
  // do something
  if(somethingGoesWrong){
     return -1;
  }
  else {
     return price;
  }
}
```

With exceptions

```java
public double getPrice() {
  try{
    // do something
    if(somethingGoesWrong) {
      throw new PriceException(
        "Price is negative");
    }
    else {
      return price;
    }
  } …
```

C sends -1 as error. Java prefers exceptions

Exception thrown, now what?

An exception is thrown. Your options:

1. Catch and "deal with it"
2. Pass along
   - Next method in the stack gets the same choice

- If nobody deals with the exception Java's "**default exception handler**" will do it.
  - Usually means your program is terminated

Catch and deal with (1)

```java
errorLabel.setText("");

try{
   String inString = inField.getText();
   int i = Integer.parseInt(inString);
   outArea.appendText("i = " + i + "\n");
}

catch (NumberFormatException exception){
   errorLabel.setText("Not a number!");
}
```

try something that may raise an exception

catch the exception when it's thrown

Control jumps from place where exception is raised to first line of catch block

23

terminated = crashes

Catch and deal with (2)

```java
errorLabel.setText("");

try{
   String inString = inField.getText();
   int i = Integer.parseInt(inString);
   outArea.appendText("i = " + i + "\n");
}

catch (NumberFormatException exception){
   errorLabel.setText("Not a number!");
}
```

- If `Integer.parseInt(inString)` raises an exception, the next line is not executed
- Instead, if catch matches Exception type, control goes to this block

24

Syntax

```
try
{
   // your code
}
[catch-clause]*
[finally-clause]
```

There can be 0, 1 or more...
Why?
1) More than one exception (type) can be thrown!
2) The exception is dealt with elsewhere!

Sometimes you want to clean up something before the program stops (no obligation).

→ Can you think of a good reason?

25

Use finally clause to unlock an open file so that other prorams can use it even after your program crashes.

try {
// code
}
catch (Exception e) {
// what should happen to either report or fix the error
}
finally {
// cleanup: close network, close file, release lock...
}

Warning to developer that things might throw an exception.
You cant fix it there.

Throws clause

- `public abstract int read() throws IOException;`
- You don't catch (= solve) the exception in read(), but you do make clear that the user of read() should catch it (or pass it further along)
- Redefining:
  - Works:
    - `public int read() throws IOException;`
    - `public int read();`
  - Doesn't work:
    - `public int read() throws Exception;`
    - `public int read() throws SQLException;`

Pass along

```
public void setNumber() throws NumberFormatException
{
  …
  String inString = inField.getText();
  int i = Integer.parseInt(inString);
  outArea.appendText("i = " + i + "\n");
}
```

You don't plan on fixing the `NumberFormatException` right here, but you make clear to the method that calls `setNumber()` that it should fix the exception (or pass along again).

If you know you might get an exception and you don't want to fix it. You must at least pass it along with " throws ExceptionName" after the method.

Create your own Exception class

```
public class DivByZeroException extends Exception{

    public DivByZeroException(){
        super();
    }

    public DivByZeroException(String s){
        super(s);
    }
}
```

...and throw it

```
public Fraction(int n, int d) throws
   DivByZeroException
{
   if (d == 0)
      throw new DivByZeroException
           ("Divisor cannot be zero");
   else
      ...
}
```

Q: Why create your own Exception class, and not just use a default Exception?

A: You want to be specific so you can handle exceptions effectively, and find the cause quicker.

Checking errors with a status method

`public static boolean isInfinite(double v)`
Returns true if the specified number is infinitely large in magnitude.

`public static boolean isNaN(double v)`
Returns true if the specified number is the special Not-a-Number (NaN) value.

- You can check, but you don't have to…

Some useful methods?

```
try
{
    int[] exampleOne = new int[3];
    int four = exampleOne[4];
}
catch(IndexOutOfBoundsException e)
{
    e.getMessage();
    e.printStackTrace();
}
```

Other interesting methods: `e.getCause();` `e.toString();`

Summary:
Exceptions allow for the clean and easy dealing of (negative scenario)
You can also work with error codes, but this make your code more complicated (possible complex if-structures)
Errors are due to limitations beyond programmers control and exceptions are bugs that the programmer should fix
Always catch specific exceptions rather than generic exceptions
Try, catch, finally
You can create your own exception class

DEBUGGING

Bugs: unexpected wrong results (origin a real bug)

Solution 1: print-statements

```java
public static int sumOfRange(int from, int to) {
      int sum = 0;
      for (int i = from; i < to; i++) {
            sum = sum + i;
            System.out.println("i is now " + i);
            System.out.println("sum is now " + sum);
      }
      return sum;
}
```

Looking at the console output tells you how the code was executed.

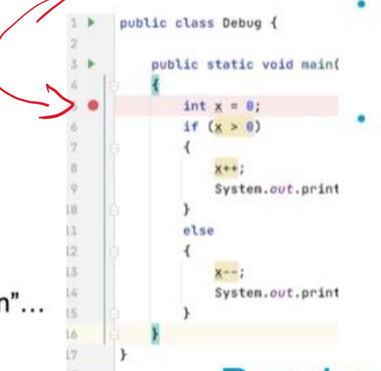
Output

```
i is now 1
sum is now 1
i is now 2
sum is now 3
i is now 3
sum is now 6
i is now 4
sum is now 10
```

Solution 2: The Debugger

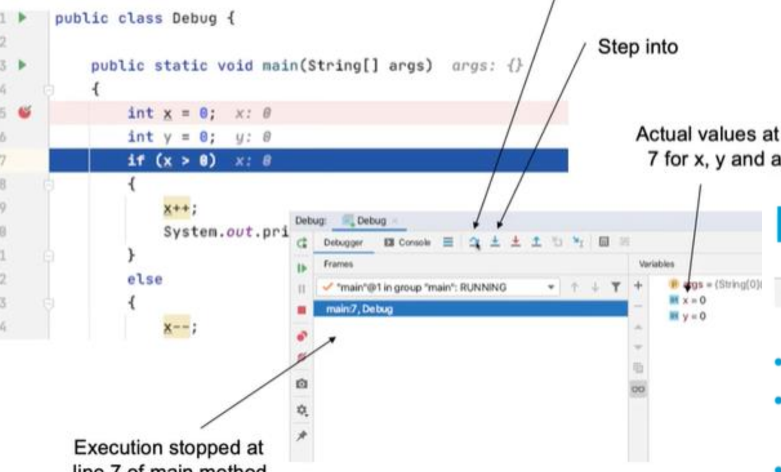
- "playing your program in slow motion"
- What happens if you click this icon?
  - Apparently the same as if you would click "run"…

Setting breakpoints



- Double click in the margin
- If you now run "debug", the VM will pause the program just before executing the statement marked with the breakpoint.

Inspecting values



Breakpoint on method

```java
public static int getSum(int a, int b)
{
    return a + b;
}
```

IntelliJ's debug view

- Step over
- Step into (go into method)
- Force step into (also go into Java libraries)
- Step out (go to caller method)
- (not active)
- Run to cursor

- Resume execution
- (not active)
- Stop debugging

"Watchpoint": breakpoint on attribute

```java
public class Debug {

    private static int z = 0;
```

(**breakpoint on an attribute). each time attribute is accessed or changed**, the debugger pauses the program. Right click to configure either feature.

The break point on a class will only pause the program when something from that class is called

Method breakpoint: debuger will pause each time method is called.

Java Input / Output: Complex

- There is the "classic" java.io library. This contains:
  - More than 50 classes
  - 17 interfaces
  - 18 exceptions-classes
- There is the "new" java.nio library (as of Java 1.7)

→ Why so many options?

Many, many options → why? (1)

- Multitude of data origins/targets:
  - Hard drive, memory card, printer, keyboard, command prompt , …
- I/O can be
  - blocking: the program/thread waits until I/O is available to read or write
  - Non-blocking: the program/thread gets the data that is available now, then continues with other things and returns later to collect/write more data

Many, many options → why? (2)

- **Java.io is stream-oriented:** read one or more bytes at a time, data is not cached, you cannot move back/forth in stream. You can use a buffer to improve performance.
- **Java.nio is block-oriented**: data is read into a buffer and later processed. You can move back/forth in the data.

java.io	java.nio
Stream-oriented	Block-oriented
Blocking I/O	Non-blocking I/O

EASIEST WAY TO READ A TEXT FILE

A simple solution: use a Scanner

```
Scanner scanner = new Scanner(new File("test.txt"));
```

Scanner Object — File Object — File path (requires as argument)

```
String line = scanner.nextLine();
// Do something with scanned line
```

Scan the next line in the file

What if there is no next line?

```
Scanner scanner = new Scanner(new File("test.txt"));

while (scanner.hasNextLine()){
   String line = scanner.nextLine();
   // Do something with scanned line
}
```

Potential exception check (`hasNextLine()`)

Scan the next line (while there is one)

FileNotFoundException is checked!

```
try {
    Scanner scanner = new Scanner(new File("test.txt"));
    while (scanner.hasNextLine()){
        String line = scanner.nextLine();
        // Do something with scanned line
    }
}
catch(FileNotFoundException e){ … }
```

We are forced to deal with potential exceptions

(This works as of Java 1.5)

15

Processing scanned input

```
…
String line = scanner.nextLine();
```

Can be processed further by using existing methods for String such as split(); or by using another Scanner…

```
Scanner linescanner = new Scanner(line);
if(linescanner.hasNext()) // is next token a String?
   String newstring = linescanner.next();

if(linescanner.hasNextInt()) // is next token an int?
   int newint = linescanner.nextInt();
```

Scanning other things

```
Scanner sc1 = new Scanner(new File("test.txt"));
```

Scan from a text file

```
Scanner sc2 = new Scanner("Your string");
```

Scan from a String

```
Scanner sc3 = new Scanner(System.in);
```

Scan user input (from console)

TEXT FILE READER

```
try {
    Scanner scanner = new Scanner(new File("src\\name.txt")); //note the double escape \\
    while (scanner.hasNextLine()) { //so that just one \ is taken.
        String text = scanner.nextLine(); //Could also have used "src/name.txt"k
        System.out.println(text);
    }
}
    catch(FileNotFoundException e){
    System.out.printf("File not found");
}
```

JAVA.IO CLASSES

The classes of Java.io

- Readers
- Writers

Designed for character streams (e.g. used for text files)

- InputStreams
- OutputStreams

Designed for byte streams (used for images or serializable objects)

- Some other functionality

There are several different implementations of readers / writers / streams for different purposes

Child classes of Reader

- BufferedReader
  - LineNumberReader
- CharArrayReader
- FilterReader
  - PushbackReader
- InputStreamReader
  - FileReader
- PipedReader
- StringReader
- URLReader

Many readers
Many optimizations

Let's try reading from a file…

Reading characters from a file

The is a class `FileReader` that we can use.

Example:

```
String filename = "input.txt";
Reader reader = new FileReader(filename);
```

So why not use `FileReader` instead of `Scanner`..?

The read methods of Reader

```
public abstract int read(char cbuf[],
    int off, int len) throws IOException
```

Reads characters into (part of) an array. The method blocks until there is input, an error occurs, or the end of the Stream is reached.

```
public int read() throws IOException
```

Reads 1 character.

You would need to guess how many characters there are:

Cumberstome because you need to provide an array yourself and the reader will read characters and store them in the array one by one.

Second one reads just 1 character

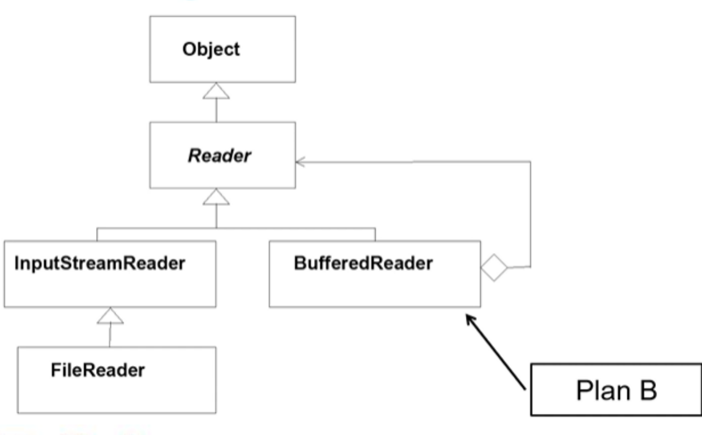
It is a legacy method and very low level.

Using read()

```
import java.io.*;
…
try{
    Reader reader = new FileReader("test.txt");
    int data = reader.read();
    while(data != -1){
        char dataChar = (char) data;
        System.out.println(dataChar);
        data = reader.read();
    }
}
```

Ancient error code

Reading characters



Buffered Reader reads multiple characters and stores also multiple characters into a Buffer (such as an array) at a time.

This is good because we want to use the memory disk reading as less as possible. Therefore reading char by char is not efficient (and neither was guessing the array size in the basic reader).

Buffering

In case of a classical HDD (but even an SSD is slow compared to modern CPU)

- Seek time
- Rotational delay
- Together: 5 ms (or so)
- Clock: 3 Ghz
- 5 ms = 5000 μs = 15.000.000 clock cycles (3 000 000 000 x 0.005 s) ← Those are lost!

- `class BufferedReader`

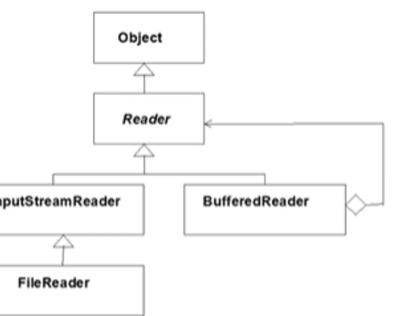
- You can read bigger chunks, which is typically faster!

I/O Latency comparisons

```
Latency Comparison Numbers
--------------------------
L1 cache reference/hit                         1.5 ns          4 cycles
Floating-point add/mult/FMA operation          1.5 ns          4 cycles
L2 cache reference/hit                         5   ns          12 ~ 17 cycles
Branch mispredict                              6   ns          15 ~ 20 cycles
L3 cache hit (unshared cache line)            16   ns          42 cycles
Mutex lock/unlock                             25   ns
L3 cache hit (modified in another core)       29   ns          75 cycles
64MB main memory reference (local CPU)        46   ns
256MB main memory reference (local CPU)       75   ns
Send 4KB over 100 Gbps HPC fabric          1,040   ns
Send 4KB over 10 Gbps ethernet            10,000   ns
Read 4KB randomly from NVMe SSD          120,000   ns
Read 1MB sequentially from NVMe SSD      208,000   ns          ~4.8GB/sec
Read 1MB sequentially from SATA SSD    1,818,000   ns          ~550MB/sec
Read 1MB sequentially from disk        5,000,000   ns          ~200MB/sec
Random Disk Access (seek+rotation)    10,000,000   ns
Send packet CA->Netherlands->CA      150,000,000   ns          150 ms
```

BufferedReader (1)

- BufferedReader *is* a Reader
- BufferedReader *has* a Reader
- Decorator "design pattern"
  - What is a design pattern?
  - What is a decorator? A "regular" Reader, but with benefits



Decorator design pattern: The buffered reader is a wrapper around a normal reader (Like Integer is an Object that wraps the primitive type int)"We can take our File Reader, and put it inside a of a Buffered Reader which creates a Buffer around the File Reader we have".

With a A String Reader or a URL reader we can put the Buffered Reader on top of them and that will provides us with a Buffer on top of the low lever reader we had before.

Reading character per character is "expensive", so reading a number of characters at the same time is more efficient!

BufferedReader (2)

```java
public class BufferedReader extends Reader{

    private char cb[];     // ← Buffer
    private Reader in;     // ← Contains a Reader
    private int nChars;
    private int nextChar;
```

And then we can use the BufferedReader to buffer the information in our FileReader in the buffer.

Basically with a BufferedReader we can "wrap" any Basic Reader when we construct the BufferedReader, which will allows us to use the methods of the BufferedReader.

BufferedReader (3)

```java
BufferedReader br = new BufferedReader(
    new FileReader("file1.txt"));
try {
    String line = br.readLine();
    while(line != null) {
        System.out.println(line);
        line = br.readLine();
    }
}
catch(IOException exception) { … }
finally {
    br.close()
}
```

We can read per line now!

Why is this line a good idea?

Difference from Scanner. Buffered Reader doesnt throw an exception (like the Scanner) when there are no more lines

Any (Basic) Reader locks the file it is reading at the moment it enters the file. If we don't close the file it will remain locked and other programs (or us from different location) wont be able to use it.

Sometimes ti can be tricky to open and close a file in the correct location. Try with resources solves the problem:

We can define the Buffered Reader, and Java (1.7+) will automatically close the file either once we reach an exception or ar done with reading the file. (automatic finally is done). **This is the elegant way to use a Buffered Reader**

BufferedReader (4)

```java
try (BufferedReader br = new BufferedReader(
    new FileReader("file1.txt"))){

    String line = br.readLine();
    while(line != null) {
        System.out.println(line);
        line = br.readLine();
    }
}
catch(IOException exception) { … }
```

Try with resources

BufferedReader with System.in

```java
import java.io.*;
…
   InputStreamReader input =
       new InputStreamReader(System.in);
   BufferedReader reader = new BufferedReader(input);

   try{
      System.out.print("Enter a string: ");
      String test = reader.readline();
   }
   catch(IOException e){
      e.printStackTrace();
   }
```

Because a Buffered Reader is a wraper of a Reader we can use Buffered Reader in combination with an Input stream reader. So now we can use User Input also with Buffer Reader methods. However it will read everything as a String. Integers and such must be Casted by the developer.

Therefore, in this particular example the Scanner object would have been more useful since it has methods to specifcally read as an Int (nextInt).

Reader vs. Scanner

So why not use a BufferedReader with a FileReader this instead of Scanner..?

It depends.

Reader
- Faster
- More efficient (buffer)
- Designed for reading characters

Scanner
- Slower
- More user friendly
- Flexible; built in support for **int**, **char**, etc.

For a bytestream the reader is better than the scanner.

WRITERS

`public abstract void write(char cbuf[], int off, int len) throws IOException`

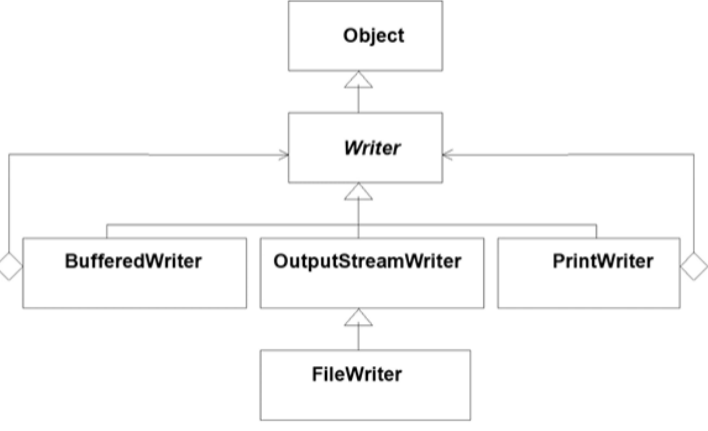
Write a portion of an array of characters.

`public void write(int c) throws IOException`

Write a single character.

`public void write(String str) throws IOException`

Write a string of characters



Writer (2)

```
public abstract void flush() throws
   IOException
```

Flush the stream. If the stream has saved any characters from the various write() methods in a buffer, write them immediately to their intended destination.

So, if you don't immediately see output on the screen, it might be because it is "stuck" in the buffer!

PrintWriter

```
public void print(boolean b)
public void print(char c)
public void print(char s[])
public void print(double d)
public void print(float f)
public void print(int i)
public void print(long l)
public void print(String s)
public void print(Object obj)
```

The circled implementations are the ones recommended for CSE1100. Using a Buffered Reader (4) or a Scanner depends on: Strings = Reader; Other = Scanner (especially numbers).

For Writting use PrintWriter because it has support for most types used in CSE1100; It does use a buffer.

WRITE FILE

```
public static void write(String str){
       try{
          PrintWriter writer = new PrintWriter("src\\file.txt");
          writer.print(str);
          writer.close();
      System.out.printf("File written");
   }
   catch(FileNotFoundException e){
      System.out.printf("File not found");
   }
}
```

WEEK 6 – TUTORIAL

1 Basic Inheritance

Create a class `Building`. A building has a `String street` and an `int value`. Make sure you encapsulate these properly. A building's street should never be able to change, but a building's value can change. Also, give the class a constructor that initialises all fields.

Create a subclass of `Building`: `House`. A house should have an `int number` with a getter. Give this class constructor that initialises all fields too.

Create another subclass of `Building`: `Office`. An office has an `int amountOfWorkers` with a getter and a setter. Give this class a constructor that initialises all the fields.

2 Distribution of Logic

You are given three classes `Person`, `Student` and `Teacher`. Inspect these classes. You can see there is a lot of code duplication. Rewrite these classes to use inheritance and to minimilise code duplication.

Implement the `equals` and `toString` methods for all three classes. `equals` should check *all* properties, e.g. two `Students` are equal if and only if their `name`, `height` and `livesInDelft` properties are equal. `toString` should return a `String` with *all* properties in a human readable way. It should also be clear from `toString` on which class it is being called. An example `toString` output might be:

```
Student:
Thomas is 1.9 metres tall and lives in Delft
```

3 Interface

Copy your `Person`, `Student` and `Teacher` classes from 'part 2: Distribution of Logic'. You are given an interface `HasToStudy`. Make `Student` implement `HasToStudy` in the following way:

- The `Student` keeps track of how many times `study` is called.
- If and only if `study` has been called at least 5 times, `willPassExam` should return true.

3.1 Testing

Test the `Student` class. Write a test where `willPassExam` returns `true` and a where `willPassExam` returns `false`. Write at least one test for every other method as well.

