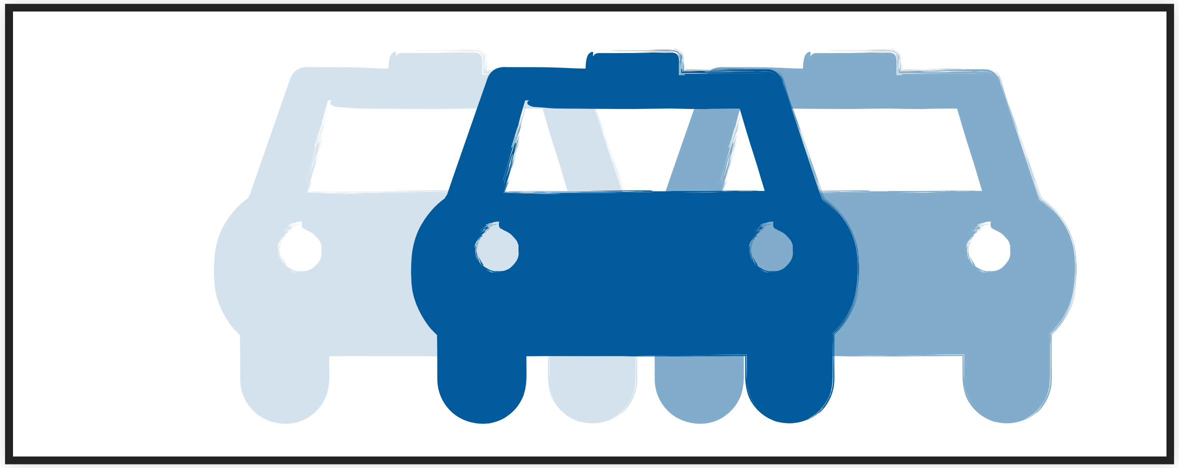
CHAPTER 2: CLASSES AND OBJECTS

LEARNING OBJECTIVES

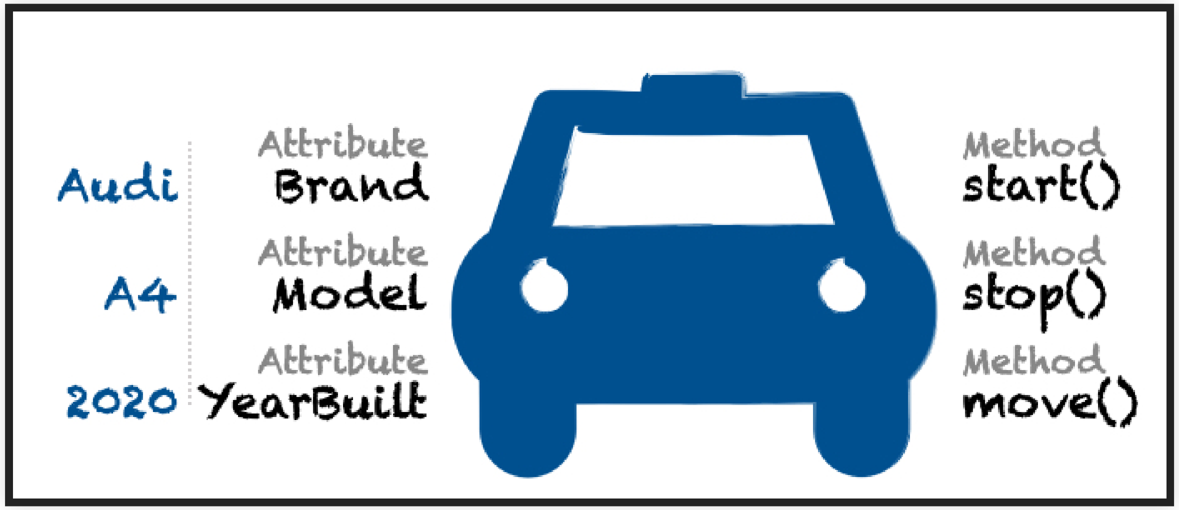
- Explain the object-oriented programming model
- Understand and create basic UML class models
- Define and use Java classes
- Know and use possibilities of object creation
- Know what method overloading means and when it is useful
- Know what references are and how they are used
- Know the "this" reference and how to use it
- Use the Java parameter passing mechanism

- Explain how the garbage collector works in Java
- Use visibility levels
- Use Java arrays
- Use strings in Java programs
- Know how Java objects can be compared
- Use static elements in Java programs
- Define and use getters and setters
- Apply the Java package concept

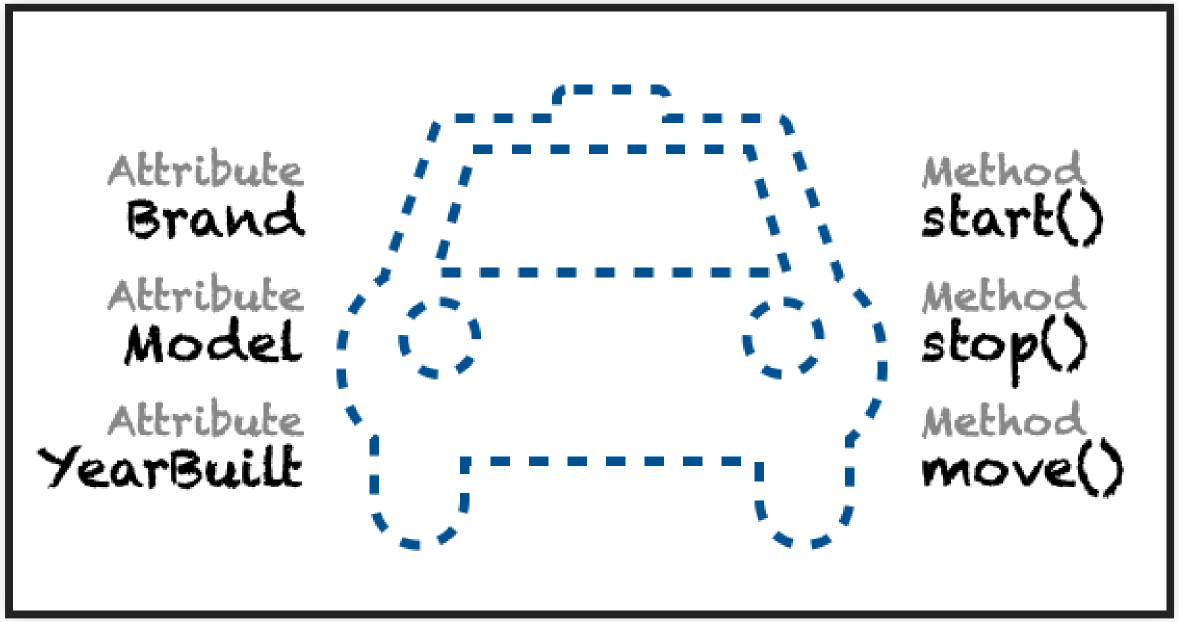
2.1 THE OBJECT ORIENTED PROGRAMMING MODEL



*A program consists of **objects** which together provide the functionality of the program.*