```java
/**
 * The type Building.
 */
public class Building {

    private final String street;
    private int value;

    /**
     * Instantiates a new Building.
     *
     * @param street the street
     * @param value  the value
     */
    public Building(String street, int
value) {
        this.street = street;
        this.value = value;
    }
}

/**
 * The type House.
 */
public class House extends Building {

    private int houseNumber;

    /**
     * Instantiates a new House.
     *
     * @param street      the street
     * @param value       the value
     * @param houseNumber the house
number
     */
    public House(String street, int
value, int houseNumber) {
        super(street, value);
        this.houseNumber = houseNumber;
    }

    /**
     * Get house number int.
     *
     * @return the int
     */
    public int getHouseNumber(){
        return this.houseNumber;
    }
}
/**
 * The type Office.
 */
public class Office extends House {

    private int amountOfWorkers;

    /**
     * Instantiates a new Office.
     *
     * @param street      the street
     * @param value       the value
     * @param houseNumber the house
number
     */
    public Office(String street, int
value, int houseNumber) {
        super(street, value,
houseNumber);
        this.amountOfWorkers =
getAmountOfWorkers();
    }

    /**
     * Get amount of workers int.
     *
     * @return the int
     */
    public int getAmountOfWorkers(){
        return this.amountOfWorkers;
    }

    /**
     * Set amount of workers.
     *
     * @param n the n
     */
    public void setAmountOfWorkers(int
n){
        this.amountOfWorkers = n;
    }
}
```

```java
import java.util.Objects;
/**
 * The type Person.
 */
public class Person {

    private String name;
    private double height;

    /**
     * Creates a person.
     *
     * @param name   The name of the person
     * @param height The height of the person
     */
    public Person(String name, double height) {
        this.name = name;
        this.height = height;
    }

    /**
     * Gets the name of the person.
     *
     * @return This person's name
     */
    public String getName() {
        return name;
    }

    /**
     * Gets the height of the person.
     *
     * @return This person's height
     */
    public double getHeight() {
        return height;
    }

    /**
     * Sets the height of the person.
     *
     * @param height The new height for this person
     */
    public void setHeight(double height) {
        this.height = height;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Person)) return false;
        Person person = (Person) o;
        return Double.compare(person.height, height) == 0 &&
                name.equals(person.name);
    }

    @Override
    public String toString() {
        return this.getClass().getTypeName() + ":\n" +
                name + " is " + height + " meters tall";
    }
}
```

```java
import java.util.Objects;

/**
 * The type Student.
 */
public class Student extends Person {

    private boolean livesInDelft;

    /**
     * Creates a student.
     *
     * @param name          The name of the student
     * @param height        The height of the student
     * @param livesInDelft Whether the student lives in Delft
     */
    public Student(String name, double height, boolean livesInDelft) {
        super(name,height);
        this.livesInDelft = livesInDelft;
    }

    /**
     * Gets whether the student lives in Delft.
     *
     * @return True iff this student lives in Delft
     */
    public boolean getLivesInDelft() {
        return livesInDelft;
    }

    /**
     * Sets whether the student lives in Delft.
     *
     * @param livesInDelft The new living status of this student
     */
    public void setLivesInDelft(boolean livesInDelft) {
        this.livesInDelft = livesInDelft;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Student)) return false;
        if (!super.equals(o)) return false; //checks if they are equal in Person fields.
        Student student = (Student) o;
        return livesInDelft == student.livesInDelft;
    }

    @Override
    public String toString() {
        String delft = "";
        if (getLivesInDelft()) {delft = "does";} else {delft = "does not";}
        return super.toString() + " and " + delft + " live in Delft";
    }
}
```

```java
import java.util.Objects;

/**
 * The type Teacher.
 */
public class Teacher extends Person {

    private int amountOfCourses;

    /**
     * Creates a new teacher.
     *
     * @param name              The name of the teacher
     * @param height            The height of the teacher
     * @param amountOfCourses The amount of courses the teacher teaches
     */
    public Teacher(String name, double height, int amountOfCourses) {
        super(name,height);
        this.amountOfCourses = amountOfCourses;
    }

    /**
     * Gets the amount of courses the teacher teaches.
     *
     * @return The amount of courses this teacher teaches
     */
    public int getAmountOfCourses() {
        return amountOfCourses;
    }

    /**
     * Sets the amount of courses the teacher teaches.
     *
     * @param amountOfCourses The new amount of courses this teacher teaches
     */
    public void setAmountOfCourses(int amountOfCourses) {
        this.amountOfCourses = amountOfCourses;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Teacher)) return false;
        if (!super.equals(o)) return false;
        Teacher teacher = (Teacher) o;
        return amountOfCourses == teacher.amountOfCourses;
    }

    @Override
    public String toString() {
        return super.toString() + " and teaches " + getAmountOfCourses() + " courses";
    }
}
```

```java
/**
 * The interface Has to study.
 */
public interface HasToStudy {

    /**
     * Study.
     */
    void study();

    /**
     * Will pass exam boolean.
     *
     * @return the boolean
     */
    boolean willPassExam();

}

import java.util.Objects;

/**
 * The type Student.
 */
public class Student extends Person
implements HasToStudy{

    private boolean livesInDelft;
    private int count;

    /**
     * Creates a student.
     *
     * @param name         The name of
the student
     * @param height       The height of
the student
     * @param livesInDelft Whether the
student lives in Delft
     */
    public Student(String name, double
height, boolean livesInDelft) {
        super(name,height);
        this.livesInDelft = livesInDelft;
        count = 0;
    }

    /**
     * Gets whether the student lives in
Delft.
     *
     * @return True iff this student
lives in Delft
     */
    public boolean getLivesInDelft() {
        return livesInDelft;
    }

    /**
     * Sets whether the student lives in
Delft.
     *
     * @param livesInDelft The new living
status of this student
     */
    public void setLivesInDelft(boolean
livesInDelft) {
        this.livesInDelft = livesInDelft;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Student))
return false;
        if (!super.equals(o)) return
false; //checks if they are equal in
Person fields.
        Student student = (Student) o;
        return livesInDelft ==
student.livesInDelft;
    }

    @Override
    public String toString() {
        String delft = "does not";
        String pass = "not pass";
        if (getLivesInDelft()) {delft =
"does";}
        if (willPassExam()) {pass =
"pass";}
        return super.toString() + ", " +
delft + " live in Delft and has studied "
+ count + " times. "
                + "Therefore he will " +
pass;
    }

    /**
     * Studies for the upcoming exam.
     */
    @Override
    public void study() {
        this.count++;
    }

    /**
     * Gets whether the next exam will be
passed.
     *
     * @return True iff the next exam
will be passed
     */
    @Override
    public boolean willPassExam() {
        return this.count >= 5;
    }
}
```

WEEK 7 - QUALITY, STRINGS, GENERICS & THREADS

QUALITY

External quality
- Mainly the user perspective
  - No crashes, good performance, …

Internal quality
- Mainly the developer perspective
  - Code is understandable
  - Code is easy to change
  - Code is testable (which has a link with external quality)

Interaction!

1. Number of Lines of Code (per method)

- Number of Lines of Code (LOC)

<= 20 LOC	>20 LOC	>40 LOC
Should be ok	Review & consider refactor	Split up

- *Rule of thumb:* if you need additional comments to explain what the next piece of code does, you should think of starting a new method.

- Note, these guidelines are approximate
- LOC is language dependent
  - Java is more lengthy than many other languages
- Conventions about
  - putting **{** on a new line
  - counting comments as LOC
  - etc.

2. Duplicated code

- Duplicated code ("copy-paste code") is "**bad**"
  - You are not only copying code, also potential bugs!
  - If you need to make changes in 1 location, chances are that you now have to make changes in multiple locations
- Don't duplicate, but put the code in a separate method that you call from multiple places
  - Although it is also clear that sometimes abstracting code in a method is difficult to impossible.
- Let's look at an example with inheritance…

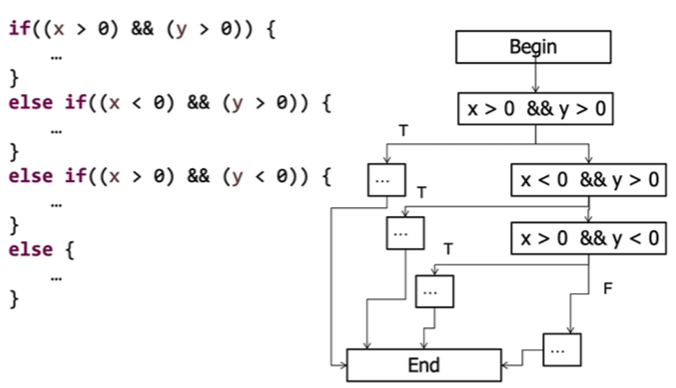
You are bound to forget make changes to these "clones", which leads to a **"late propagation".** A bug caused because you forgot to make changes in all locations that you copied code to/from

```
class Intercity extends Train {
   …
   public boolean equals(Object other) {
      return (other instanceof Intercity) && super.equals(other);
}
```

Code duplication with inheritance

- Think about the division of functionality between child classes and their parents.
- Only make child classes responsible for logic they "own"

3. Cyclomatic complexity

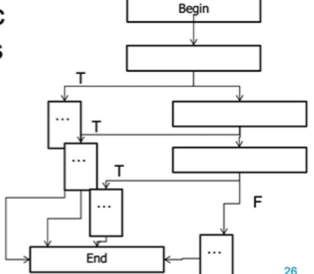


3. Cyclomatic complexity (2)

- How many linearly independent paths exist in this piece of code?
  → 4
- That is what the cyclomatic complexity (CC) expresses

A quick calculation of CC is done by counting the conditions (of if, while, for, …) and adding 1

In this example: 3 + 1 = 4



26

3. Cyclomatic complexity (3)

- Why is this score important?
  - High CC implies difficult to understand
  - High CC implies difficult to test
    - Each possible path needs to be tested
- Opinions vary:
  - CC between 0-5: perfect
  - CC 6-10: difficult to understand/test, think about restructuring/refactoring
  - CC > 30: extremely complex

4. Avoid long parameter lists

- ```java
  public void calculateDistance(
   int x1, int y1, int z1,
   int x2, int y2, int z2)
  ```
- Could also be:
  ```java
  public void calculateDistance(
      Point point1, Point point2)
  ```
- In other words, create new classes if you encounter parameter lists with a logical grouping!
- What is too long?!

it depends...

5. Check the style of your code

- Follow code conventions (indentation, whitespaces, variable/method naming, …)
- Provide and maintain documentation
  - Keep methods short, then the method name becomes self-describing
  - Ensure meaningful identifier names for variables
- Use whitespace to make clear divisions in your code
- Write Javadoc comments

Javadoc comments contain:
- A short description of the method on first line(s).
- Description of all arguments with **@param** keyword (if applicable)
- Description of what is returned with **@return** keyword (if applicable)
  ```

6. Testing

- For this course we say one test per method is good.

BUT

If your method has a cyclomatic complexity of 6, then you would need to write at least 6 tests to cover each path that runs through your code

- What is easier to test: a short or a long method?

There are tools that can help

checkstyle

- Static analysis tools
- Inspects your code and gives warnings about:
  - Unused variables
  - Code style
  - Cyclomatic complexity
  - etc.

There are tools that can help (2)

- Software project managers
  - Manages dependencies
  - Automated project building
  - Automated static analysis checking
- HUGE help when working in a team

STRINGS

Scanner

- Works with a String, but also with a file (Stream), …
- With a Scanner you can **tokenize** a String, Stream, …
  - A token can be a String, but also numbers, booleans, etc.
  - You can test for the next token:
    - `hasNext()` → String?
    - `hasNextBoolean()` → boolean?
    - `hasNextInt()` → integer?
    - …
  - Subsequently read with:
    - `next()` // a String
    - `nextLine()` // a String containing a line
    - `nextInt()` // an int
    - …
- But you can provide your own delimiter:

```
Scanner(input).useDelimiter(":");
```

Scanner (3)

```
String input = "1 fish 2 fish red fish blue fish";
Scanner s = new
   Scanner(input).useDelimiter("\\s*fish\\s*");
System.out.println(s.nextInt());
System.out.println(s.nextInt());
System.out.println(s.next());
System.out.println(s.next());
s.close();
```

→ 1
2
red
blue

https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Scanner.html

56

StringTokenizer

- Resembles a Scanner, but only works with Strings and also only returns Strings

```
StringTokenizer st =
   new StringTokenizer("this is a test");
while (st.hasMoreTokens()) {
    System.out.println(st.nextToken());
}
```

→ this
is
a
test

String.split()

- Is very similar to StringTokenizer, only the result is presented differently
- Tokenizer *enumerates*
- String.split returns an array
  - Can be useful if you need to 4th field/token…

```
String[] result = "this is a test".split("\\s");
for (int x=0; x < result.length; x++)
   System.out.println(result[x]);
```

CLASS SCANNER

java.lang.Object
java.util.Scanner

All Implemented Interfaces:

Closeable, AutoCloseable, Iterator<String>

```
public final class Scanner
extends Object
implements Iterator<String>, Closeable
```

A simple text scanner which can parse primitive types and strings using regular expressions.

A `Scanner` breaks its input into tokens using a delimiter pattern, which by default matches whitespace. The resulting tokens may then be converted into values of different types using the various `next` methods.

For example, this code allows a user to read a number from `System.in`:

```
Scanner sc = new Scanner(System.in);
int i = sc.nextInt();
```

As another example, this code allows `long` types to be assigned from entries in a file `myNumbers`:

```
Scanner sc = new Scanner(new File("myNumbers"));
while (sc.hasNextLong()) {
    long aLong = sc.nextLong();
}
```

The scanner can also use delimiters other than whitespace. This example reads several items in from a string:

```
String input = "1 fish 2 fish red fish blue fish";
Scanner s = new Scanner(input).useDelimiter("\\s*fish\\s*");
System.out.println(s.nextInt());
System.out.println(s.nextInt());
System.out.println(s.next());
System.out.println(s.next());
s.close();
```

prints the following output:

```
1
2
red
blue
```

The same output can be generated with this code, which uses a regular expression to parse all four tokens at once:

```
String input = "1 fish 2 fish red fish blue fish";
Scanner s = new Scanner(input);
s.findInLine("(\\d+) fish (\\d+) fish (\\w+) fish (\\w+)");
MatchResult result = s.match();
for (int i=1; i<=result.groupCount(); i++)
    System.out.println(result.group(i));
s.close();
```

The default whitespace delimiter used by a scanner is as recognized by `Character.isWhitespace()`. The `reset()` method will reset the value of the scanner's delimiter to the default whitespace delimiter regardless of whether it was previously changed.

A scanning operation may block waiting for input.

The `next()` and `hasNext()` methods and their companion methods (such as `nextInt()` and `hasNextInt()`) first skip any input that matches the delimiter pattern, and then attempt to return the next token. Both `hasNext()` and `next()` methods may block waiting for further input. Whether a `hasNext()` method blocks has no connection to whether or not its associated `next()` method will block. The `tokens()` method may also block waiting for input.

The `findInLine()`, `findWithinHorizon()`, `skip()`, and `findAll()` methods operate independently of the delimiter pattern. These methods will attempt to match the specified pattern with no regard to delimiters in the input and thus can be used in special circumstances where delimiters are not relevant. These methods may block waiting for more input.

When a scanner throws an `InputMismatchException`, the scanner will not pass the token that caused the exception, so that it may be retrieved or skipped via some other method.

Depending upon the type of delimiting pattern, empty tokens may be returned. For example, the pattern `"\\s+"` will return no empty tokens since it matches multiple instances of the delimiter. The delimiting pattern `"\\s"` could return empty tokens since it only passes one space at a time.

A scanner can read text from any object which implements the `Readable` interface. If an invocation of the underlying readable's `read()` method throws an `IOException` then the scanner assumes that the end of the input has been reached. The most recent `IOException` thrown by the underlying readable can be retrieved via the `ioException()` method.

When a `Scanner` is closed, it will close its input source if the source implements the `Closeable` interface.

A `Scanner` is not safe for multithreaded use without external synchronization.

Unless otherwise mentioned, passing a `null` parameter into any method of a `Scanner` will cause a `NullPointerException` to be thrown.

A scanner will default to interpreting numbers as decimal unless a different radix has been set by using the `useRadix(int)` method. The `reset()` method will reset the value of the scanner's radix to `10` regardless of whether it was previously changed.

GENERICS

The array list is a generic, because you specify the type if the object you want to store.

To do, re-organize slides. Then make Threads on IntelliJ, copy and paste the 2 types of implementations here and update the table of contents accordingly, then do the tutorial, update again.

Assume: this *great* class

```
public class IntegerList
{
   …
   public Integer sum() {
    // returns the sum of all integers in the list
   }

   public Integer product() {
    // returns the product of all integers in the list
   }
   …
}
```

But I also want to have such a list for "BigDecimal" instead of "Integer"

Are you gonna do everything again for the other type? (Nope)

Old way to do it:

ArrayList stringlist2 = new ArrayList(); //ArrayList without type specified, elements are stored as Object

stringlist2.add(new String("Student 1")); //Allowed since Student is a child class of Object
stringlist2.add(new String("Student 2"));
String content2 = (String) stringlist2.get(0); //Cast back to String when retrieving element from list. Will raise an exception at runtime

Solutions

- I will faithfully implement "BigDecimalList"
  - Will work, but introduces code duplication…
- I make sure to declare everything with Objects, then I can decide at runtime whether to use an IntegerList or a BigDecimalList
  - public Object sum() {…}
  - Not really a "clean" solution… you need to cast everything
- I can make use of **generics**
  - MyList<Integer>, MyList<BigDecimal>, …

Java Generics

(C++ calls this templating)

Specify that this list can only contain String

```
ArrayList<String> stringlist = new ArrayList<String>();
stringlist.add(new String("Student 1"));
stringlist.add(new String("Student 2"));
String content = stringlist.get(0);
```

No cast required!

Can you make this yourself?

```
public class Box<T> {
  private T t;

  public void add(T t) {
   this.t = t;
  }

  public T get() {
   return t;
  }
}
```

Generics and inheritance

- Is this possible?
```
List<String> ls = new ArrayList<String>(); //1
List<Object> lo = ls; //2
```

- No!
  - Suppose it would work, then the following situation would be possible:
    - `lo.add(new Object());`
    - `String s = ls.get(0);` → ls and lo point to the same List object, which is why you would expect a String, but you've actually put in an Object
  - The Java compiler will not allow statement 2!

Type erasure (2)

```
Pair<String, Integer> p1 = new Pair<>("foo", 1);
Pair<Integer, String> p2 = new Pair<>(1, "foo");

System.out.println((p1 instanceof Pair)); //true
System.out.println((p2 instanceof Pair)); //true
```

Runtime doesn't know / care about Integer or String
A pair is a pair

Autoboxing

- All eight primitve types have "wrapper" classes
  - `Integer` for `int`
  - `Character` for `char`
  - etc.
- You can create an: `ArrayList<Integer>`
- Java can automatically "wrap" and "unwrap" primitives
- This is called "**autoboxing / -unboxing**"

Type erasure

Generic arguments are compile-time only. The runtime does not know about type specializations.

```
public class Pair<A, B> {
    private A a;
    private B b;
}
```

→

```
public class Pair {
    private Object a;
    private Object b;
}
```

Runtime doesn't know about types A and B

Type constraints

- We can place constraints to generic type arguments

```
public class Pair<A, B extends Number> {
    private A a;
    private B b;
}

Pair<String, Integer> p1 =
   new Pair<>("foo", 1); // ok

Pair<Integer, String> p2 =
   new Pair<>(1, "foo"); // fails
```

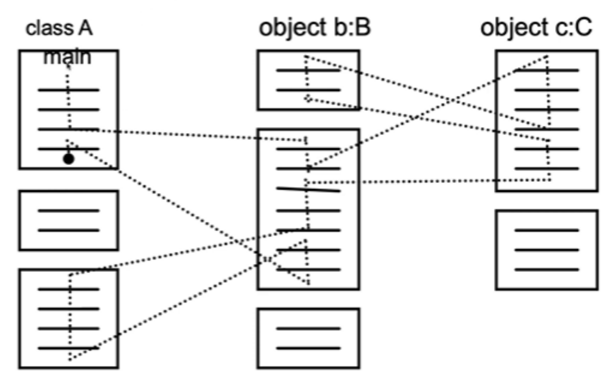
THREADS

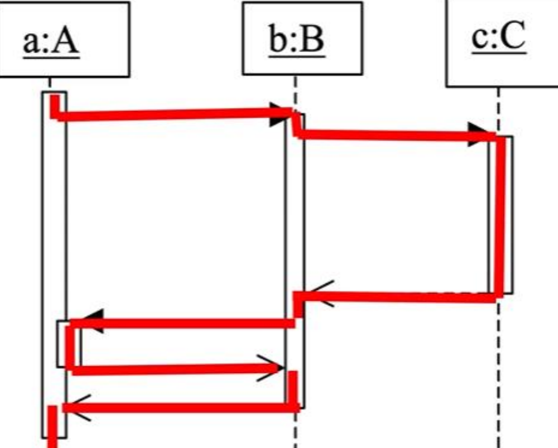
- You want to do multiple things at the same time within the same program
  - You are writing an email in Outlook and meanwhile Outlook checks whether there are any new email messages on the server, it synchronizes your calendar, …
  - A webserver wants to serve web pages to multiple persons (browsers) at the same time
- You stay within the same program ("process")

Threads versus processes

- A process ≈ a running program
- In a computer multiple processes run in parallel
  - This way, your computer can do multiple things "at the same time"
- Normally, each process has its own piece of private memory
  - You can pass information between processes (e.g., through *sockets*, which we'll see later)
- Within 1 process you can also do things "at the same time"
  - Threads!
  - You are working in the same piece of memory

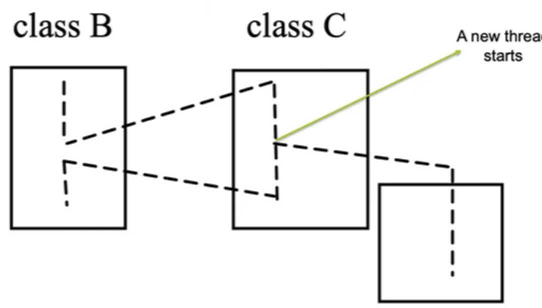
Thread of control





Integration diagrams

A second thread of control



Concurrency vs parallelism

- Concurrency
  - Doing many things at the same time
- Parallelism
  - Splitting one big job on multiple processors

Doing things in parallel

- Within 1 program
- How does this work?
  - With 1 processor? → interleaving
    - Each thread gets a bit of time
    - Order of threads is not pre-determined (varies per time that you execute, varies per operating system, …) →Heisenbugs → bugs that appear because of different expectations w.r.t. thread scheduling
  - With > 1 processor/core
    - Threads can run on different cores in parallel
  - With processors that have *hyperthreading* (e.g., Intel Core i5/i7 and Pentium IV)
    - Certain parts of the processor, responsible for the state, are duplicated. The actual execution resources are not duplicated.

With 1 processor you cant really do things on parallel, but because the cpu alternates so fast between processes it still feels to the user that it is executing things in parallel.

The last option allows the processor to duplicate and share some parts i.e. registers so that both threads can borrow things from the same processor while they are being executed o parallel.

Multi-threading

- Not necessary
- Risky: a lot can go wrong
- Useful for *trivially parallelizable* problems
  - Serving web pages
  - Processing multiple files with the same algorithm, e.g. converting a directory of WAVs to MP3s
  - Accepting user input while waiting for, e.g., a web page to load

CLASS THREAD

```
public class Thread
extends Object
implements Runnable
```

A *thread* is a thread of execution in a program. The Java Virtual Machine allows an application to have multiple threads of execution running concurrently.

Every thread has a priority. Threads with higher priority are executed in preference to threads with lower priority. Each thread may or may not also be marked as a daemon. When code running in some thread creates a new `Thread` object, the new thread has its priority initially set equal to the priority of the creating thread, and is a daemon thread if and only if the creating thread is a daemon.

When a Java Virtual Machine starts up, there is usually a single non-daemon thread (which typically calls the method named `main` of some designated class). The Java Virtual Machine continues to execute threads until either of the following occurs:

- The `exit` method of class `Runtime` has been called and the security manager has permitted the exit operation to take place.
- All threads that are not daemon threads have died, either by returning from the call to the `run` method or by throwing an exception that propagates beyond the `run` method.

There are two ways to create a new thread of execution. One is to declare a class to be a subclass of `Thread`. This subclass should override the `run` method of class `Thread`. An instance of the subclass can then be allocated and started. For example, a thread that computes primes larger than a stated value could be written as follows:

```
class PrimeThread extends Thread {
    long minPrime;
    PrimeThread(long minPrime) {
        this.minPrime = minPrime;
    }

    public void run() {
        // compute primes larger than minPrime
         . . .
    }
}
```

The following code would then create a thread and start it running:

```
PrimeThread p = new PrimeThread(143);
p.start();
```

The other way to create a thread is to declare a class that implements the `Runnable` interface. That class then implements the `run` method. An instance of the class can then be allocated, passed as an argument when creating `Thread`, and started. The same example in this other style looks like the following:

```
class PrimeRun implements Runnable {
    long minPrime;
    PrimeRun(long minPrime) {
        this.minPrime = minPrime;
    }

    public void run() {
        // compute primes larger than minPrime
         . . .
    }
}
```

The following code would then create a thread and start it running:

```
PrimeRun p = new PrimeRun(143);
new Thread(p).start();
```

Every thread has a name for identification purposes. More than one thread may have the same name. If a name is not specified when a thread is created, a new name is generated for it.

Unless otherwise noted, passing a `null` argument to a constructor or method in this class will cause a `NullPointerException` to be thrown.

THE CLASS THREAD

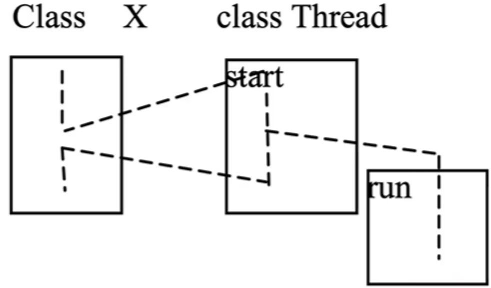
The class Thread

- This class provides methods for constructing and managing threads
- The Thread class is not a thread itself
  - `public Thread(String name)`
  - We don't have a new thread yet, but the thread has a name.
- The class Thread contains (amongst others) this method:
  - `public native synchronized void start()`
  - This is where the new thread is started
  - But what will this new thread do?

What should the new thread do?

- The class Thread has a method
  - `public void run()`
- That is (almost) empty!
- Redefine it in a subclass
- The instructions in `run()` are the things that this thread will do
- BUT: you call `run()` through `start()`

A second thread



A SIMPLE EXAMPLE

A subclass of Thread

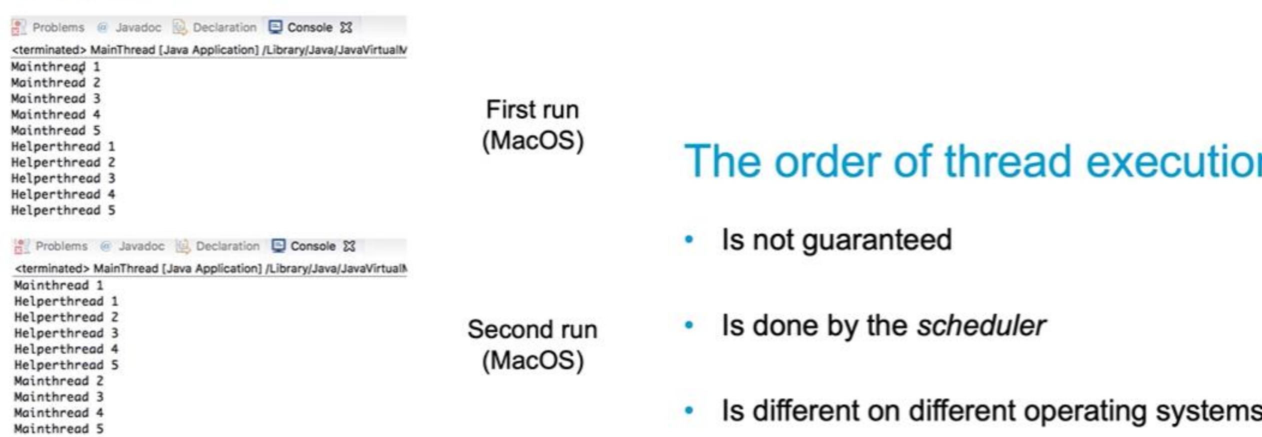
```
public class HelperThread extends Thread{
   public HelperThread(String name){
      super(name);
   }

   public void run(){
      System.out.println("Helperthread 1");
      System.out.println("Helperthread 2");
      System.out.println("Helperthread 3");
      System.out.println("Helperthread 4");
      System.out.println("Helperthread 5");
   }
}
```

The main program

```
public static void main(String[] args){
      HelperThread h = new
         HelperThread("Helperthread");
      h.start();
      System.out.println("Mainthread 1");
      System.out.println("Mainthread 2");
      System.out.println("Mainthread 3");
      System.out.println("Mainthread 4");
      System.out.println("Mainthread 5");
}
```

Output



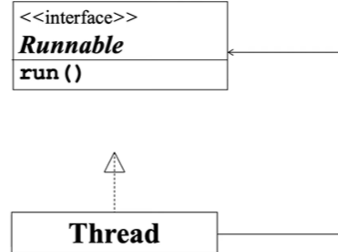
scheduler is part of the operating system, java virtual machine cant do nothing about it.

My class already inherits

- Can I start a thread in, e.g., a Frame?
- Not like this:
  - `public class PopUpFrame extends Frame extends Thread`
- How can we make this work?

THE INTERFACE RUNNABLE

The interface Runnable



Class Thread

- Has an attribute target:
  - `private Runnable target;`
- Constructors of Thread:
  - `Thread()`
    - Allocates a new Thread object.
  - `Thread(Runnable)`
    - Allocates a new Thread object.
  - …

Method run in Thread

If this thread was constructed using a separate Runnable object, then that Runnable object's run method is called; otherwise, this method does nothing and returns. Subclasses of Thread should override this method.

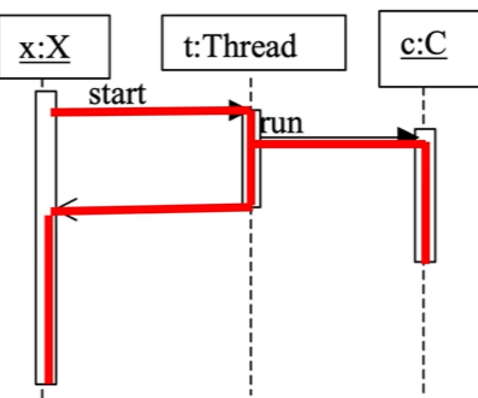
```
public void run() {
 if (target != null)
       target.run();
}
```

How to work with a Runnable?

```
public class Alternative implements Runnable {
   private Thread fThread;

   // Example: initialize and start in constructor
   public Alternative(){
        fThread = new Thread(this);
        // this implements Runnable!
        fThread.start();
   }
   public void run(){
    // do something
   }
}
```

The interface Runnable has 1 method
→ public void run()



The `Runnable` interface should be implemented by any class whose instances are intended to be executed by a thread. The class must define a method of no arguments called `run`.

This interface is designed to provide a common protocol for objects that wish to execute code while they are active. For example, `Runnable` is implemented by class `Thread`. Being active simply means that a thread has been started and has not yet been stopped.

In addition, `Runnable` provides the means for a class to be active while not subclassing `Thread`. A class that implements `Runnable` can run without subclassing `Thread` by instantiating a `Thread` instance and passing itself in as the target. In most cases, the `Runnable` interface should be used if you are only planning to override the `run()` method and no other `Thread` methods. This is important because classes should not be subclassed unless the programmer intends on modifying or enhancing the fundamental behavior of the class.

SYNCHRONIZATION PROBLEMS

Synchronization

- What happens if two threads access the same data at the same time?
- What happens if two teachers use the blackboard at the same time?

X = 0

- Thread 1

```
for(int k=0; k<100; k++) {
    a = x;
    a = a + 1;
    x = a;
}
```

- Thread 2

```
for(int k=0; k<100; k++) {
    b = x;
    b = b + 1;
    x = b;
}
```

X = 0

- Thread 1

```
for(int k=0; k<100; k++) {
    a = x;
    a = a + 1;
    x = a;
}
```

- Thread 2

```
for(int k=0; k<100; k++) {
    b = x;
    b = b + 1;
    x = b;
}
```

Synchronization

- Keyword: **synchronized**
- Within an object only 1 thread can be active in a synchronized method
  - So: as soon as 1 thread enters a synchronized method of an object O, no other thread can enter a synchronized method of O
- Deadlock ....

Deadlock

- I am thread T1
- I am busy in object o1
- I am executing the (synchronized) method m1
- In m1 there is a call:
  - o2.m3
- But method m3 is also synchronized
- So I have to wait

- I am thread T2
- I am busy in object o2
- I am executing the (synchronized) method m2
- In m2 there is a call:
  - o1.m4
- But method m4 is also synchronized
- So I have to wait

Chain reaction of waits! Which can lead to a full circle, where the program will never continue.

How to use synchronized

```
public class SynchronizedCounter {

    private int c = 0;

    public synchronized void increment()  {
        c++;
    }

    public synchronized void decrement() {
        c--;
    }
}
```

Alternative use of synchronized

```
public class SynchronizedCounter {

    private int c = 0;

    public void increment()  {
        synchronized(this) {
            c++;
        }
    }

    public synchronized void decrement() {
        c--;
    }
}
```

What to remember

- Threads allow for multiple parallel actions within 1 process
  - What is the difference between threads and processes?
- The order in which multiple threads are executed is not fixed
  - Heisenbugs
- The class thread, the interface Runnable
- Deadlock and the synchronized keyword

```java
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Objects;
import java.util.Scanner;

/**
 * The type Address.
 */
public class Address {
    private String city;
    private String zipCode;
    private String street;
    private int number;

    /**
     * Instantiates a new Address.
     *
     * @param city    the city
     * @param zipCode the zip code
     * @param street  the street
     * @param number  the number
     */
    public Address(String city, String zipCode, String street, int number) {
        this.city = city;
        this.zipCode = zipCode;
        this.street = street;
        this.number = number;
    }

    /**
     * Read address.
     *
     * @param sc the sc
     * @return the address
     */
    public static Address read(Scanner sc){
        if (sc == null) {throw new IllegalArgumentException("Wrong address format");}
        String street = ""; //declare and initlisie inside the if statement
        String zipCode = "";
        String city = "";
        if (sc.hasNext()) {
            street = sc.next();
            int number = sc.nextInt();
            zipCode = sc.next();
            city = sc.next();
            //System.out.println(street + " " + number + "\n" + zipCode + " " + city);
            Address address = new Address(city,zipCode,street,number);
            return address;
        }
        throw new IllegalArgumentException("Wrong address format");
    }


    /**
     * Gets city.
     *
     * @return the city
     */
    public String getCity() {
        return city;
    }
```

```
    /**
     * Gets zip code.
     *
     * @return the zip code
     */
    public String getZipCode() {
        return zipCode;
    }

    /**
     * Gets street.
     *
     * @return the street
     */
    public String getStreet() {
        return street;
    }

    /**
     * Gets number.
     *
     * @return the number
     */
    public int getNumber() {
        return number;
    }


    /**
     * Prints a human friendly address string
     *
     * @return the friendly String
     */
    @Override
    public String toString() {
        return street + " " + number + ", " +
                zipCode + " " + city;
    }


    /**
     * Equals method that compares all address elements
     *
     * @returns boolean of equal
     */
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Address)) return false;
        Address that = (Address) o;
        return this.getNumber() == that.getNumber() &&
                this.getZipCode().equals(that.getZipCode());
    }
}
```

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;
import static org.junit.jupiter.api.Assertions.*;

class AddressTest {
    private Address a1;
    private Address a2;
    private Scanner sc;
    private Scanner wrongSc;

    @org.junit.jupiter.api.BeforeEach
    void setUp() {
        try {
            sc = new Scanner(new File("src/address.txt"));
            wrongSc = new Scanner(new File("src/Waddress.txt"));
        } catch (FileNotFoundException e) {}
    }
    @org.junit.jupiter.api.Test
    void Address() {
        a1 = Address.read(sc);
        assertNotNull(a1,"Address()");
    }
    @org.junit.jupiter.api.Test
    void read() {
        assertThrows(IllegalArgumentException.class,
                ()->{
            a1 = Address.read(wrongSc);
                });
    }
    @org.junit.jupiter.api.Test
    void getCity() {
        a1 = Address.read(sc);
        assertEquals("Rotterdam",a1.getCity(),"getCity()");
    }
    @org.junit.jupiter.api.Test
    void getZipCode() {
        a1 = Address.read(sc);
        assertEquals("3051JC",a1.getZipCode(),"getZipCode()");
    }
    @org.junit.jupiter.api.Test
    void getStreet() {
        a1 = Address.read(sc);
        assertEquals("Emmalaan",a1.getStreet(),"getStreet()");
    }
    @org.junit.jupiter.api.Test
    void getNumber() {
        a1 = Address.read(sc);
        assertEquals(23,a1.getNumber(),"getNumber()");
    }
    @org.junit.jupiter.api.Test
    void toStringTest() {
        a1 = Address.read(sc);
        assertEquals("Emmalaan 23, 3051JC Rotterdam",a1.toString(),"toString()");
    }
    //notEqual, //equal same object //equal different object //Null
    @org.junit.jupiter.api.Test
    void equalsTest() {
        a1 = Address.read(sc);
        a2 = a1;
        assertEquals(a1,a2);
    }
}
```

```java
import java.util.Objects;
import java.util.Scanner;

/**
 * The type House.
 */
public class House {
    private Address address;
    private int nRooms;
    private int salePrice;

    /**
     * Instantiates a new House.
     *
     * @param address   the address
     * @param nRooms    the n rooms
     * @param salePrice the sale price
     */
    public House(Address address, int nRooms, int salePrice) {
        this.address = address;
        this.nRooms = nRooms;
        this.salePrice = salePrice;
    }

    /**
     * Gets address.
     *
     * @return the address
     */
    public Address getAddress() {
        return address;
    }

    /**
     * Gets rooms.
     *
     * @return the rooms
     */
    public int getnRooms() {
        return nRooms;
    }

    /**
     * Gets sale price.
     *
     * @return the sale price
     */
    public int getSalePrice() {
        return salePrice;
    }

    /**
     * Costs at most boolean.
     *
     * @param price the price
     * @return the boolean
     */
    public boolean costsAtMost(int price){
        return (price-this.salePrice >= 0);
    }
```

```
    /**
     * Read house.
     *
     * @param sc the sc
     * @return the house
     */
    public static House read(Scanner sc){
        if (sc == null) {throw new IllegalArgumentException("Wrong house format");}
        Address address = Address.read(sc);

        int nRooms = 0;
        int salePrice = 0;
        if (sc.hasNext()) {
            //sc.useDelimiter(","); https://stackoverflow.com/questions/28766377/how-do-
i-use-a-delimiter-with-scanner-usedelimiter-in-java
            nRooms = sc.nextInt();
            sc.next(); // skip (OK practice!)
            sc.next(); // skip
            salePrice = sc.nextInt();
            House house = new House(address,nRooms,salePrice);
            return house;
        }
        throw new IllegalArgumentException("Wrong house format");
    }


    /**
     * Prints a human friendly address, room and price string
     *
     * @return the friendly String
     */
    @Override
    public String toString() {
        return address +
                ": " + nRooms +
                " rooms, " + salePrice + " euros";
    }


    /**
     * Equals method that compares only the address elements and if it is a house object.
     *
     * @returns boolean of equal
     */
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof House)) return false;
        House house = (House) o;
        return getAddress().equals(house.getAddress());
    }
}
```

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.*;
/**
 * The type House catalog.
 */
public class HouseCatalog {
    private Set<House> houses;

    /**
     * Instantiates a new House catalog.
     */
    public HouseCatalog(){
        houses = new HashSet<>();
    }

    /**
     * Add house.
     *
     * @param house the house
     */
    public void addHouse(House house){
        houses.add(house);
    }

    public int getSize(){
        return houses.size();
    }

    /**
     * Houses cost at most list.
     * @param price the price
     * @return the list
     */
    public List<House> housesCostAtMost(int price){
        List<House> catalog = new ArrayList<House>();
        for(House arbitraryName : houses) {
            // for each House in houses (which is the class attribute: private Set<House> houses)
            if (arbitraryName.getSalePrice() <= price) //condition
                catalog.add(arbitraryName);
            // add the house (which we gave an arbitraryName to the house(s) in the Set.
        }
        return catalog;
    }

    /**
     * Read house catalog.
     * @param fileName the file name
     * @return the house catalog
     */
    public static HouseCatalog read(String fileName){
        try {
            Scanner sc = new Scanner(new File(fileName));
            if (!sc.hasNext()) throw new IllegalArgumentException("Invalid file");
            HouseCatalog catalog = new HouseCatalog();
            int n = sc.nextInt();
            for (int i = 0; i<n; i++) {
                House house = House.read(sc);
                catalog.addHouse(house);
            }
            sc.close();
            return catalog;
        }
        catch(FileNotFoundException e){
            throw new IllegalArgumentException("Invalid file");
        }
    }

    // should add javadoc
    @Override
    public String toString() {
        String result = "";
        for (House house : houses){
            result += house.toString() + "\n";
        }
        return result;
    }

}
```

```
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

import static org.junit.jupiter.api.Assertions.*;

class HouseCatalogTest {

    private Address a1;
    private Address a2;
    private House h1;
    private House h2;
    private HouseCatalog c;
    private Scanner sc;

    @BeforeEach
    void setUp() {
        try {
            sc = new Scanner(new File("src/address.txt"));
        } catch (FileNotFoundException e) {}
        c = HouseCatalog.read("src/houses.txt");
        h1 = House.read(sc);
    }

    @Test
    void HouseCatalog(){
        assertNotNull(c);
    }

    @Test
    void addHouse() {
        assertEquals(3,c.getSize(),"addHouse()");
    }

    @Test
    void housesCostAtMost() {
        assertEquals(2,c.housesCostAtMost(100000).size(),"housesCostAtMost()");
    }
}
import java.io.File;
import java.io.FileNotFoundException;
import java.util.List;
import java.util.Scanner;
public class HousingApplication {

    public static void main(String[] args) {
        HouseCatalog c1 = HouseCatalog.read("src/houses.txt");
        Scanner input = new Scanner(System.in);
        int n;
        do {
            System.out.println("Enter the maxium price. Zero to exit");
            n = input.nextInt();
            // get list of all houses at maximum price
            List<House> houseList = c1.housesCostAtMost(n);
            for(House house : houseList){
                System.out.println(house);
            }
        } while(n != 0);
    }
}
```

WEEK 8 - FUNCTIONAL JAVA

Imperative programming	Declarative programming
Order the computer how to execute operations line by line.	Tell the computer what you want mathematically, but don't describe how.

Imperative programming: explicitly construct a for loop and specify what happens there.

Declarative: give a mathematical definition and let the computer figure it out.

Imperative example (Java):

```java
List<Integer> input = Arrays.asList(3, 4, 5, 6, 7);
List<Integer> greater = new ArrayList<Integer>();
for (int i = 0; i < input.size(); i++) {
    Integer element = input.get(i);
    if (element > 5) {
        greater.add(element);
    }
}
```

Filters element and returns greater than 5

Declarative example (Haskell):

```haskell
input = [3, 4, 5, 6, 7]
greater = filter (> 5) input
```

The Lambda calculus (1)

- Possible notation of an increment function:

$$f(x) = x + 1$$

- Lambda notation:

$$\underbrace{\lambda x.}_{\text{Head}}\underbrace{x + 1}_{\text{Body}}$$

- Possible notation of an addition function:

$$f(x, y) = x + y$$

- Lambda notation:

$$\lambda x. \lambda y. x + y$$

Lambda functions in Java are based on this theory.

Pros of functional programming:

- math functions take fewer lines of code and it is more simple doing it in declarative notation (concise and readable)
- Makes it easier to run things in parallel
- Flexible and powerful code
- Easier to test

Cons:

- Math can become abstract
- It can be hard to translate a real world problem to a math notation

Examples of programming languages

- Haskell (purely functional)
- Scala (hybrid)
- Java (functional elements since version 8)

Lambda's in Java

- These functions (should) avoid state and mutable data (variables that change their values).
- They (should) always produce the same output for a given input.

- $\lambda x. x + 1$
  In Java: `x -> x + 1`
- $\lambda x. \lambda y. x + y$
- In Java: `(x, y) -> x + y`

FUNCTIONS

```
import java.util.function.Function;
public class Main {

    public static void main(String[] args) {
   Function<Integer, Integer> increment = x-> x + 2;
        System.out.println(increment.apply(3)); // prints 5
    }
}

import java.util.function.BiFunction;
...
   BiFunction<Integer, Integer, Integer> add = (x, y) -> x + y;
   //param 1 type, param 2 type, return type
   System.out.println(add.apply(3,2)); // prints 5
...
```

PREDICATE AND SUPPLIER

```
import java.util.function.Predicate;
import java.util.function.Supplier;
...

   Predicate<Integer> greaterThanFive = x -> x > 5;
   // = Function<Integer,Boolean> greaterThanFive = x -> x > 5;
    // Predicate is a function that returns a boolean by definition
   System.out.println(greaterThanFive.test(4)); // false

    Supplier<Integer> fourSupplier = () -> 4;
    // Doesn't take a parameter, you specify the return type.
```

STREAMS AND FILTERS

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class Main {

    public static void main(String[] args) {
        // create Integer List
        List<Integer> input = Arrays.asList(3,4,5,6,8);

        // A stream is the functional equivalent of a loop that gets get.element(i)
        // List<Type> has a built in method .stream(), that returns each of the elements
as a stream.
        // .collect(Collectors.toList()) returns the "modified by methods" stream back to
a list
        // then we combine stream method and returnToList:
        // "just an amount of data coming in, which we read 1 by 1", "apply a method to
each of those data", "return the resulting effect as a list".
        // List = stream.method.collect
        List<Integer> output = input.stream().filter(x -> x >
5).collect(Collectors.toList());
        System.out.println(output.toString());
    }
}
```

Stream pipelines

source	[3, 4, 5, 6, 7]
0 or more intermediate operations (lazy)	filter
terminal operation	toArray

Creating a stream from a source (1)

- From individual values:
```
Stream<String> nameStream = Stream.of("John", "Kate");
```
- From an array:
```
int[] numbers = { 3, 4, 5, 6, 7 };
IntStream numberStream = Arrays.stream(numbers);
```
- From a collection (such as a `List`):
```
List<String> names = Arrays.asList("John", "Kate");
Stream<String> nameStream = names.stream();
```

Creating a stream from a source (2)

- From a range:
```
IntStream range = IntStream.range(0, 4); // 0, 1, 2, 3
```
- From the lines of a file:
```
BufferedReader reader =
    new BufferedReader(new FileReader("input.txt"));
Stream<String> lines = reader.lines();
```
- Or you can create an infinite stream:
```
IntStream evenNumbers =
    IntStream.iterate(0, i -> i + 2);
```

Stream is evaluated “lazy” therefore Java will not fill up the memory of the stream until we try to collect the entire stream at the end.

So in the infinite iteration it will only generate up until the value that you ask Java to compute.

Intermediate operations (1)

Example input:
```
DoubleStream grades = DoubleStream.of(5.1, 6.2, 7.0);
```

- `filter`
```
// Result: 6.2, 7.0
DoubleStream passing = grades.filter(x -> x >= 5.75);
```
- `map`
```
// Result: 6.1, 7.2, 8.0
DoubleStream plusOne = grades.map(x -> x + 1);
```

map: all elements are adjusted and hence present in the collected stream afterwards.

Intermediate operations (2)

- `skip` and `limit`
  Can be used for paging (“results 21-30 from 941”):
```
DoubleStream paged = grades.skip(20).limit(10);
```
- `distinct`
```
// Result: 1, 3
IntStream distinctValues =
    IntStream.of(1, 3, 3).distinct();
```
- `sorted`
```
// Result: 3, 4, 5
IntStream sorted = IntStream.of(5, 4, 3).sorted();
```

distinct: stream contains only unique values (non-duplicates).

Terminal operations (1)

- allMatch, anyMatch, noneMatch

```
// Probably false:
boolean allPassed = grades.allMatch(x -> x >= 5.75);
// Hopefully true:
boolean anyonePassed = grades.anyMatch(x -> x >= 5.75);
```

- findFirst

```
OptionalInt firstEven =
    IntStream.of(3, 4, 5, 6)
        .filter(x -> x % 2 == 0)
        .findFirst(); // Finds 4 (the first even number).
```

allMatch: returns a boolean, true if all elements meet the criteria

anyMatch: returns true if at least 1 meets

- max, sum, count

```
OptionalInt max = IntStream.of(3, 4, 5, 6).max(); // 6
int sum = IntStream.of(3, 4, 5, 6).sum();         // 18
long count = IntStream.of(3, 4, 5, 6).count();    // 4
```

Turn your stream back into a collection:

- `stream.toArray()`
- `stream.collect(Collectors.toList());`

Streams are unmodifiable, when we are doing the operation, stream().filter().toArray() we are actually "consuming" an existing stream and creating a new one, but the original remains intact.

Suppose we have a stream of Songs that have been performed by Artists. We can 'query' that stream as follows:

Different notation for: x -> Song.getArtist(x)

```
List<String> result = songs
        .map(Song::getArtist)
        .filter(artist -> artist.getYearOfBirth() > 1960)
        .map(Artist::getName)
        .distinct()
        .sorted(Comparator.reverseOrder())
        .limit(3)
        .collect(Collectors.toList());
```

- Streams can only be traversed once.
- Stick to "pure functions" when working with your stream (don't update variables; just return data).

NullPointerException...

```
String color = person.getCar().getColor();
```

What if a person does not have a car?
→ NullPointerException

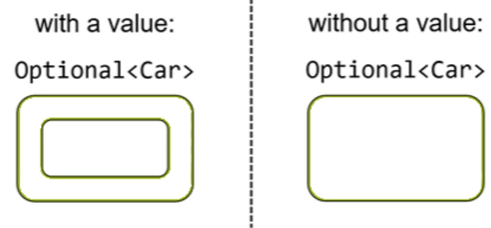
With a null check:

```
String color;
Car car = person.getCar();
if (car != null) {
    color = car.getColor();
}
else {
    color = "none";
}
```

... or use the Optional class!

The Optional class

- This container object either contains a value, or does not.



We can wrap a container around our car: Optional<Car>

We always have an object, which may or may not have a car in it. The behaviour would change

Specifying an optional result

```
import java.util.Optional;

public class Person {
    private Optional<Car> car;

    public Person() {
        this.car = Optional.empty();
    }

    public Person(Car car) {
        this.car = Optional.of(car);
    }

    public Optional<Car> getCar() {
        return car;
    }
}
```

Handling an optional result

```
String color = person.getCar().map(x ->
    x.getColor()).orElse("none");
```


Some stream methods give an optional result, because the stream might be empty. For example:

- `OptionalDouble maxGrade = grades.max();`
- ```
  Optional<Person> result = persons
        .filter(x -> x.getName().startsWith("A"))
        .findFirst();
  ```

Compared to the version with the null check:

- We can now write our code more concisely.
- We are forced to handle the situation where a person does not have a car.

VAR

Annoyances with declaring variables

This takes a lot of typing:

```
ArrayList<String> list = new
ArrayList<String>();
```

and this too:

```
Optional<Car> car = person.getCar();
```

What if Java could just infer the types for you?

The var keyword (since Java 10)

You can now write:

```
var list = new ArrayList<String>();
```

and also:

```
var car = person.getCar();
```

var vs. the type system

- The types are still static.
- If Java cannot figure out the type, it will tell you (and your program will not compile).

```
var number = 3;
number = "hello";
```

Compiler error: "Cannot convert from String to int"

```
var even = x -> x % 2 == 0;
```

Compiler error: "Lambda expression needs an explicit target-type"

var in an IDE

In an IDE (such as Eclipse), you can find out the type by hovering over the **var** keyword or over the variable name.

In practie it may be better to not user var and just type everything out so its explicit what the type of the var is
```

Tutorial 7: IO

CSE1100 - Object Oriented Programming

1 Basic Reader

You are given a basic `Store` and `Product` class and a `StoreApplication`. The `StoreApplication` uses the `read` method in `Store` to read *resources/store.csv*. Implement the `read` method.

2 Basic Writer

You are given four basic classes. `WeatherReport` has a `write` method. This method should write the weather report to the given writer in the following format:

```
3/8/2020,RAIN,20C,1BFT,20mm
4/8/2020,SUN,25C,2BFT,12h
```

Here on August 3rd 2020 it was a rainy day. It was 20 degrees, there was a wind speed of 1 beaufort and 20 millimeters of rain fell. On August 4th it was a sunny day. It was 25 degrees, there was a wind speed of 2 beaufort and there were 12 hours of sun. One test is provided. Writing more tests is encouraged.

3 Complicated Reader

You are given the following classes:

- `CoffeeMenu`: This is a menu for a coffee bar. It contains all the coffees customers can order. It also contains all the additions a customer can put in their coffee, for example vanilla or sugar.
- `CoffeeItem`: This is an item on the menu, e.g. cappuccino or espresso. It contains a list allergies and a list of possible variations, like iced or double.
- `CoffeeVariation`: This is a variation on a coffee item.
- `Addition`: This is an addition a customer can potentially add to their order, regardless of what coffee they order.

Implement a the `read` method in `CoffeeMenu`. For the format see *resources/coffee.txt*. It starts with *COFFEES*, followed by the list of coffees. These in turn start with *COFFEE,name,price,optional list of allergies*, followed by the list of variations for that coffee. Each variation simply contains the name and extra cost. After all the coffees, the additions start with *ADDITION*. Each addition then is of the form *ADDITION,name,price,optional list of allergies*.

3.1 Testing

Verify your read method(s) work with tests. Write at least one test for reading every class.

4 Serializable

We want to make a brand new RPG in Java, so we have started with a `Character` class. This class has some attributes: `name`, `level` and `xpPoints`. It also has a special field `playerSecret`. This secret will be used for some internal magic in our game, but as you can see by the `equals` method it does not contribute to the equality of two characters. The secret will be randomly generated once for every character.

4.1 Reading and writing characters

Because we want to use this class in a game where it might make sense to not use too much storage space, we will store the characters as binary data. Have a look at the `testCharacterReadAndWrite` test. Try to understand what is happening.

Now run the test, it should fail. What did we forget in our `Character` class? Fix the `Character` class such that the test passes. Don't worry about the second test yet, we will fix that in **??**.

4.2 Keeping our secrets secret

Because we want to generate a new secret every time we save and load our game and because we do not want malicious users to have our secret, we do not ever want to save the secret to a file.

This is why we have the second test `testCharacterDoesNotSaveSecret`. Take a look at what it does and run it. This test will also not pass, but for a different reason: Java will by default store all fields of the class, including `playerSecret`. Luckily for us there is a keyword to fix this. Try to find out how we can fix our problem, without implementing a custom read/write method.

Basic Readers

```
public class Product {

    private String name;
    private String category;
    private double price;

    /**
     * Creates a product.
     *
     * @param name The name of the product
     * @param category The category of the product
     * @param price The price of the product
     */
    public Product(String name, String category, double price) {
        this.name = name;
        this.category = category;
        this.price = price;
    }

    /**
     * Gets the name of the product.
     *
     * @return This product's name
     */
    public String getName() {
        return name;
    }

    /**
     * Gets the category of the product.
     *
     * @return This product's category
     */
    public String getCategory() {
        return category;
    }

    /**
     * Gets the price of the product.
     *
     * @return This product's price
     */
    public double getPrice() {
        return price;
    }

    @Override
    public String toString() {
        return "Product{" +
                "name='" + name + '\'' +
                ", category='" + category + '\'' +
                ", price=" + price +
                '}';
    }
}
```

```java
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;
public class Store {

    /**
     * Reads a store from the given input.
     *
     * @param input The scanner to read from
     * @return The store read from the input
     */
    public static Store read(Scanner input) {
        Store store = new Store();
        input.useDelimiter(",|\\n");
        while(input.hasNextLine()){
            String name = input.next();
            String category = input.next();
            Double price = input.nextDouble();
            Product product = new Product(name,category,price);
           store.addProduct(product);
        }
        input.close();
        return store;
    }

    private List<Product> products;

    /**
     * Creates a store.
     */
    public Store() {
        this.products = new ArrayList<>();
    }

    /**
     * Adds a product to the store.
     *
     * @param product The product to add
     */
    public void addProduct(Product product) {
        products.add(product);
    }

    /**
     * Gets the products in the store.
     *
     * @return The list of products in this store
     */
    public List<Product> getProducts() {
        return products;
    }

    /**
     * Converts this store to a string representation of the form Store[products]
     *
     * @return The string representation of this store
     */
    @Override
    public String toString() {
        return "Store(products: " + products + ")";
    }

}
```

otherwise last column would be not parsed

Writer

```
import java.io.*;
import java.util.List;
import java.util.Optional;

public class WeatherReport {

    private List<DailyWeather> dailyReports;

    /**
     * Creates a weather report.
     *
     * @param dailyReports The list of daily reports in the weather report
     */
    public WeatherReport(List<DailyWeather> dailyReports) {
        this.dailyReports = dailyReports;
    }

    /**
     * Writes this weather report to the given writer.
     *
     * @param writer The writer to write to
     */
    public void write(Writer writer) {
        for(int i = 0; i< dailyReports.size(); i++){
            System.out.println(dailyReports.get(i).getClass().toString());
            try {
                writer.append(dailyReports.get(i).getDate().toString());
                writer.append(dailyReports.get(i).getTemperature() + "C");
                writer.append("," + dailyReports.get(i).getWindSpeed() + "BFT");
                switch(dailyReports.get(i).getClass().toString()){
                    case("class SunnyDay"):
                        writer.append("," + ((SunnyDay)
(dailyReports.get(i))).getSunHours() + "h\n");
                        break;
                    case("class RainyDay"):
                        writer.append("," + ((RainyDay)
(dailyReports.get(i))).getMillimetersRained() + "mm\n");
                        break;
                    default:
                        writer.write("\n");(",");//append can take null values whereas
write (legacy) cant.
                }
                writer.close();
            }
            catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
    /**
     * Gets the list of daily reports.
     *
     * @return The list of daily reports in this weather report
     */
    public List<DailyWeather> getDailyReports() {
        return dailyReports;
    }

}
```

```java
import org.junit.jupiter.api.Test;
import java.io.StringWriter;
import java.util.List;
import static org.junit.jupiter.api.Assertions.assertEquals;

public class WeatherTest {

    @Test
    public void testSunnyDay() {
        StringWriter writer = new StringWriter();
        SunnyDay sunny = new SunnyDay(new Date(1, 9, 2020), 20.1, 2, 10);
        WeatherReport report = new WeatherReport(List.of(sunny));

        report.write(writer);

        assertEquals("1/9/2020,SUN,20.1C,2BFT,10h\n", writer.toString());
    }

}
import java.io.*;
import java.util.ArrayList;
import java.util.List;

public class Main {
    public static void main(String[] args){

        List<DailyWeather> days = new ArrayList<DailyWeather>();

            Date yesterday = new Date(9,10,2020);
            Date today = new Date(10,10,2020);
            Date tomorrow = new Date(11,10,2020);

            DailyWeather day0 = new DailyWeather(yesterday, 15.0, 19);
            DailyWeather day1 = new SunnyDay(today, 25.0, 1,10);
            DailyWeather day2 = new RainyDay(tomorrow, 10.0, 30,500);

        days.add(day0);
        days.add(day1);
        days.add(day2);

        WeatherReport report = new WeatherReport(days);

        try {

            //Writer is abstract, so you create a child abstract
            Writer writer = new StringWriter();
            report.write(writer);

            //Both PrintWriter and StringWriter are child of Writer
            Writer printer = new PrintWriter("2_Basic_Writer/out/file.txt");
            ((PrintWriter) printer).print(writer);
            printer.close();

            System.out.println("file created");
            System.out.println(writer);
        }
        catch(FileNotFoundException e) {
            System.out.println("file path not found");
        } catch (IOException e) { //StringWriter also needs to be caught
            e.printStackTrace();
        }
    }
}
```

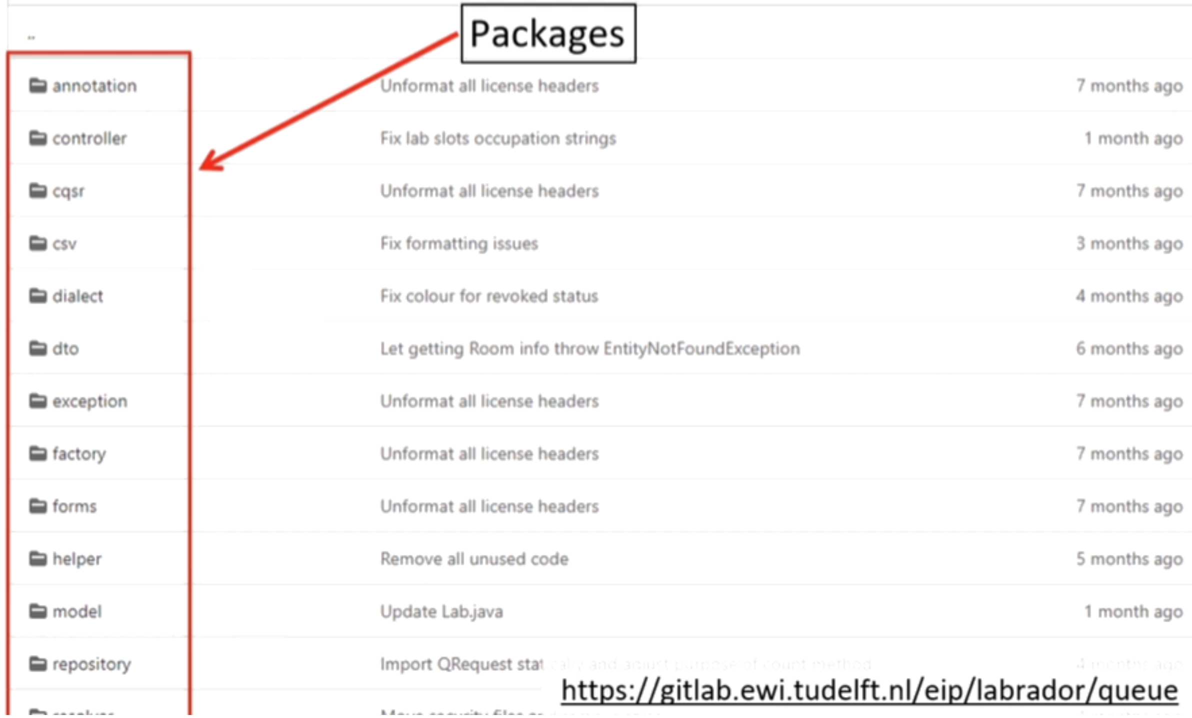
```java
import java.lang.reflect.Array;
import java.util.*;
public class CoffeeMenu {

    private List<CoffeeItem> coffees;
    private List<Addition> additions;

    /**
     * Reads a coffee menu from the given input.
     *
     * @param input The scanner to read from
     * @return The coffee menu read from the input
     */
    public static CoffeeMenu read(Scanner input) {

        List<CoffeeItem> coffees = new ArrayList();
        List<Addition> additions = new ArrayList();

        List<String> coffeeMain = new ArrayList<String>();
        List<String> allergies = new ArrayList<String>();
        List<CoffeeVariation> variations = new ArrayList<CoffeeVariation>();

        while (input.hasNextLine()) {

            String[] array = input.nextLine().split(",");
            System.out.println(array[0]);

            if (array[0].equals("COFFEE") || array[0].equals("ADDITIONS")) {
                if (coffeeMain.size() != 0) {           // After all variations have been checked
                    CoffeeItem coffee = new CoffeeItem( // Construct the CoffeeItem
                            coffeeMain.get(0),          //name (fetched from new List, not the .txt)
                            Double.valueOf(coffeeMain.get(1)),//price (fetched from new List)
                            allergies,
                            variations);
                    coffees.add(coffee);

                    coffeeMain = new ArrayList<String>();        //cleaning up the ArrayLists
                    allergies = new ArrayList<String>();
                    variations = new ArrayList<CoffeeVariation>();

                    if (array[0].equals("ADDITIONS")) {
                        break;
                    }      //last coffee

                }      //necessary to include the last coffee.
                coffeeMain.add(array[1]); //name
                coffeeMain.add(array[2]); //price
                for (int i = 3; i < array.length; i++) {       //remaining elements are allergies
                    allergies.add(array[i]);
                }

            }
            if (array[0].equals("VARIATION")) {
                CoffeeVariation variation = new CoffeeVariation(array[1],
Double.valueOf(array[2]));
                variations.add(variation);
            }
        }

        while (input.hasNextLine()) {
            String[] array = input.nextLine().split(",");
            System.out.println(array[0]);

            if (array[0].equals("ADDITION")){
                String name = array[1];                          //name
                Double price = Double.valueOf(array[2]);         //price
                for (int i = 3; i < array.length; i++) {         //remaining elements are allergies
                    allergies.add(array[i]);
                }
                Addition addition = new Addition(name, price, allergies);
```

```java
                additions.add(addition);
                allergies = new ArrayList<String>();
            }
        }
        CoffeeMenu menu = new CoffeeMenu(coffees,additions);
        return menu;
    }

    /**
     * Creates a coffee menu.
     *
     * @param coffees The list of coffees on the menu
     * @param additions The list of additions on the menu
     */
    public CoffeeMenu(List<CoffeeItem> coffees, List<Addition> additions) {
        this.coffees = coffees;
        this.additions = additions;
    }

    /**
     * Gets the coffees on the menu.
     *
     * @return The list of coffees on this menu
     */
    public List<CoffeeItem> getCoffees() {
        return coffees;
    }

    /**
     * Gets the additions on the menu.
     *
     * @return The list of coffees on this menu
     */
    public List<Addition> getAdditions() {
        return additions;
    }

    /**
     * Checks whether an object is equal to the coffee menu.
     *
     * @param other The other object
     * @return True iff the other is an identical coffee menu
     */
    @Override
    public boolean equals(Object other) {
        if (other instanceof CoffeeMenu) {
            CoffeeMenu that = (CoffeeMenu) other;
            return this.coffees.equals(that.coffees) && this.additions.equals(that.additions);
        }
        return false;
    }

    @Override
    public String toString() {
        return "CoffeeMenu{" +
                "coffees=" + coffees +
                ", additions=" + additions +
                '}';
    }
}
```

- Increase in size = decrease in clarity

Countermeasure #1: Improved Structure

- Distribute code over different methods and classes
- Group classes that belong together in “packages”



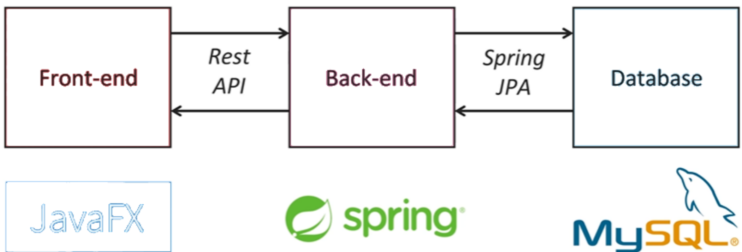
All the code has been divided among several packages.
Each folder represent a packages of functionality that is tight together.

Countermeasure #2: Avoid reinventing the wheel

When creating an application, you likely want to do a lot of things that have been done before…

- In terms of this course:
  - Creating some kind of data structure (e.g. a list).
  - Applying Mathematical concepts.
  - Reading from a file.
  - …

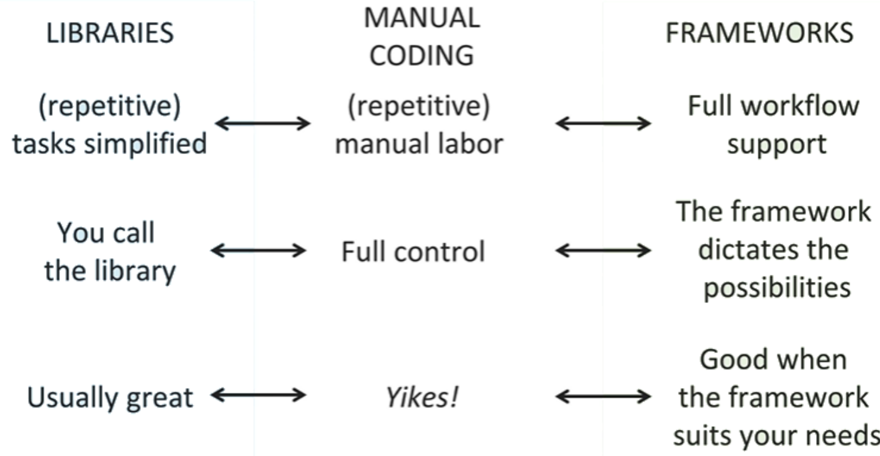
A typical application



We use already available frameworks.
Example Task in real world: Build a wooden door:

(Full control, Lot of labor) **Manual coding**: You start with bare hands, so you create “class” saw.
Then you can chop planks, and then craft the door. Can re-use the saw for other things.
(Partial control, Less labor) **Library**: You can access an already existing “class” saw.
You chose how often to use the saw. Then you chop, and craft the door. Can re-use the saw for other things
(Very limited control, No labor) **Framework**: Just hand the tree to to a factory and they return the crafted door.
Can’t do other thing other than doors.

Often the sweet spot is in the middle, libraries.



Thumbrule: keep manually coding as short as possible, often just for new things that need to be created and cant found elsewhere.

Countermeasure #3: Ensure everyone is on the same page

- Software project managers
  - Manages dependencies
  - Automated project building
  - Automated static analysis checking

- HUGE help when working in a team

Using frameworks and libraries in a project will cause dependencies.

You are subject to version changes.

Project Managers helps us define specific versions of frameworks adn automatically download all the correct versions of dependencies in the project. Great when working with other people.

They can automate project building, and check code style (static analysis checking).

Countermeasure #4: Version control & continuous integration

- Version control (git)
  - Allows for branching and merging paths
  - Enables easy peer reviewing
- Continuous integration
  - Pipeline that runs checks (such as all tests) whenever new code is added to the project.

Use this when you add new features in your program. It's like an online software update platform.

I.e. you put a program (resources?) online, then other people contribute by adding their own branches and after review the changes can be merged into a new working version.

Automated testing of new code.

Conclusion

- Keep code volume down
- Create a clear structure
- Use libraries/frameworks when possible
- Use project managers to
  - Keep track of dependencies
  - Keep setup consistent when there are multiple users
- Use a version control system & continoues integration

WEEK 9 – CLIENT/SERVER

How does your mobile phone app work?

- How does an app, say Twitter communicate with the server? Answer: web API
- It asks for information from a server through a URL → https://api.twitter.com/1.1/followers/list.json
- It gets back JSON (JavaScript Object Notation)

"users": [{ "id": 2960784075, "id_str": "2960784075", "name": "Jesus Rafael Abreu M", "screen_name": "chuomaraver", "location": "", "profile_location": null, …

How twitter app works? It communicates to the server via the web API. This web API asks information from the server through a URL and it typically gets back JSON or XML

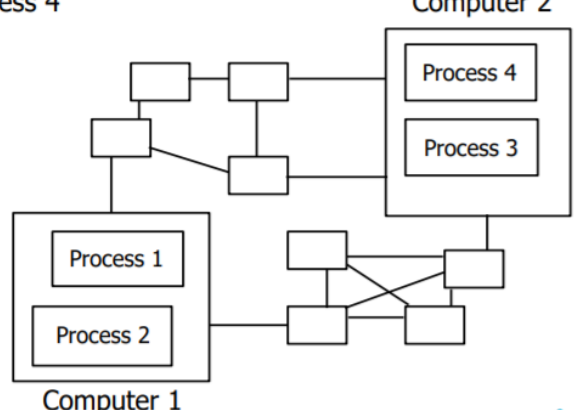
High level way of communicating between a client and a server.
These web APIs rely on sockets technology

Web API

- > 18 000 open Web APIs listed https://www.programmableweb.com/apis/directory
- Google Maps, Twitter, YouTube, Facebook, LinkedIn, …

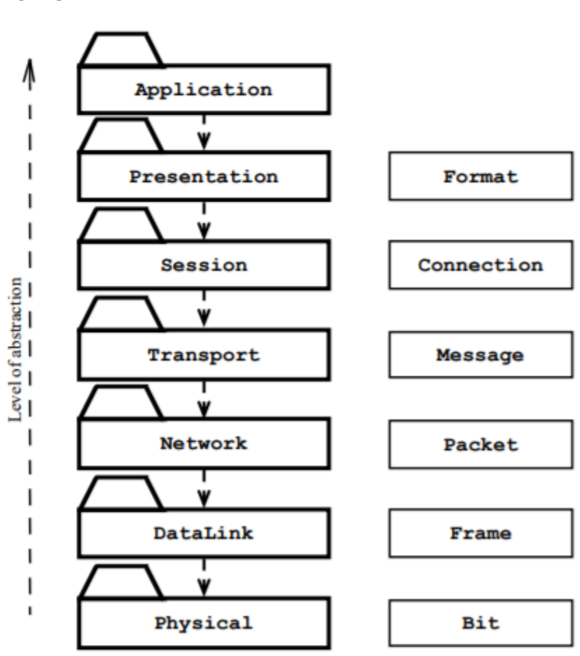
What is the problem?

Send a row of bytes from computer 1 to computer 2, more specifically from process 1 to process 4



How a network works: ISO Reference model for Networks

- ISO's OSI Reference Model
  - ISO = International Standard Organization
  - OSI = Open System Interconnection
- Reference model defines 7 layers of network protocols and strict methods of communication between the layers.



Why 7 layers?

- Easy to change a particular technology (or implementation) without having to change everything
- Example:
  - Change IP4 into IP6 (32 bit IP addresses → 128 bit addresses)
    - Why was this change necessary?
  - Being able to choose between TCP and UDP
- Often used combination: TCP/IP

We are running out of IP addresses

Internet Protocol

- Rules for communication between devices

- Transmission Control Protocol (TCP)
  - Messages between processes
  - Transport layer in the OSI model
- Internet Protocol (IP)
  - Packets between devices
  - Network layer in the OSI model

IP (Internet Protocol) layer

- Initiating device
- Receiving device
- Time to live
  - How long does a packet stay on the network before it may disappear (if the host is unreachable for example)
- Checksum

- The IP (Internet Protocol) work at layer 3 (the so-called network layer) of the ISO/OSI model

TCP (Transmission Control Protocol) layer

- Port of the initiator
- Port of the receiver
  - This identifies the process on your computer (e.g., to make a distinction between a Skype message and a WhatsApp message)
  - Your webserver typically listens to port 80
- Ordering of packages (if a package skipped ahead)
- Flow control (to avoid congesting the receiver)
- Checksum

- In the ISO/OSI model TCP is located at the transport layer (layer 4)

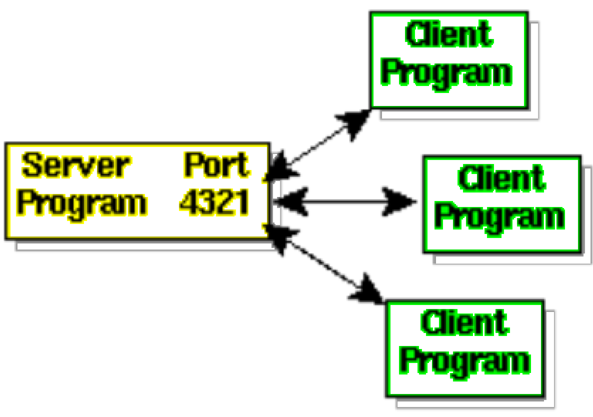
IP-address

- Consists of 4 bytes
  - 127.0.0.1
  - Is true for IPv4, but we are transferring to IPv6 (128 bits)

- A name server translates the machine name (“dutiae.its.tudelft.nl”) into an IP address

- Your own computer typically is “127.0.0.1” or “localhost”
  - So if you want to send messages between processes on your computer, you can use that name/address

Client-server architecture

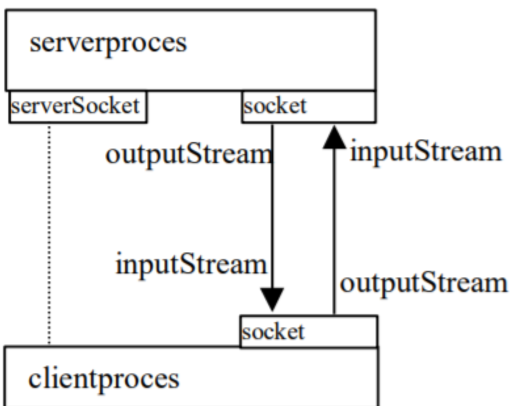


Client server model:

Server listens on a specific port that communicates with one or more client programs.

- How does this happen in Java?
  - Two important classes
    - Socket
    - ServerSocket

Overview of situation in Java



Server process delivers information

Client process provides/requests information to the server.

Sockets are used on both sides

The class ServerSocket

- `public ServerSocket(int port) throws IOException`
  - Creates a server socket on a specified port.
- `public Socket accept() throws IOException`
  - Listens for a connection to be made to this socket and accepts it.
  - The method blocks until a connection is made.
- `public void close() throws IOException`
  - Closes this socket

Responsible for setting up the communication between the client and the server.

It creates a server socket on a specific port

It doesnt need its own IP address because it retrieves that from the computer.

Accept method it listens to a client process comming in with a request. And as soon as it gets it it will return a socket object. Accept is a blocking method, it wont do nothing until there's an external request.

Close method: stop listenning to whater is incomming on the specific port

class Socket

- `public Socket(String host, int port) throws UnknownHostException, IOException`
  - Creates a stream socket and connects it to the specified port number on the named host.
- `public InputStream getInputStream() throws IOException`
- `public OutputStream getOutputStream() throws IOException`
- `public synchronized void close() throws IOException`

Needs to know host name or ip address and port number.

It has inputs to connect the input and output stream and a method to close to free up the port

A client class

```
public class ClientProgram {

    private Socket socket;
    private PrintWriter out;
    private BufferedReader in;

    public static void main(String[] args) {
        ClientProgram program = new
            ClientProgram("localhost", 80);
        program.sendAndReceive("Hello there!");
    }
```

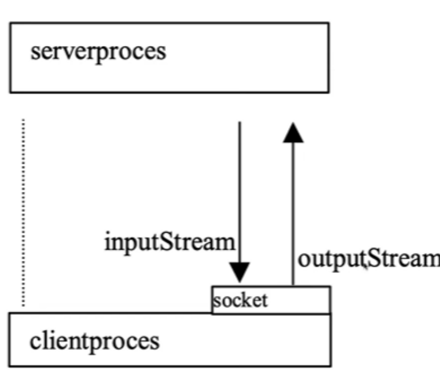
Let's look at a client program where we have some attributes (Sockets, PrintWriter BufferedReader)

In the main we create a new Client Program that will comunicate with Local holst (so everything happens locally on your own computer) and we use port 80 because we know that we are not running a web server right now so this port is free on this compuer.

And we what we send is a simple hello world.

Create Clientsocket

```
public ClientProgram(String hostname, int port) {
    try {
        socket = new Socket(hostname, port);
        out = new
            PrintWriter(socket.getOutputStream());
        in = new BufferedReader(
            new InputStreamReader(
                socket.getInputStream()));
    }
    catch (IOException e) {
        e.printStackTrace();
    }
}
```



In the client program, within a try block, we create a new socket with the hostname and port, we connect a print writter to the output stream, so that I can write to this output stream and the input stream of the socket I actually read from a Buffered reader, an obvious question is why an IO exception has to be thrown? i.e. if the hostnmae doesnt exist or if the port has already been taken, that's why.

Sending/receiving data...

```
public void sendAndReceive(String message)
{
    out.println(message);
    out.flush();

    String returnmessage = new String();
    try {
        returnmessage = in.readLine();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

We need a send and receive method where we actually send the hello message to.

So first we print the message that we just got, we flush it

and then the return message reads from in.readLine()

The server class

```
public class ServerProgram {

    private ServerSocket serversocket;
    private Socket socket;
    private PrintWriter out;
    private BufferedReader in;

    public static void main(String[] args) {
        ServerProgram program = new ServerProgram(80);
        program.listen();
        program.echo();
    }
```

Server Program: It needs the same attributes as the client program.

But the server socket as well.

The server Program Constructor initalisas a Server Socket that listens to specific port

Create a ServerSocket...

```
public ServerProgram(int port) {
    try {
        serversocket = new ServerSocket(port);
    }
    catch(IOException e) {
        e.printStackTrace();
    }
}
```

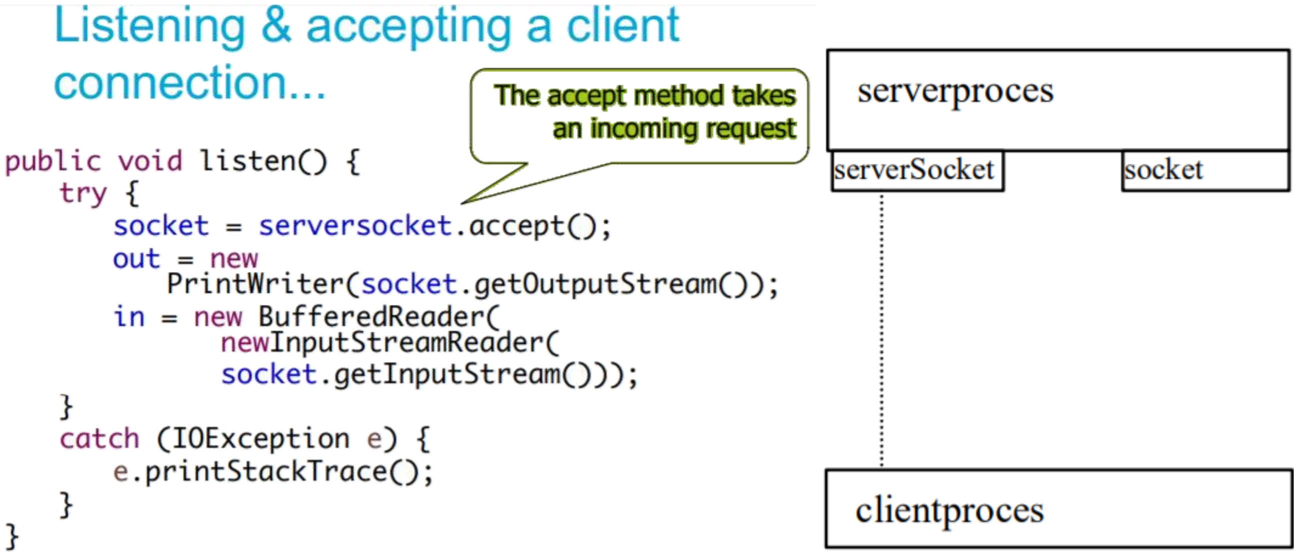
The serversocket listens to incoming requests

When can it go wrong (and an exception is thrown)?
Example: when the port is already "occupied"

serverproces

serverSocket

clientproces

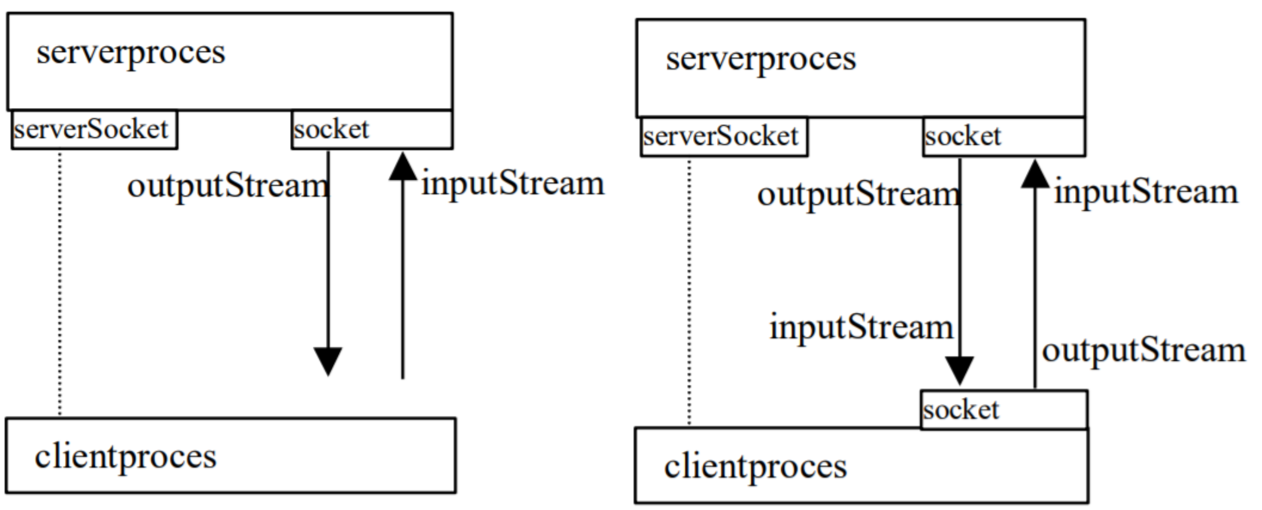


Now that we created the server socket, we need to connect it to the client via the accept method.

The method listen() does that. listen() actively listens to incomming communication at a specific port. As soon as the communication comes in, the accept method will create an instance of a socket.

With that socket variable we can connect the output stream and the input stream and then communication can start.

At this point the server socket is ready to accept incomming requests. Once the incomming request is there, the regular socket is created and we can connect the input and output streams.



Now we have functionally complete model where the client and the server can communicate with each other.

… react to input.

```
public void echo() {
    try {
        System.out.println("Ready to receive...");
        String dataFromNetwork = in.readLine();
        System.out.println("ECHO: " + dataFromNetwork);
        out.println(dataFromNetwork);
        out.flush();
    }
    catch (IOException e) {
        e.printStackTrace();
    }
}
```

The serve still needs to react to the input, we do it in the echo method, which simply puts everything that we received from the network back on the net...

we do it with the in.readLine() (to read the information from the network) we print it on the console, to see what is being transfered, and we are printing it back to the network again.

We put the out.flush() just to ensure that nothing stays in the buffer and the information is sent immideatly.

But…

- What did the `accept()` method of `ServerSocket` do?

```
public Socket accept() throws IOException
```
Listens for a connection to be made to this socket and accepts it. The method blocks until a connection is made.

- This means: your program can only do 1 thing at a time (because it is a blocking method!)

More than 1 client

- Give each new client its own thread that handles all the requests of that client

However, we can use threads and one solution could be that for each new incomming client a new thread is spawned that handles all the requests of that client. I.e. if you build a web server

Create a ServerSocket

```
public void listenSocket(){
    try{
        server = new ServerSocket(4321);
    }
    catch (IOException e)
    {...}
```

For each client a thread

```
while(true){
    ClientWorker w;
    try{
        Socket client = server.accept();
        w = new ClientWorker(client, textArea);
        Thread t = new Thread(w);
        t.start();
    }
    catch (IOException e) {...}
  }
}
```

What we can do is to listen on a socket on a port (4321) and every time an incomming request is accepted a new socket is created and the socket is provided to a CLient Worker and this client worker actually implemments a runnable so that we can actually start a new thread from that runnable.

That would also mean that the while loop finishes and everything starts again, we create a new client worker and we again wait on the Socket line until an incomming request is accepted by our server socket.

The class ClientWorker

```
class ClientWorker implements Runnable {
    private Socket client;
    private TextArea textArea;

    ClientWorker(Socket client, TextArea textArea)
    {
        this.client   = client;
        this.textArea = textArea;
    }
public void run(){
    String line;
    BufferedReader in = null;
    PrintWriter out = null;
    try
    {
        in = new BufferedReader(
                new InputStreamReader(
                    client.getInputStream()));
        out = new PrintWriter(
                client.getOutputStream(), true);
    } catch (IOException e) {..}
```

The class ClientWorker could look like this, it implements Runnable. It contains a socket a text area that has someting to do with a graphical user interface program.

And in the run method, which is actually called when you call the start method, we connect the streams so that communication can happen.

… and process input

```
while(true){
    try{
        //Read from socket
        line = in.readLine();
        //Send data back to client
        System.out.println(line);
        //Append data to text area
        textArea.append(line);
    }catch (IOException e) {...}
  }
}
```

Everything that we read from the network we actually put in the graphical user interface with textArea.append(line);

Take away:

You can use Sockets to communicate between proceses. You can also communicate between computers on the network.

We will still directly encounter the Socket, but many of the communication between 2 processes is nowadays handled through webAPIs and many libraries exist for Java. Socket feature: You can communicate between 2 processes written in different languages

CSE1100 — Object Oriented Programming Assignment 5

October 19, 2020

Preliminaries

Before you start this assignment, create a new project in your OOP workspace. All solution files for this assignment should be placed in this project.

Make sure to also read the lab manual on Weblab or Brightspace (Content → Assignments) for the official guidelines.

Make sure the methods satisfy the pre- and post-condition, if they are specified.

Keep in mind that this week again you have to write your own unit tests. These unit tests should be developed in a similar fashion to the unit tests that were provided in assignments 1 and 2. E.g. aim for 1 assert per test, clear name of the test and description for assertions to explain what the expected behaviour is. **Every method should at least have 1 unit test, but aim for more!**

All methods, fields and the class itself should be annotated with JavaDoc documentation. JUnit tests are not required to have JavaDoc.

Assignment 5.0

In this assignment you will use the following file format:

```
4
FOR SALE:
Emmalaan 23
3051JC Rotterdam
7 rooms
saleprice 300000
boiler
SOLD:
Emmalaan 25
3051JC Rotterdam
5 rooms
saleprice 280000
centralheating
FOR RENT:
Javastraat 88
4078KB Eindhoven
3 rooms
rentprice 500
boiler
RENTED:
Javastraat 90
4078KB Eindhoven
2 rooms
rentprice 400
centralheating
```

Assignment 5.1

This assignment will be based on the classes you defined in the previous assignment.

Modify `House` to be an abstract class and add the fields `boolean available` and `HeatingSystem heatingSystem`. Define the abstract method `String getAvailibilityText()`.

Now also create two concrete classes which extend from `House`: `OwnerOccupiedHouse` and `RentalHouse`. Both classes implement `String getAvailibilityText()` to return the correct header text, i.e., "FOR SALE", "FOR RENT", "SOLD", "RENTED".

Modify the `read` method in `House` to parse the new file format. This method should now create either a `OwnerOccupiedHouse` or `RentalHouse` depending on the information that is retrieved from file.

Create the interface `HeatingSystem` which has 1 method:
`char getEnergyEfficiency(int rooms)`.
Implement two concrete classes which implement `HeatingSystem`: `CentralHeating` and `Boiler`. `CentralHeating` returns an energy efficiency between A–E, where A is for 1–2 rooms, B for 3–4 rooms, etc. `Boiler` is less efficient and has energy efficiency A for 1 room, B for 2 rooms, etc.

You will likely want to add additional (abstract) methods to `House` and its children to prevent code duplication. Excessive code duplication will result in disapproval of the teaching assistant.

Assignment 5.2

Using the modified/new classes defined in Assignment 5.1, write a new `HousingApplication`. This application will be a more comprehensive manager of several houses in the system.

The application must have 4 options:

1. "Add a new OwnerOccupiedHouse". Ask 1 question at a time for the values to be inserted into the fields of the newly created `OwnerOccupiedHouse` (concretely, you should ask for street, housenumber, postalcode, city, number of rooms, price, heating system and of course whether it is for sale or sold).
2. "Add a new RentalHouse". Ask 1 question at a time for the values to be inserted into the fields of the newly created `RentalHouse`. *Think about code duplication, many of the questions are the same as for the OwnerOccupiedHouses, can you reuse this code?*
3. "Show houses". This option asks the user several questions, the answers of which will be used to filter the houses in the `HouseCatalog`. Filtering can be done in several ways; for this assignment you will implement two different forms of filtering. For the first two questions:
    - Should the house be for sale or for rent?
    - Should the house be available right now?

   You need to implement one or more methods that filter input by looping over the elements in the input.
   For the latter two questions:

- What should the maximum price be? (0 means all houses)
- What is the minimum energy efficiency? (E means all houses) This filter should be irrespective of the `HeatingSystem` used in the houses.

You need to use a `Stream` and `filter()` function(s).

After all questions have been answered by the user, filter the list of houses using the methods described above and print out the houses that satisfy all requirements.

4. "Close the application". Close the application, but **write all houses back to the file you initially read from**. The easiest way to do this is to overwrite the existing file (instead of appending).

How can you make such a text-based application? Create a loop in which you ask the user to enter a number that reflects the above options 1 to 4. Then, depending on the user input, you present the user with a "wizard" that asks questions (for options 1–3) or you close the application (option 4 is chosen). Why is the loop important? Because otherwise you could only perform 1 action and then the application quits.
Note: Make sure that newly added houses are immediately available in the "Show houses" option.
Note: Prevent as much code duplication as possible while implementing options 1 and 2.

```java
import java.io.File;
import java.io.FileNotFoundException;
import java.util.ArrayList;
import java.util.Comparator;
import java.util.List;
import java.util.Scanner;
import java.util.function.Predicate;
import java.util.stream.Collectors;

/**
 * The type Housing application.
 */
public class HousingApplication {
//separate program data path with test path, so create a public class "config"
// static public final...

    /**
     * The entry point of application.
     *
     * @param args the input arguments
     */
    public static void main(String[] args) {

        //read catalog from file only once at the start of the program
        HouseCatalog catalog = HouseCatalog.read("src/file.txt");

        int n;
        do {
            System.out.println("Select an option");
            System.out.println("1. Add a new OwnerOccupiedHouse");
            System.out.println("2. Add a new RentalHouse");
            System.out.println("3. Show Houses");
            System.out.println("4. Close the application");
            //System.out.println("5. Show all houses");

            Scanner input = new Scanner(System.in);
            n = input.nextInt();
            switch (n) {
                case (1):
                    addOwnerHouse(catalog);
                    break;
                case (2):
                    addRentalHouse(catalog);
                    break;
                case (3):
                    showHouses(catalog);
                    break;
                case(4):
                    System.out.println("Saving changes...");
                    catalog.write();
                    break;
                case(5):
                    showAll(catalog);
                    break;
                default:
                    System.out.println("Please choose a valid number");
                    break;
            }

        } while (n != 4);

    }
```

```
    /**
     * Scan house house container.
     *
     * @return the house container
     */


    //Add scanner in the main.
    //https://www.w3schools.com/java/java_enums.asp
    //public static HouseContainer scanHouse(Scanner sc, Enum e){
    public static HouseContainer scanHouse(){
        Scanner input = new Scanner(System.in);
        System.out.println("Provide the following (all mandatory) fields, without
spaces:");

        System.out.println("Street:");
        String street = input.next();

        System.out.println("House number:");
        int houseNumber = input.nextInt();

        System.out.println("Postal Code:");
        String postalCode = input.next();

        System.out.println("City:");
        String city = input.next();

        Address address = new Address(city,postalCode,street,houseNumber);

        System.out.println("Number of rooms:");
        int rooms = input.nextInt();

        System.out.println("Price:");
        int price = input.nextInt();

        System.out.println("Available (Y) Not Available (N):");
        boolean available = (input.next().equals("Y")) ? true : false;

        System.out.println("Boiler (B) or Central Heater (H):");
        HeatingSystem heat;
        heat = (input.next().equals("B")) ? new Boiler() : new CentralHeating();

        HouseContainer h = new HouseContainer(address,rooms,price,available,heat);
        return h;
    }

    /**
     * Add owner house.
     */

    public static void addOwnerHouse(HouseCatalog catalog){
        House h = scanHouse();
        House ownHouse = new OwnerOccupiedHouse(
                h.getAddress(),
                h.getnRooms(),
                h.getSalePrice(),
                h.isAvailable(),
                h.heatObject()
                );
        catalog.addHouse(ownHouse);
        System.out.println(ownHouse + " added");
    }
```

```java
/**
 * Add rental house.
 */
public static void addRentalHouse(HouseCatalog catalog){
    House h = scanHouse();
    House rentHouse = new RentalHouse(
            h.getAddress(),
            h.getnRooms(),
            h.getSalePrice(),
            h.isAvailable(),
            h.heatObject()
    );
    catalog.addHouse(rentHouse);
    catalog.addHouse(rentHouse);
    System.out.println(rentHouse + " added");
}

/**
 * Show houses with filters (loop and function based).
 *
 * @param catalog the catalog
 */
public static void showHouses(HouseCatalog catalog){
    Scanner input = new Scanner(System.in);

    //Enum can also be used here
    System.out.println("Rental (R) or Sale (S):");
    String HouseType = (input.next().equals("R")) ? "RENT" : "S"; //Sold/Sale

    System.out.println("Should the house be available (Y/N)?:");
    String available = (input.next().equals("Y")) ? "FOR" : "D"; //solD/renteD

    List<House> houseList = catalog.getHouseList();

    //Loop filtering
    List<House> loop = new ArrayList<House>();

    for(House house : houseList){
        if (house.getAvailabilityText().contains(HouseType) &&
                house.getAvailabilityText().contains(available)){
            loop.add(house);
        }
    }

    System.out.println("Max price:");
    int maxPrice = input.nextInt();
    Predicate<House> atMostPrice = x -> x.getSalePrice() <= maxPrice;

    System.out.println("Max Label:");
    char maxLabel = input.next().charAt(0);
    Predicate<House> atMostLabel = x -> x.getLabel() <= maxLabel;

    Comparator<House> priceOrdered = Comparator.comparingInt(House::getSalePrice);
    //Create custom comparators
    // stream filtering
    //https://winterbe.com/posts/2014/07/31/java8-stream-tutorial-examples/
    List<House> result = loop
            .stream()
            .filter(atMostPrice)
            .filter(atMostLabel)
            .sorted(priceOrdered)
            .collect(Collectors.toList());
```

functions

```
        for (House house : result) {
            System.out.println(house);
        }
    }

    /**
     * Show all Houses.
     *
     * @param catalog the catalog
     */
    public static void showAll(HouseCatalog catalog){
        List<House> houses = catalog.getHouseList();
        for (House house : houses){
            System.out.println(house);
        }
    }
}
```

```java
import java.util.Objects;
import java.util.Scanner;

/**
 * The type House.
 */
public abstract class House { //1. Make House abstract
    private boolean available; //2. add available field
    private HeatingSystem heatingSystem; //3. add heatingSystem field
    private Address address;
    private int nRooms;
    private int salePrice;

    /**
     * Availability of the house
     * @return returns FOR SALE/RENT SOLD/RENTED
     */
    public abstract String getAvailabilityText(); // 4. Define abstract method
getAvailabilityText()

    /**
     * Instantiates a new House.
     *
     * @param address   the address
     * @param nRooms    the n rooms
     * @param salePrice the sale price
     */
    public House(Address address, int nRooms, int salePrice, boolean available,
HeatingSystem heatingSystem) {
        this.address = address;
        this.nRooms = nRooms;
        this.salePrice = salePrice;
        this.available = available;         //2b add fields to constructor
        this.heatingSystem = heatingSystem; //3b add fields to constructor
    }

    /**
     * Is available boolean.
     *
     * @return the boolean
     */
    public boolean isAvailable() {
        return available;
    }

    public HeatingSystem heatObject(){
        return this.heatingSystem;
    }

    /**
     * Get heat type.
     *
     * @return the string
     */
    public String getHeat(){
        return this.heatingSystem.toString();
    }

    /**
     * Get Energy label char.
     *
     * @return the char
     */
```

```
    public char getLabel(){
        return heatingSystem.getEnergyEfficiency(nRooms);
    }

    /**
     * Gets address.
     *
     * @return the address
     */
    public Address getAddress() {
        return address;
    }

    /**
     * Gets rooms.
     *
     * @return the rooms
     */
    public int getnRooms() {
        return nRooms;
    }

    /**
     * Gets sale price.
     *
     * @return the sale price
     */
    public int getSalePrice() {
        return salePrice;
    }

    /**
     * Costs at most boolean.
     *
     * @param price the price
     * @return the boolean
     */
    public boolean costsAtMost(int price){
        return (price-this.salePrice >= 0);
    }

    /**
     * Read house.
     *
     * @param sc the sc
     * @return the house
     */
    public static House read(Scanner sc){
        if (sc == null) {throw new IllegalArgumentException("Wrong house format");}

        //Try to avoid unnecessary delimiter changes.
        //i.e. use a second scanner, or just adjust your Strings later...
        // Scanner scanner2 = new Scanner(sc.nextLine()); // so I don't depend on the
other scanner.

        String availableType = sc.useDelimiter(":").next();       //9 Modify the read
method
        boolean availaiblity = (availableType.contains("FOR")) ? true : false; //9 Modify
the read method
        sc.reset();                                   //9 Modify the read method
        sc.skip(":");                                 //9 Modify the read method
        Address address = Address.read(sc);
        HeatingSystem heat;                           //9 Modify the read method
```

```
        if (sc.hasNext()) {
            int nRooms = sc.nextInt();
            sc.next(); // skip 'rooms'
            sc.next(); // skip 'saleprice'
            int salePrice = sc.nextInt();
            heat = (sc.next().equals("boiler")) ? new Boiler() : new CentralHeating();
//9 Modify the read method

            if (availableType.contains("RENT")) {
                House house = new RentalHouse( //9 Modify the read method
                        address,
                        nRooms, salePrice,
                        availaiblity,
                        heat
                );
                return house;
            } // else: sale houses
                House house = new OwnerOccupiedHouse( //9 Modify the read method
                        address,
                        nRooms, salePrice,
                        availaiblity,
                        heat
                );
            return house;
        }
        throw new IllegalArgumentException("Wrong house format");
    }


    /**
     * Prints a human friendly address, room and price string
     *
     * @return the friendly String
     */
    @Override
    public String toString() {
        return (getAvailabilityText()+ " ($"+ salePrice +"): " + address +
                ": Rooms " + nRooms +
                " (Heating Type: " + heatingSystem) +
                ", Energy Label: " + getLabel() + ")";
    }


    /**
     * Equals method that compares only the address elements and if it is a house object.
     *
     * @returns boolean of equal
     */
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof House)) return false;
        House house = (House) o;
        return getAddress().equals(house.getAddress());
    }
}
```

```
import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintWriter;
import java.util.*;

/**
 * The type House catalog.
 */
public class HouseCatalog {
    private Set<House> houses;

    /**
     * Instantiates a new House catalog.
     */
    public HouseCatalog(){
        houses = new HashSet<>();
    }

    /**
     * Gets the houses.
     *
     * @return the houses
     */
    public List<House> getHouseList() {
        List<House> catalog = new ArrayList<House>();
        for (House house : houses){
            catalog.add(house);
        }
        return catalog;
    }

    /**
     * Add house.
     *
     * @param house the house
     */
    public void addHouse(House house){
        houses.add(house);
    }

    public int getSize(){
        return houses.size();
    }

    /**
     * Houses cost at most list.
     *
     * @param price the price
     * @return the list
     */
    public List<House> housesCostAtMost(int price){
        List<House> catalog = new ArrayList<House>();
        for(House arbitraryName : houses) {
            // for each House in houses (which is the class attribute: private Set<House>
houses)
            if (arbitraryName.getSalePrice() <= price) //condition
                catalog.add(arbitraryName);
            // add the house (which we gave an arbitraryName to the house(s) in the Set.
        }
        return catalog;
    }

    /**
```

```
     * Read house catalog.
     *
     * @param fileName the file name
     * @return the house catalog
     */
    public static HouseCatalog read(String fileName){
        //create static private field path.
        try {
            Scanner sc = new Scanner(new File(fileName));
            if (!sc.hasNext()) throw new IllegalArgumentException("Invalid file");
            HouseCatalog catalog = new HouseCatalog();
            int n = sc.nextInt();
            for (int i = 0; i<n; i++) {
                House house = House.read(sc);
                catalog.addHouse(house);
            }
            sc.close();
            return catalog;
        }
        catch(FileNotFoundException e){
            throw new IllegalArgumentException("Invalid file");
        }
    }

    public void write(){
        try{
            PrintWriter writer = new PrintWriter("src/file.txt");
            String housesString = "";
            int n = 0;

            for (House house : houses){
                String priceType = (house.getAvailabilityText().contains("RENT")) ?
"rentprice " : "saleprice ";
                housesString += house.getAvailabilityText() +":\n"+
                        house.getAddress().getStreet() + " " +
                        house.getAddress().getNumber() + "\n" +
                        house.getAddress().getZipCode() + " " +
                        house.getAddress().getCity() + "\n" +
                        house.getnRooms() + " rooms\n" +
                        priceType + house.getSalePrice() + "\n" +
                        house.getHeat() + "\n";
                n++;
            }
            String str = n + "\n" + housesString;
            writer.print(str);
            writer.close();
            System.out.println("Success");
        }
        catch (FileNotFoundException e){
            System.out.println("Could not write file");
        }
    }

    @Override
    public String toString() {
        String result = "";
        for (House house : houses){
            result += house.toString() + "\n";
        }
        return result;
    }

}
```

```java
public class OwnerOccupiedHouse extends House{ //5. Create Owner Occupied child of House

    private String header; //7b add header field

    /**
     * Instantiates a new Owner occupied house.
     *
     * @param address       the address
     * @param nRooms        the n rooms
     * @param salePrice     the sale price
     * @param available     the available
     * @param heatingSystem the heating system
     */
    public OwnerOccupiedHouse(Address address, int nRooms, int salePrice, boolean
available, HeatingSystem heatingSystem) {
        super(address, nRooms, salePrice, available, heatingSystem);
        this.header = (available) ? "FOR SALE" : "SOLD";
    }

    /**
     * Availability of the house
     * @return returns SOLD/RENTED
     */
    @Override
    public String getAvailabilityText() { //7. Implement getAvailiblityText() method.
        return header;
    }
}

public class RentalHouse extends House{ //6. Create Rental child of House

    private String header; //8b add header field

    /**
     * Instantiates a new Rental house.
     *
     * @param address       the address
     * @param nRooms        the n rooms
     * @param salePrice     the sale price
     * @param available     the available
     * @param heatingSystem the heating system
     */
    public RentalHouse(Address address, int nRooms, int salePrice, boolean available,
HeatingSystem heatingSystem) {
        super(address, nRooms, salePrice, available, heatingSystem);
        this.header = (available) ? "FOR RENT" : "RENTED";
    }

    /**
     * Availability of the house
     * @return returns FOR SALE/RENT
     */
    @Override
    public String getAvailabilityText() { //8. Implement getAvailiblityText() method.
        return header;
    }
}
```

```java
public class HouseContainer extends House{

    public HouseContainer(Address address, int nRooms, int salePrice, boolean available,
HeatingSystem heatingSystem) {
        super(address, nRooms, salePrice, available, heatingSystem);
    }

    @Override
    public String getAvailabilityText() {
        return null;
    }
}
public class CentralHeating implements HeatingSystem { //12. implement subclass

    public CentralHeating() {
    }

    @Override
    public char getEnergyEfficiency(int rooms) { //14. Implement method

        switch(rooms){
            case(1):
            case(2):
                return 'A';
            case(3):
            case(4):
                return 'B';
            case(5):
            case(6):
                return 'C';
            case(7):
            case(8):
                return 'D';
            default:
                return 'E';
        }
    }
    @Override
    public String toString(){
        return "centralheating";
    }
}
public class Boiler implements HeatingSystem {//13. Implement subclass

    /**
     * Calculates the Energy Efficency Label
     * @param rooms - number of rooms
     * @return the char label
     */
    @Override
    public char getEnergyEfficiency(int rooms) { //15. Implement method

        if (rooms>=5)
            return 'E';
        return (char) ('A' + (rooms-1));

    }
    @Override
    public String toString(){
        return "boiler";
    }
}
```

Char letter

```java
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Objects;
import java.util.Scanner;

/**
 * The type Address.
 */
public class Address {
    private String city;
    private String zipCode;
    private String street;
    private int number;

    /**
     * Instantiates a new Address.
     *
     * @param city    the city
     * @param zipCode the zip code
     * @param street  the street
     * @param number  the number
     */
    public Address(String city, String zipCode, String street, int number) {
        this.city = city;
        this.zipCode = zipCode;
        this.street = street;
        this.number = number;
    }

    /**
     * Read address.
     *
     * @param sc the sc
     * @return the address
     */
    public static Address read(Scanner sc){
        if (sc == null) {throw new IllegalArgumentException("Wrong address format");}
        if (sc.hasNext()) {
            String street = sc.next();
            int number = sc.nextInt();
            String zipCode = sc.next();
            String city = sc.next();
            Address address = new Address(city,zipCode,street,number);
            return address;
        }
        throw new IllegalArgumentException("Wrong address format");
    }


    /**
     * Gets city.
     *
     * @return the city
     */
    public String getCity() {
        return city;
    }

    /**
     * Gets zip code.
     *
     * @return the zip code
     */
```

```java
    public String getZipCode() {
        return zipCode;
    }

    /**
     * Gets street.
     *
     * @return the street
     */
    public String getStreet() {
        return street;
    }

    /**
     * Gets number.
     *
     * @return the number
     */
    public int getNumber() {
        return number;
    }


    /**
     * Prints a human friendly address string
     *
     * @return the friendly String
     */
    @Override
    public String toString() {
        return street + " " + number + ", " +
                zipCode + " " + city;
    }


    /**
     * Equals method that compares all address elements
     *
     * @returns boolean of equal
     */
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Address)) return false;
        Address that = (Address) o;
        return this.getNumber() == that.getNumber() &&
                this.getZipCode().equals(that.getZipCode());
    }
}
```

JUNIT – TEMPLATE

```
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;


class TrainTest {

    //same variables as in the class
    private String name;
    private boolean intercity;

    @BeforeEach
    void setUp() {
        //initialise with the case that you will recreate everywhere
        name = "Rotterdam-Utrecht";
        intercity = true;
    }

    @Test
    void Train() {
        //not null check for constructor
        Train train = new Train(name,intercity);
        assertNotNull(train);
    }
    @Test
    void getName() {
        Train train = new Train(name,intercity);
        assertEquals("Rotterdam-Utrecht",name);
    }
    @Test
    void isIntercity() {
        Train train = new Train(name,intercity);
        assertEquals(true,intercity);
    }
    @Test
    void testToString() {
        Train train = new Train(name,intercity);
        assertEquals("train id: 0,train name: Rotterdam-Utrecht intercity:
yes",train.toString());
    }
    @Test
    void equalsSame() {
        //Equals case with same memory address
        Train train0 = new Train(name,intercity);
        Train train1 = train0;
        assertEquals(train0,train1);
    }
    @Test
    void equalsOther() {
        //Equals case with different memory address
        Train train0 = new Train(name,intercity);
        Train train1 = new Train("Rotterdam-Utrecht",true);
        assertEquals(train0,train1);
    }
    @Test
    void notEqualsOther() {
        //Not equals case with different memory address
        Train train0 = new Train(name,intercity);
        Train train1 = new Train("Rotterdam-Amsterdam",false);
        assertNotEquals(train0,train1);
    }
}
```

ABSTRACT CLASS – TEMPLATE

CLASS

```
import ...

public abstract class Station {
    private String name;
    private double xCoordinate;
    private double yCoordinate;

    /**
     * Gets total departures.
     *
     * @return the total departures
     */


    public abstract List<Departure> getDepartures();// mandatory to implement by childern
...
public Station(String name, double xCoordinate, double yCoordinate) {
    this.name = name;
    this.xCoordinate = xCoordinate;
    this.yCoordinate = yCoordinate;
}
```

NO

1*M

CANT CALL ABSTRACT CLASS CONSTRUCTOR TO INSTIATE OBJECT, OBJECT MUST BE INSTANCE OF CHILD

HAS CHILDS THAT EXTEND ONLY SUCH CLASS

```
import ...

public class TrainStation extends Station {
    private Set<TrainDeparture> departures;

                                                    //constructor matching super
    /**
     * Instantiates a new Train station.
     *
     * @param name        the name
     * @param xCoordinate the x coordinate
     * @param yCoordinate the y coordinate
     */
    public TrainStation(String name, double xCoordinate, double yCoordinate) {
        super(name, xCoordinate, yCoordinate);
        departures = new HashSet<TrainDeparture>()
                                          //initialise Set with newHashSet<T>();

    }
                                                           //implement methods
```

INTERFACE – TEMPLATE

INTERFACES

```java
import java.util.Date;

public interface EuropeTravel {

    String getDepartureCountry();
    String getArrivalCountry();
    int getEmissions();

}
```

```java
import java.util.Date;

public interface Departure {

    String departureTime();
    String arrivalTime();

}
```

HAVE SUBCLASS(ES) THAT IMPLEMENT THE INTERFACE(S) (AMONG POSSIBLY OTHERS, INC A PARENT)

```java
import java.util.Date;
import java.util.List;

public class TrainDeparture implements Departure,EuropeTravel{
    private Train train;
    private String departureTime;
    private String arrivalTime;
    private String departureCountry;
    private String arrivalCountry;
    private List<Double> departureCoordinates;

    private List<Double> arrivalCoordinates;

    public TrainDeparture(Train train, String departureTime, String arrivalTime, String
departureCountry, String arrivalCountry, List<Double> departureCoordinates, List<Double>
arrivalCoordinates) {
        this.train = train;
        ...
        this.arrivalCoordinates = arrivalCoordinates;
    }

    //Implement interface 1
    @Override
    public String departureTime() {
        return departureTime;
    }

    @Override
    public String arrivalTime() {
        return arrivalTime;
    }
    //Implement interface 2
    @Override
    public String getDepartureCountry() {
        return departureCountry;
    }

    @Override
    public String getArrivalCountry() {
        return arrivalCountry;
    }

    @Override
    public int getEmissions() {
        double distance = Math.sqrt(
            Math.pow(departureCoordinates.get(0)-arrivalCoordinates.get(0),2)+
            Math.pow(departureCoordinates.get(1)-arrivalCoordinates.get(1),2)
        );
        return (int) Math.round(distance*280);
    }
```

STREAM, FILTER, SORTER, FOR EACH – TEMPLATE

```
System.out.println("Rental (R) or Sale (S):");
String houseType = (input.next().equals("R")) ? "RentalHouse" : "OwnerOccupiedHouse";
...

Predicate<House> ownership = x -> x.getClass().getTypeName().equals(houseType);
Predicate<House> availability = x -> x.isAvailable() == available;
Predicate<House> atMostPrice = x -> x.getSalePrice() <= maxPrice;
Predicate<House> atMostLabel = x -> x.getLabel() <= maxLabel;
Comparator<House> priceOrdered = Comparator.comparingInt(House::getSalePrice);
```

```
List<House> result = houseList
        .stream()
        .filter(ownership)                    //predicate
...
        .sorted(priceOrdered.reversed())      //comparator default ASC, here DESC
        .skip(20)                             //House offset skip/limit + 1 = page number
        .limit(10)                            //Houses displayed
        .distinct()                           //unique Houses
        .sorted(Comparator.comparing(House::getCity,
String.CASE_INSENSITIVE_ORDER).reversed())
        .collect(Collectors.toList());

// for each House "house" in result:
for (House house : result) {
    System.out.println(house);
}
```

PRINTING CITIES (DOUBLE MAP) IN REVERSE ORDER

```
List<String> result = houseList
        .stream()
        .map(x -> x.getAddress().getCity())
        .distinct()
        .sorted()
        .collect(Collectors.toList());

Collections.sort(result, Collections.reverseOrder());

// for each House "house" in result:
for (String city : result) {
    System.out.println(city);
}
```

PRINTING EUR CONVERSION (MAPPED WITH MATH) (VIA ARRAYLIST INSTEAD OF LIST)

```
Object[] result = houseList
            .stream()
            .map(x -> x.getSalePrice()*0.84)
            .sorted()
            .distinct()
            .toArray();

    // for each House "house" in result:
    for (Object price : result) {
        System.out.println(price);
    }
}
```

OPTIONAL – TEMPLATE

```java
import java.util.Optional;

/**
 * The type Train.
 */
public class Train {
    private String name;
    private boolean intercity;
    private Optional<Engineer> engineer;

    public Optional<Engineer> getEngineer(){
        return engineer;
    }

    public void setEngineer(Engineer engineer) {
        this.engineer = Optional.of(engineer);
    }

    public Train(String name, boolean intercity) {
        this.name = name;
        this.intercity = intercity;
        this.engineer = Optional.empty();
    }

    public Train(String name, boolean intercity,Engineer engineer) {
        this.name = name;
        this.intercity = intercity;
        this.engineer = Optional.of(engineer);
    }

    /**
     * Returns string of the object
     * @return
     */
    @Override
    public String toString() {

        return name + " " + ((intercity) ? "Intercity" : "Sprinter") + " Engineer: "
                + engineer.map(Engineer::getName).orElse("Not Found");
    }
}
```

VAR VS TYPE – TEMPLATE

TYPE NOTATION

```java
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Comparator;
import java.util.List;
import java.util.stream.Collectors;

public class Main {

    public static void main(String[] args) {
        Train train0 = new Train("Rotterdam-Utrecht", true);
        TrainStation rdamTrSt = new TrainStation("Rotterdam Centraal", 4.47, 51.92);
        TrainStation utreTrSt = new TrainStation("Utrecht Centraal", 5.11, 52.09);
        Departure departure0 = new TrainDeparture(train0,
                "12:00",
                "13:00",
                "NL",
                "NL",
                rdamTrSt.getCoordinates(),
                utreTrSt.getCoordinates());
        TrainDeparture departure1 = new TrainDeparture(train0,
                "12:30",
                "13:30",
                "NL",
                "NL",
                rdamTrSt.getCoordinates(),
                utreTrSt.getCoordinates());
        rdamTrSt.addDeparture((TrainDeparture) departure0);
        rdamTrSt.addDeparture(departure1);

        Train train1 = new Train("Rotterdam-Amsterdam", false);
        TrainDeparture departure2 = new TrainDeparture(train1,
                "10:30",
                "10:30",
                "NL",
                "NL",
                rdamTrSt.getCoordinates(),
                List.of(4.90, 52.38));
        rdamTrSt.addDeparture(departure2);

        train0.setEngineer(new Engineer("Sergio"));

        List<Departure> departures = rdamTrSt.getDepartures();

        //for each "departure" in departures:
        List<Departure> display = departures.stream()
                .sorted(Comparator.comparing(Departure::departureTime).reversed())
                .collect(Collectors.toList());
        for (Departure departure : display) {
            System.out.println(departure);
        }
    }
}
```

VAR NOTATION

IT IS THE EXACT SAME THING, JAVA ASSUMES THE TYPE FROM THE DECLARATION

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Comparator;
import java.util.List;
import java.util.stream.Collectors;

public class Main {

    public static void main(String[] args) {
        var train0 = new Train("Rotterdam-Utrecht", true);
        var rdamTrSt = new TrainStation("Rotterdam Centraal", 4.47, 51.92);
        var utreTrSt = new TrainStation("Utrecht Centraal", 5.11, 52.09);
        var departure0 = new TrainDeparture(train0,
                "12:00",
                "13:00",
                "NL",
                "NL",
                rdamTrSt.getCoordinates(),
                utreTrSt.getCoordinates());
        var departure1 = new TrainDeparture(train0,
                "12:30",
                "13:30",
                "NL",
                "NL",
                rdamTrSt.getCoordinates(),
                utreTrSt.getCoordinates());
        rdamTrSt.addDeparture((TrainDeparture) departure0);
        rdamTrSt.addDeparture(departure1);

        var train1 = new Train("Rotterdam-Amsterdam", false);
        var departure2 = new TrainDeparture(train1,
                "10:30",
                "10:30",
                "NL",
                "NL",
                rdamTrSt.getCoordinates(),
                List.of(4.90, 52.38));
        rdamTrSt.addDeparture(departure2);

        train0.setEngineer(new Engineer("Sergio"));

        var departures = rdamTrSt.getDepartures();

        //for each "departure" in departures:
        var display = departures.stream()
                .sorted(Comparator.comparing(Departure::departureTime).reversed())
                .collect(Collectors.toList());
        for (Departure departure : display) {
            System.out.println(departure);
        }
    }
}
```

Types are still static! If Java can't figure it out program wont compile. It is still prefered to not use var and declare the types for better clarity.

THREADS – TEMPLATE

RUN THEM

```
public class RunningThreads {

    public static void main(String[] args){
        MyThread j = new MyThread("MyThread");
        MyThread k = new MyThread("MyThread");
        j.start(); //they will run on "parallel"
        k.start(); //the outputs will alternate from one thread to the other randomly
    }
}
```

MAKE THEM

```
public class MyThread extends Thread {

    public MyThread(String name) {
        super(name);
    }

    @Override
    public void run(){
        //THE CODE THAT WILL BE RUN IN PARALLEL
        for(int i=0; i<200; i++){
            System.out.println(i);
        }
    }
}
```

ALTERNATIVE: NOT BE A CHILD OF THREAD AND IMPLEMENT RUNNABLE INTERFACE

```
public class MyRunnableThread extends OtherParent implements Runnable {

    //Runnable field
    private Thread thread;

    //Runnable + Parent constructor
    public MyRunnableThread() {
        super();
        thread = new Thread(this);
        thread.start();
    }

    //Implements run()
    public void run(){
        //THE CODE THAT WILL BE RUN IN PARALLEL
        for(int i=0; i<200; i++){
            System.out.println(i);
        }
    }
}
```

It could also not extend anything and just implement Runnable

RUN THE RUNNABLE ABOVE

```
public class RunningThreads {
    public static void main(String[] args){
        MyRunnableThread runThread = new MyRunnableThread();
        runThread.run();
    }
}
```

j.start(): starts a new thread and calls the run() method (of the variable j's class) that will run in the new thread.
k.run() you executes .run() in the current thread (that was created during the initalisation of k?)

TUTORIAL 8

LAMBDA THREAD WRITER

```java
import java.io.*;
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;

public class WriterApplication {

    /**
     * Asks the user for input and asynchronously writes the user's input to file.
     */
    public static void main(String[] args) {

        List<String> toWrite = new ArrayList<>();
        toWrite.add("The user put in:");

        Scanner scanner = new Scanner(System.in);
        System.out.println("What to write?");
        toWrite.add(scanner.nextLine());
        scanner.close();

        // Anonymous class
//        Thread thread = new Thread(new Runnable() {
//            @Override
//            public void run() {
//                writeToFile(toWrite);
//            }
//        });

        // Lambda
        Thread thread = new Thread(() -> writeToFile(toWrite));

        thread.start();
        System.out.println("Writing...");

    }

    private static void writeToFile(List<String> dataToWrite) {
        File file = new File("output.txt");

        try {

            PrintWriter writer = new PrintWriter(new BufferedWriter(new
FileWriter(file)));
            for (String line : dataToWrite) {
                writer.println(line);
            }
            writer.close();

        } catch (IOException e) {
            System.err.println("Writing failed");
            e.printStackTrace();
        }
    }

}
```

LAMBDA FILTERING BY INSTANCE AND ITS METHODS

```
import ...
public class VenueCatalogue {

    private List<Venue> venues;

    /**
     * Creates a venue catalogue.
     */
    public VenueCatalogue() {
        this.venues = new ArrayList<>();
    }
    /**
     * Gets the list of quality venues.
     *
     * @return The list of venues with 3 stars or more
     */
    public List<Venue> qualityVenues() {
        return venues.stream()
                .filter(v -> v.getStars() >= 3)
                .collect(Collectors.toList());
    }
    /**
     * Gets the list of vegan restaurants names.
     *
     * @return The list of names of restaurants that have vegan menu options
     */
    public List<String> veganRestaurantNames() {
        return venues.stream()
                .filter(v -> v instanceof Restaurant && ((Restaurant) v).hasVeganFood())
                .map(Venue::getName)
                .collect(Collectors.toList());
    }
    /**
     * Gets the list of cheap drinking locations.
     *
     * @return The list of locations of bars where a beer costs less than 2 euros
     */
    public List<String> cheapDrinkingLocations() {
        return venues.stream()
                .filter(v -> v instanceof Bar && ((Bar) v).getCostOfOneBeer() < 2)
                .map(v -> v.getLocation())
                .collect(Collectors.toList());
    }
    /**
     * Adds a venue to the catalogue.
     *
     * @param venue The venue to add
     */
    public void addVenue(Venue venue) {
        this.venues.add(venue);
    }
    /**
     * Gets the list of venues.
     *
     * @return The list of venues in this catalogue
     */
    public List<Venue> getVenues() {
        return venues;
    }

}
```

Select which statement is **false**.

Select one answer

- [ ] An array is an arrangement of consecutive memory locations.
- [x] An array grows automatically when you add elements to it.

- [ ] Using an array can lead to an ArrayIndexOutOfBoundsException.

- [ ] If you pass an array as a parameter, the memory address of that array is copied rather than the values that are stored in the array.

Which statement about `static` methods and `static` attributes is **false**?

Select one answer

- [ ] A `static` method has no access to non-`static` class attributes
- [x] A Java program will not compile when calling a `static` method through an object.
- [ ] A Java program will not compile when calling a non-`static` method through the class.
- [ ] A non-`static` method has access to `static` class attributes

```java
String a = new String("Train");
String b = new String("Train");
```

Study the code block above. Which of the following asserts test(s) whether the values of two strings `a` and `b` are the same? **Multiple answers are possible.**

Select all that apply

- [ ] `assertTrue(a == b);`
- [x] `assertEquals(a, b);`
- [x] `assertTrue(a.equals(b));`
- [ ] `assertSame(a, b);`

Given the following piece of code:

```
1  @Test
2  public void testConstructor() {
3      Address a1 = new Address("Mekelweg", 4);
4      Address a2 = new Address("Mekelweg", 4);
5      assertEquals(a1, a2);
6  }
```

Pick the answer that best applies to the test case above.

Select one answer

Executing this test gives us confidence that both the constructor and the equals method are implemented correctly.

Executing this test gives us confidence that the constructor is tested well, but not the equals method.

Executing this test gives us confidence that the equals method is tested well, but not the constructor.

None of the above.

Select which statement about inheritance in Java is **false**.

Select one answer

In Java all classes inherit from the `Object` class directly or indirectly. The `Object` class is root of all classes.

Multiple inheritance is not allowed in Java.

A non-abstract child class *must* override an abstract method inherited from its parent by defining a method with the same signature.

Child classes can *override* but cannot *overload* methods defined in their superclass(es).

SPEED CONVERSION

```
package weblab;

class SpeedConversion {

  // m/s to km/h.
  public static double msToKmh(double speed){
      return speed*3600/1000;
  }

}
```

RECTANGLE

```
package weblab;

class Rectangles {

    public static String rectangleBuilder(int width, int height){
        String result = "";
        for (int i = 0; i<height; i++){
            for (int j = 0; j<width;j++){
                result += "*";
            }
            result += "\n";
        }
        return result;
    }

}
```

GRADE CALCULATOR

```
package weblab;

class OOPGradeCalculator {

  public static double midTermBonus(double midTermScore) {
    if (midTermScore<6)
        return 0;
    if (midTermScore<7)
        return 0.1;
    if (midTermScore<8)
        return 0.2;
    if (midTermScore<9)
        return 0.3;
    if (midTermScore<10)
        return 0.4;
    return 0.5;
  }

  public static double finalExamScore(double writtenExamScore, double computerExamScore) {
    return (writtenExamScore >= 5.0 && computerExamScore >= 5.0) ? (writtenExamScore + comput
erExamScore)/2 : Math.min(writtenExamScore,computerExamScore);
  }

  public static double finalGrade(double finalExamScore, double midTermBonus) {
    return (finalExamScore >= 5.75) ? Math.min(midTermBonus + finalExamScore,10) : finalExamS
core;
  }

}
```

EQUALS METHOD

```
package weblab;

class Student {
    private String name;
    private int studentNumber;

    public Student (String name, int studentNumber){
        this.name = name;
        this.studentNumber = studentNumber;
    }

    public boolean equals(Object other){
        if (this == other) return true;
        if (!(other instanceof Student)) return false;
        Student that = (Student) other;
        return this.name.equals(that.name) &&
        this.studentNumber == that.studentNumber;
    }
}
```

Marine Management

A certain marine zoo near Harderwijk wants to start a renovation for their park. Before they start moving animals around they want to get a better overview of all of the different shelters they have and which types of animals the shelters are suitable for. This way they know where they can temporarily relocate certain animals. You will get two TXT files that contain data about shelters and animals. *You may assume the provided files are free of errors.*

Info about the shelters

```
Coastal A.1 470 False 40
Tundra B.1 680 True
Reef C.1 745 True 38
Tundra B.2 1860 False
Coastal A.2 2210 True 280
Coastal A.3 160 True 0
Tundra B.3 1340 True
Reef C.2 390 True 17
Coastal A.4 1185 True 130
```

This example file "`shelters.txt`" is available to you on the S-drive of your computer.

Please note that for the shelters, the first word on each line indicates the **type of the shelter**. Following the type are the characteristics of the different shelters (listed in order of appearance):

Coastal Shelter

- ID (name)
- Volume (in m^3)
- Availability (true / false)
- Land surface area (in m^2)

Constant attributes (not in the input file, the same for all coastal shelters):

- Water type: "*Cool Eutrophic*"
- Climate: "*Temperate*"

Tundra Shelter

- ID (name)
- Volume (in m^3)
- Availability (true / false)

Constant attributes (not in the input file, the same for all tundra shelters):

- Water type: "*Cool Eutrophic*"
- Climate: "*Polar*"

Reef Shelter

- ID (name)
- Volume (in m^3)
- Availability (true / false)
- Number of unique coral types

Constant attributes (not in the input file, the same for all reef shelters):

- Water type: "*Warm Trophic*"
- Climate: "*Tropical*"

Info about the animals

```
Penguin; 300; Tundra, Coastal
Duck; 50; Coastal
Rock Crab; 80; Coastal, Tundra, Reef
Shark; 900; Reef, Coastal
Herring; 40; Tundra, Coastal
Orca; 1400; Tundra
Bottlenose Dolphin; 600; Coastal
```

This example file "`animals.txt`" is available to you on the S-drive of your computer.

The characteristics of the animals are (in order of appearance in the file):

Animal

- Name
- Volume required (in m^3)
- List of possible shelters, in order of preference, from most preferred to least preferred.

Application requirements

The zoo asks you to design an application with a textual interface that can do the following;

- Ask the user for two **filenames** that should be read. The order of reading is irrelevant, as long as you specify clearly to the user what input you expect.
- **Read** the delivered file format ("`shelters.txt`") & ("`animals.txt`") and **store** it in a data structure.
- **Show** several different menu options (see next page for more information)
- **Close** the application.

Secondary requirements

- Consider the usefulness of applying **inheritance** and / or **interfaces**.
- Write **unit tests**. See the grading scheme for more details.
- Ensure it is easy to extend the program in the future (new shelters / properties).
- Make sure you use a nice **programming style**; code indentation, whitespaces, logical identifier names, reasonable method lengths, data types, etc.
- Provide **Javadoc** for all non-test code.
- Provide an **equals** method for every class, except the class that contains the `main()` method.
    - Since the *water type* and *climate* properties are constant for all shelters of the same type, those **should not be checked** in your equals methods!

Issues with reading

In the event that you experience trouble with implementing a reader, you are allowed to create hard-coded lists of Objects with the information from the .txt files. You can use this to implement the rest of the assignment, and make it possible to get points for a working application. However, if you do this you **will receive 0 points** for your reader, regardless of its state. Please refer to the grading sheet on the last page of this exam to view a detailed division of points.

Interface

To enable interaction with the user, a **command line interface** should be provided. Since the interface allows the user to interact with information that you need to read from files, make sure that all information has been read before the menu is shown.
After selecting an option the user should return to the main menu, and the application should continue running until the "stop" option (number 6) is selected.
The interface should look like this:

```
Please make your choice:
   1 – Show all shelters
   2 – Show all animals
   3 – Show all shelters suitable for a specific animal
   4 – Show the optimal shelters for a specific animal
   5 – Show the constant properties per shelter type
   6 – Stop the program
```

Option 1 – Showing all shelters

This option should list all the shelters that are held in the data structure (and all their object specific properties). This information should be shown in a user friendly way.

For instance:

```
Coastal Shelter
ID: A.1 – Volume: 470 m3 – Available: False – Land surface: 40 m2
```

Option 2 – Showing all animals

This option should list all the animals that are held in the data structure (and all their object specific properties). This information should be shown in a user friendly way.

For instance:

```
Penguin – Requires: 300 m3 – Preferred shelter: Tundra, Coastal
```

Option 3 – Show all shelters suitable for a specific animal

In order to see where they can relocate animals, the zoo wants to be able to see all shelters a specific animal can move to. An animal can move to a shelter if it is of a type that the animal is compatible with, and has a volume that is equal to or greater than the minimum volume the animal requires. A suitable shelter will be shown, regardless of its availability. The zoo wants to be able to enter the name of an animal and get this overview.
If the animal entered is not known, or if there are no suitable shelters available, this should be communicated clearly to the user (without crashing the program).

For instance, entering `Penguin` should return:

```
Coastal Shelter
ID: A.1 – Volume: 470 m3 – Available: False – Land surface: 40 m2
Tundra Shelter
ID: B.2 – Volume: 1860 m3 – Available: False
Tundra Shelter
ID: B.1 – Volume: 680 m3 – Available: True
Coastal Shelter
ID: A.2 – Volume: 2210 m3 – Available: True – Land surface: 280 m2
Tundra Shelter
ID: B.3 – Volume: 1340 m3 – Available: True
Coastal Shelter
ID: A.4 – Volume: 1185 m3 – Available: True – Land surface: 130 m2
```

Option 4 – Show optimal shelters for a specific animal

In order to easily find the "optimal" shelter for an animal the zoo wants the following: when the user enters the name of an animal this option should print a filtered and sorted list of shelters. The list should be:

- **Filtered** on shelters that are **large enough** for the animal.
- **Filtered** on shelters that are **available**.
- **Sorted** primarily **descending on the preference of the animal** (shelters of the type that is most preferred by the animal should be listed first; shelters of the type that match the second preference of the animal after that, and so on).
- **Sorted** secondarily **ascending on the volume of the shelter** (the shelter that has the smallest possible size within a shelter type, should be listed first).

If the animal entered is not known, or if there are no suitable shelters available, this should be communicated clearly to the user (without crashing the program).

For instance, entering `Penguin` should return:

```
Tundra Shelter
ID: B.1 - Volume: 680 m3 - Available: True
Tundra Shelter
ID: B.3 - Volume: 1340 m3 - Available: True
Coastal Shelter
ID: A.4 - Volume: 1185 m3 - Available: True - Land surface: 130 m2
Coastal Shelter
ID: A.2 - Volume: 2210 m3 - Available: True - Land surface: 280 m2
```

Option 5 – Show shelter properties

This option should print the *water type* and *climate* properties for a shelter type entered by the user. All properties should be shown in a user friendly way.
If the shelter type entered is not known, this should be communicated clearly to the user (without crashing the program).

For instance, entering `Coastal` should return:

```
Coastal Shelter
Warm Trophic Water - Temperate Climate
```

Option 6 – Quitting the application

This option should terminate the application.

Grade composition

1.5 points **Compilation**
Ensuring that your entire project does not contain any errors.
Any compilation error (red error indicator in Eclipse) results in a final grade of 1.

2.0 *points* **Inheritance**
Proper use of inheritance (*1 point*). Additionally there should be a good division of logic between classes & interfaces (*0.5 points*) as well as the proper use of (non-)access modifiers (*0.5 points*).

0.5 points **`equals()` implementation**
Correct implementation of `equals()` in all classes that are part of your data model.

1.5 point **File reading**
Being able to read the user-specified files, and parsing the information into Objects (*0.75 points per well-working file reader*). *A partially functioning reader may still give some points.*

1 points **Code style**
Ensure you have code that is readable. This includes (among others) clear naming, proper use of whitespaces, length and complexity of methods, Javadoc, etc.

0.5 points **Interface**
Having a well-working (looping) interface (*0.2 points*). Having a functioning option 1, 2 & 5 (*0.1 points each*). *A partially functioning interface may still give some points.*

1. 5 points **Data processing**
Having a well-working implementation of options 3 (*0.5 points*) & 4 (*0.7 points*), and clearly informing the user when no shelters / animal was found (*0.3 points*).

1.5 points **Unit testing**
Having one key-class tested completely (*0.6 points*), having all other classes except main completely tested (*0.6 points*) and having tested the methods to read files properly, without actually using the files (*0.3 points*).

Penalties

-1 point **Using a package**
Using a package other than the ‘default package’.

-0.5 points **Hardcoding the filename(s) of the input file**
Not asking the user to enter file names will lead to a deduction of points.

-0.5 points **Handing in a wrongly named ZIP file**
Double check the filename you use before handing in.

MAIN

```java
import java.io.File;
import java.io.FileNotFoundException;
import java.util.ArrayList;
import java.util.Comparator;
import java.util.List;
import java.util.Scanner;
import java.util.stream.Collectors;

/**
 * The type Main.
 */
public class Main {

    private static List<Shelter> shelters;
    private static List<Animal> animals;

    /**
     * The entry point of application.
     *
     * @param args the input arguments
     */
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        getFiles(input);
        String selection;
        do {
            showMenu();
            selection = input.next();
            switch (selection) {
                case ("1"):
                    showShelters(shelters);
                    break;
                case ("2"):
                    showAnimals(animals);
                    break;
                case ("3"):
                    showMatches(input);
                    break;
                case ("4"):
                    showOptimalMatches(input);
                    break;
                case ("5"):
                    showShelterProperties(input);
                    break;
                case ("6"):
                    //Quit program
                    break;
                case ("7"):
                    Animal.addAnimal(animals, input);
                    break;
                case ("8"):
                    Animal.write(animals, input);
                    break;
                default:
                    System.out.println("Unknown option");
                    break;
            }
        } while (!(selection.equals("6")));
    }
```

```java
/**
 * Gets files.
 *
 * @param input the input
 */
public static void getFiles(Scanner input) {
    System.out.println("shelter file name:");
    String shelterFile = input.next();
    Scanner sc;
    try {
        sc = new Scanner(new File("src/" + shelterFile));
        shelters = Shelter.read(sc);
        sc.close();
        System.out.println(shelters.size() + " shelters loaded");
    } catch (FileNotFoundException e) {
        shelters = new ArrayList<Shelter>();
        System.out.println("Shelter file not found, 0 shelters loaded");
    }

    System.out.println("animal file name:");
    String animalFile = input.next();
    try {
        sc = new Scanner(new File("src/" + animalFile));
        animals = Animal.read(sc);
        System.out.println(animals.size() + " animals loaded");
        sc.close();
    } catch (FileNotFoundException e) {
        animals = new ArrayList<Animal>();
        System.out.println("Animal file not found, 0 animals loaded");
    }
}

/**
 * Show menu.
 */
public static void showMenu() {
    System.out.println("Please make your choice:\n" +
            "1 - Show all shelters\n" +
            "2 - Show all animals\n" +
            "3 - Show all shelters suitable for a specific animal\n" +
            "4 - Show the optimal shelters for a specific animal\n" +
            "5 - Show the constant properties per shelter type\n" +
            "6 - Stop the program");
    System.out.println("Beyond exam scope options:\n" +
            "7 - Add animal\n" +
            "8 - Write animal file");
}

/**
 * Show shelters.
 *
 * @param shelters the shelters
 */
public static void showShelters(List<Shelter> shelters) {
    if (shelters.size() == 0) System.out.println("0 shelters");
    for (Shelter shelter : shelters) {
        System.out.println(shelter);
    }
}
```

```
    /**
     * Show animals.
     *
     * @param animals the animals
     */
    public static void showAnimals(List<Animal> animals) {
        if (animals.size() == 0) System.out.println("0 animals");
        for (Animal animal : animals) {
            System.out.println(animal);
        }
    }

    /**
     * Show matches.
     *
     * @param input the input
     */
    public static void showMatches(Scanner input) {
        System.out.println("Animal:");
        String animalName = input.useDelimiter(",|\n|\r\n").next();
        try {
            List<Shelter> matches = Shelter.sheltersMatch(shelters, animals, animalName);
            for (Shelter match : matches) {
                System.out.println(match);
            }
            if (matches.size() == 0) System.out.println("No shelters available");
        } catch (IllegalArgumentException e) {
            System.out.println("Animal not found");
        }
    }

    /**
     * Show optimal matches.
     *
     * @param input the input
     */
    public static void showOptimalMatches(Scanner input) {
        System.out.println("Animal:");
        String animalNameFilter = input.useDelimiter(",|\n|\r\n").next();
        try {
            List<Shelter> original = Shelter.sheltersMatch(shelters, animals,
animalNameFilter);
            List<Shelter> result = original.stream()
                    .filter(Shelter::isAvailable)
                    .sorted(Comparator.comparingInt(Shelter::getVolume))
                    .sorted(Comparator.comparingInt(shelter ->
shelter.getRank(Animal.getAnimal(animals, animalNameFilter))))
                    .collect(Collectors.toList());
            for (Shelter filtered : result) {
                System.out.println(filtered);
            }
            if (result.size() == 0) System.out.println("No shelters available");
        } catch (IllegalArgumentException e) {
            System.out.println("Animal not found");
        }
    }
```

```
    /**
     * Show shelter properties.
     *
     * @param input the input
     */
    public static void showShelterProperties(Scanner input) {
        System.out.println("Shelter type:");
        String type = input.useDelimiter(",|\n|\r\n").next();
        switch (type) {
            case ("Reef"):
                System.out.println(type + ": Water: " + (ReefShelter.water) + ", Climate:
" + ReefShelter.climate);
                break;
            case ("Tundra"):
                System.out.println(type + ": Water: " + (TundraShelter.water) + ",
Climate: " + TundraShelter.climate);
                break;
            case ("Coastal"):
                System.out.println(type + ": Water: " + (CoastalShelter.water) + ",
Climate: " + CoastalShelter.climate);
                break;
            default:
                System.out.println("Shelter type unknown");
                break;
        }
    }
}
```

ANIMAL

```java
import java.io.FileNotFoundException;
import java.io.PrintWriter;
import java.util.*;

/**
 * The type Animal.
 */
public class Animal {
    private String name;
    private int volume;
    private List<String> shelterPreference;

    /**
     * Instantiates a new Animal.
     *
     * @param name              the name
     * @param volume            the volume
     * @param shelterPreference the shelter preference
     */
    public Animal(String name, int volume, List<String> shelterPreference) {
        this.name = name;
        this.volume = volume;
        this.shelterPreference = shelterPreference;
    }

    /**
     * Gets name.
     *
     * @return the name
     */
    public String getName() {
        return name;
    }

    /**
     * Gets volume.
     *
     * @return the volume
     */
    public int getVolume() {
        return volume;
    }

    /**
     * Gets shelter preference.
     *
     * @return the shelter preference
     */
    public List<String> getShelterPreference() {
        return shelterPreference;
    }
```

```
    /**
     * Human readable toString
     *
     * @return the string
     */
    @Override
    public String toString() {
        return name + ", volume: " + volume + "m3, shelter preference: " +
shelterPreference;
    }
    /**
     * Animal equals
     *
     * @param o the other animal
     * @return the boolean
     */
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Animal)) return false;
        Animal animal = (Animal) o;
        return getVolume() == animal.getVolume() &&
                Objects.equals(getName(), animal.getName()) &&
                Objects.equals(getShelterPreference(), animal.getShelterPreference());
    }
    /**
     * Read list.
     *
     * @param sc the sc
     * @return the list
     */
    public static List<Animal> read(Scanner sc) {
        List<Animal> result = new ArrayList<Animal>();
        int l = 1;
        sc.useDelimiter("; |\r\n|\n");
        while (sc.hasNextLine()) {
            String name = (sc.hasNext()) ? sc.next() : "-1";
            int volume = (sc.hasNextInt()) ? sc.nextInt() : -1;
            List<String> shelterPreference = Arrays.asList(sc.next().split(", "));
            if (name.equals("-1") || volume == -1) {
                System.out.println("Animal at line " + l + " is not recognized: " +
                        name + ", " + volume + shelterPreference);
                break;
            }
            result.add(new Animal(name, volume, shelterPreference));
            l++;
        }
        return result;
    }
    /**
     * Get animal animal.
     *
     * @param animals    the animals
     * @param animalName the animal name
     * @return the animal
     */
    public static Animal getAnimal(List<Animal> animals, String animalName) {
        for (Animal animal : animals) {
            if (animal.getName().equals(animalName))
                return animal;
        }
        throw new IllegalArgumentException("Animal not found");
    }
```

//BEYOND SCOPE OF THE EXAM: ADD, WRITE

```
    /**
     * Add animal to list.
     *
     * @param animals the animal list
     * @param input   the animal name
     * @return the list with the new animal
     */

    public static List<Animal> addAnimal(List<Animal> animals, Scanner input) {
        input.useDelimiter(",|\n|\r\n");

        System.out.println("Name:");
        String name = input.next();

        System.out.println("Volume:");
        int volume = (input.hasNextInt()) ? input.nextInt() : -1;
        if (volume == -1) {
            System.out.println("Only type integers");
            return animals;
        }

        System.out.println("Semicolon separated descending shelter preference:");
        String str = input.next();
        List<String> shelterPreference = Arrays.asList(str.split("\\s*;\\s*"));
        Animal animal = new Animal(name, volume, shelterPreference);
        animals.add(animal);
        System.out.println(animal + " has been added");
        return animals;
    }

    /**
     * Write to animal list to file
     *
     * @param animals the animal list
     * @param input   the file name
     */
    public static void write(List<Animal> animals, Scanner input) {
        System.out.println("File name:");
        try {
            String fileName = input.next();
            PrintWriter writer = new PrintWriter("src/" + fileName);
            String animalsString = "";
            for (Animal animal : animals) {
                int n = animal.getShelterPreference().toString().length();
                animalsString += animal.getName() + "; " +
                        animal.getVolume() + "; " +
                        animal.getShelterPreference().toString().substring(1,n-1) + "\n";
            }
            // to avoid IndexOutOfBoundsException when there are 0 animals
            animalsString = (animalsString.equals("")) ? " " : animalsString;
            //remove last \n to read file next time without errors.
            writer.print(animalsString.substring(0, animalsString.length() - 1));

            writer.close();
            System.out.println(fileName + " saved");
        } catch (FileNotFoundException e) {
            System.out.println("file could not be saved");
        }
    }
}
```

SHELTER

```
import java.util.ArrayList;
import java.util.List;
import java.util.Objects;
import java.util.Scanner;

/**
 * The type Shelter.
 */
public abstract class Shelter {
    private String type;
    private String id;
    private int volume;
    private boolean available;

    /**
     * Instantiates a new Shelter.
     *
     * @param type      the type
     * @param id        the id
     * @param volume    the volume
     * @param available the available
     */
    public Shelter(String type, String id, int volume, boolean available) {
        this.type = type;
        this.id = id;
        this.volume = volume;
        this.available = available;
    }
    /**
     * Gets type.
     *
     * @return the type
     */
    public String getType() {
        return type;
    }
    /**
     * Gets id.
     *
     * @return the id
     */
    public String getId() {
        return id;
    }
    /**
     * Gets volume.
     *
     * @return the volume
     */
    public int getVolume() {
        return volume;
    }
    /**
     * Is available boolean.
     *
     * @return the boolean
     */
    public boolean isAvailable() {
        return available;
    }
```

```
    /**
     * Readable string
     *
     * @return the string
     */
    @Override
    public String toString() {
        return type + ", " + id + ", volume: " + volume + ", available: " + available;
    }

    /**
     * Shelter equals
     *
     * @param o the shelter
     * @return the boolean
     */
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Shelter)) return false;
        Shelter shelter = (Shelter) o;
        return getVolume() == shelter.getVolume() &&
                isAvailable() == shelter.isAvailable() &&
                Objects.equals(getType(), shelter.getType()) &&
                Objects.equals(getId(), shelter.getId());
    }

    /**
     * Match boolean.
     *
     * @param animal the animal
     * @return the boolean
     */
    public boolean match(Animal animal) {
        for (String shelterType : animal.getShelterPreference())
            if (animal.getVolume() <= this.volume && shelterType.contains(this.type))
return true;
        return false;
    }

    /**
     * Gets rank.
     *
     * @param animal the animal
     * @return the rank
     */
    public int getRank(Animal animal) {
        int i = 0;
        for (String shelterType : animal.getShelterPreference()) {
            if (animal.getVolume() <= this.volume && shelterType.contains(this.type))
return i;
            i++;
        }
        return Integer.MAX_VALUE;
    }
```

```
    /**
     * Read list.
     *
     * @param sc the sc
     * @return the list
     */
    public static List<Shelter> read(Scanner sc) {
        List<Shelter> result = new ArrayList<Shelter>();
        int l = 1;
        while (sc.hasNextLine()) {
            String type = sc.next();
            String id = sc.next();
            int volume = sc.nextInt();
            boolean available = sc.next().equals("True");
            int n = (sc.hasNextInt()) ? sc.nextInt() : -1;
            switch (type) {
                case ("Coastal"):
                    result.add(new CoastalShelter(type, id, volume, available, n));
                    break;
                case ("Tundra"):
                    result.add(new TundraShelter(type, id, volume, available));
                    break;
                case ("Reef"):
                    result.add(new ReefShelter(type, id, volume, available, n));
                    break;
                default:
                    System.out.println("Shelter at line " + l + " was not recognized");
                    break;
            }
            l++;
        }
        return result;
    }

    /**
     * Shelters match list.
     *
     * @param shelters   the shelters
     * @param animals    the animals
     * @param animalName the animal name
     * @return the list
     */
    public static List<Shelter> sheltersMatch(List<Shelter> shelters, List<Animal>
animals, String animalName) {
        List<Shelter> result = new ArrayList<Shelter>();
        for (Shelter shelter : shelters) {
            if (shelter.match(Animal.getAnimal(animals, animalName))) {
                result.add(shelter);
            }
        }
        return result;
    }
}
```

COASTAL SHELTER

```
/**
 * The type Coastal shelter.
 */
public class CoastalShelter extends Shelter {
    /**
     * The constant water.
     */
    public static final String water = "Cool Eutrophic";
    /**
     * The constant climate.
     */
    public static final String climate = "Temperate";

    private int surface;

    /**
     * Instantiates a new Shelter.
     *
     * @param type      the type
     * @param id        the id
     * @param volume    the volume
     * @param available the available
     * @param surface   the surface
     */
    public CoastalShelter(String type, String id, int volume, boolean available, int
surface) {
        super(type, id, volume, available);
        this.surface = surface;
    }

    /**
     * Readable string
     *
     * @return the string
     */
    @Override
    public String toString() {
        return super.toString() + ", land surface: " + surface + "m2";
    }

    /**
     * Coastal equals
     *
     * @param o the shelter
     * @return the boolean
     */
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof CoastalShelter)) return false;
        if (!super.equals(o)) return false;
        CoastalShelter that = (CoastalShelter) o;
        return surface == that.surface;
    }
}
```

REEF SHELTER

```java
/**
 * The type Reef shelter.
 */
public class ReefShelter extends Shelter {
    /**
     * The constant water.
     */
    public static final String water = "Warm Trophic";
    /**
     * The constant climate.
     */
    public static final String climate = "Tropical";

    private int count;

    /**
     * Instantiates a new Shelter.
     *
     * @param type      the type
     * @param id        the id
     * @param volume    the volume
     * @param available the available
     * @param count     the count
     */
    public ReefShelter(String type, String id, int volume, boolean available, int count)
{
        super(type, id, volume, available);
        this.count = count;
    }

    /**
     * Readable string
     *
     * @return the string
     */
    @Override
    public String toString() {
        return super.toString() + ", reef count: " + count;
    }

    /**
     * Reef equals
     *
     * @param o the shelter
     * @return the boolean
     */
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof ReefShelter)) return false;
        if (!super.equals(o)) return false;
        ReefShelter that = (ReefShelter) o;
        return count == that.count;
    }
}
```

TUNDRA SHELTER

```java
/**
 * The type Tundra shelter.
 */
public class TundraShelter extends Shelter {
    /**
     * The constant water.
     */
    public static final String water = "Cool Eutrophic";
    /**
     * The constant climate.
     */
    public static final String climate = "Polar";

    /**
     * Instantiates a new Shelter.
     *
     * @param type      the type
     * @param id        the id
     * @param volume    the volume
     * @param available the available
     */
    public TundraShelter(String type, String id, int volume, boolean available) {
        super(type, id, volume, available);
    }
}
```

ANIMAL TEST

```java
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;

import static org.junit.jupiter.api.Assertions.*;

class AnimalTest {
    private String name;
    private int volume;
    private List<String> shelterPreference;
    private Animal a;
    private Scanner sc;

    @BeforeEach
    void setUp() {
        name = "Seal";
        volume = 300;
        shelterPreference = List.of("Tundra", "Coastal");
        a = new Animal(name, volume, shelterPreference);
    }

    @Test
    void Animal() {
        assertNotNull(a);
    }

    @org.junit.jupiter.api.Test
    void getName() {
        assertEquals("Seal", a.getName());
    }
```

```
    @org.junit.jupiter.api.Test
    void getVolume() {
        assertEquals(300, a.getVolume());
    }

    @org.junit.jupiter.api.Test
    void getShelterPreference() {
        assertEquals(List.of("Tundra", "Coastal"), a.getShelterPreference());
    }

    @org.junit.jupiter.api.Test
    void testToString() {
        assertEquals("Seal, volume: 300m3, shelter preference: [Tundra, Coastal]",
a.toString());
    }

    @org.junit.jupiter.api.Test
    void sameEquals() {
        Animal b = a;
        assertEquals(a, b);
    }

    @Test
    void diffEquals() {
        Animal b = new Animal("Seal", 300, List.of("Tundra", "Coastal"));
        assertEquals(a, b);
    }

    @Test
    void diffNotEquals() {
        Animal b = new Animal("Whale", 30000, List.of("Tundra", "Coastal"));
        assertNotEquals(a, b);
    }

    @Test
    void read() {
        sc = new Scanner("Penguin; 300; Tundra, Coastal\nDuck; 50; Coastal");
        List<Animal> animals = new ArrayList<Animal>();
        animals.add(new Animal("Penguin", 300, List.of("Tundra", "Coastal")));
        animals.add(new Animal("Duck", 50, List.of("Coastal")));
        assertEquals(animals, Animal.read(sc));
    }

    @Test
    void getAnimal() {
        List<Animal> animals = new ArrayList<Animal>();
        Animal penguin = new Animal("Penguin", 300, List.of("Tundra", "Coastal"));
        animals.add(penguin);
        assertEquals(penguin, Animal.getAnimal(animals, "Penguin"));
    }

    //Beyond the scope of the exam
    @Test
    void addAnimal() {
        List<Animal> animals = new ArrayList<Animal>();
        Animal penguin = new Animal("Penguin", 300, List.of("Tundra", "Coastal"));
        animals.add(penguin);

        List<Animal> animalsFromMethod = Animal.addAnimal(new ArrayList<Animal>(),
                new Scanner("Penguin\n300\nTundra;Coastal"));
        assertEquals(animals, animalsFromMethod);
    }
}
```

SHELTER TEST

```
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;
import static org.junit.jupiter.api.Assertions.*;

class ShelterTest {
    private Shelter c;
    private Shelter t;
    private Shelter r;
    private Scanner sc;

    @BeforeEach
    void setUp() {
        c = new CoastalShelter("Coastal", "A.1", 470, true, 40);
        t = new TundraShelter("Tundra", "B.2", 500, true);
        r = new ReefShelter("Reef", "C.3", 1000, true, 5);
    }

    @Test
    void CoastalShelter() {
        assertNotNull(c);
    }

    @Test
    void TundraShelter() {
        assertNotNull(t);
    }

    @Test
    void ReefShelter() {
        assertNotNull(r);
    }

    @Test
    void getType() {
        assertEquals("Coastal", c.getType());
    }

    @Test
    void getId() {
        assertEquals("A.1", c.getId());
    }

    @Test
    void getVolume() {
        assertEquals(470, c.getVolume());
    }

    @Test
    void isAvailable() {
        assertTrue(c.isAvailable());
    }

    @Test
    void CoastalString() {
        assertEquals("Coastal, A.1, volume: 470, available: true, land surface: 40m2",
c.toString());
    }

    @Test
```

```
    void TundraString() {
        assertEquals("Tundra, B.2, volume: 500, available: true", t.toString());
    }

    @Test
    void ReefString() {
        assertEquals("Reef, C.3, volume: 1000, available: true, reef count: 5",
r.toString());
    }

    @Test
    void notEquals() {
        assertNotEquals(t, c);
    }

    @Test
    void sameEquals() {
        Shelter d = c;
        assertEquals(d, c);
    }

    @Test
    void diffEquals() {
        Shelter d = new CoastalShelter("Coastal", "A.1", 470, true, 40);
        assertEquals(d, c);
    }

    @Test
    void match() {
        Animal penguin = new Animal("Penguin", 300, List.of("Tundra", "Coastal"));
        assertTrue(c.match(penguin));
    }

    @Test
    void read() {
        sc = new Scanner("Coastal A.1 470 False 40\nTundra B.1 680 True");
        List<Shelter> shelters = new ArrayList<Shelter>();
        shelters.add(new CoastalShelter("Coastal", "A.1", 470, false, 40));
        shelters.add(new TundraShelter("Tundra", "B.1", 680, true));
        assertEquals(shelters, Shelter.read(sc));
    }

    @Test
    void getRank() {
        Animal penguin = new Animal("Penguin", 300, List.of("Tundra", "Coastal"));
        assertEquals(1, c.getRank(penguin));
        assertEquals(0, t.getRank(penguin));
        assertEquals(Integer.MAX_VALUE, r.getRank(penguin));
    }

    @Test
    void sheltersMatch() {
        Animal penguin = new Animal("Penguin", 300, List.of("Tundra"));
        List<Animal> animals = new ArrayList<Animal>();
        animals.add(penguin);
        sc = new Scanner("Coastal A.1 470 False 40\nTundra B.1 680 True");
        List<Shelter> allShelters = Shelter.read(sc);
        List<Shelter> oneShelters = new ArrayList<Shelter>();
        oneShelters.add(new TundraShelter("Tundra", "B.1", 680, true));
        assertEquals(oneShelters, Shelter.sheltersMatch(allShelters, animals,
"Penguin"));
    }
}
```

TABLE OF CONTENTS

Topic	page	Topic	page
• Tutorial 1 for loop vs while	5	Debugging	74
• Equals method	8, 13	Text file scanner	75
• JavaDoc example	13	Buffered Reader	78
• Constructors and getters	10	Text file writer	79
• Assignment 1 – Point	16	Week 6 – Tutorial (Inheritance)	80
• Assignment 1 – Circle	19	Week 6 – Tutorial Abstreact Implementation	84
• Arrays	22	Week 7 – Quality, Strings, Generics, Threads	85
• Recursion	22	String scanner, tokens, custom delimiter	87
• ArrayList small example More: JavaDoc	23	Scanner Documentation	88
• JUnit More: JUnitDoc	24	Generics	90
• Override/Overload	24	Threads	91
• Week 3 Tutorial – Array o fill in array (for loop) o array min method o contains o filtering array (26)	25	Parallelism	92
• Week 3 Tutorial – Testing o getters and constructor o equals (28)	27	Thread class	93
• Container Class	29	Interface Runnable	96
• Sets union, intersection, equal	31	Assignment 4 – Scanner Reader and Set	98
• Inheritance o Liskov Principle	31	Foreach with set	103
• Access control rules	32	Week 8 - Function and Bifunction	106
• Parent-child construction	32	Week 8 - Predicate and Supplier	106
• Type checker	33	Week 8 - Streams and Filters	106
• Polymorphism	34	Range, Reader, infinite streams and intermediate operations	107
• Binding (dynamic, static)	34	Terminal operations (for streams)	108
• Forbidding redefinitions (final)	34	Stream Query	108
• Abstract class	35	Optional	109
• Interface, Multiple inheritance	35	Var	109
• Java 8 interface	35	Week 8 - Tutorial 7	110
• Static	36	Multiple Delimiters	113
• Week 4 Tutorial – Container Test	39	Method using Writer class object	114
• Week 4 Tutorial – ArrayList Solution	41	Week 8 - Bigger Picture	118
• Week 4 Tutorial – Array List Test	42	Week 9 – Client/Server	120
• Week 4 Tutorial – Exception thrown Test	43	Assignment 5 – filter, sorted functions	135
• Assignment 2 - Arrays	45	Assignment 5 - number to char	143
• Assignment 2.2: Prime Numbers	48	Template: JUnit Test	146
• Assignment 2.3: StringList	52	Template: Abstract and Interface	147
• Assignment 3.1: Date	56	Template: Stream, filter, sorted, For each	149
• Assignment 3.2: Dateset	59	Template: Optional result	150
• Assignment 3.3: Person	64	Template: Var	151
• Assignment 3.4: DatePicker	67	Template: Threads	153
• Cummulative intersection	68	Exam: Midterm Practice Exam	156
• Exceptions	71	Exam: 2019 – Marine Management: Main class – Menu and Filter/Sorter function	165
• Catching Exceptions	72	Animal class – Reader, Writter, Adder	169
• Passing Exceptions	73	Shelter class and Shelter childs, JUnit Tests	172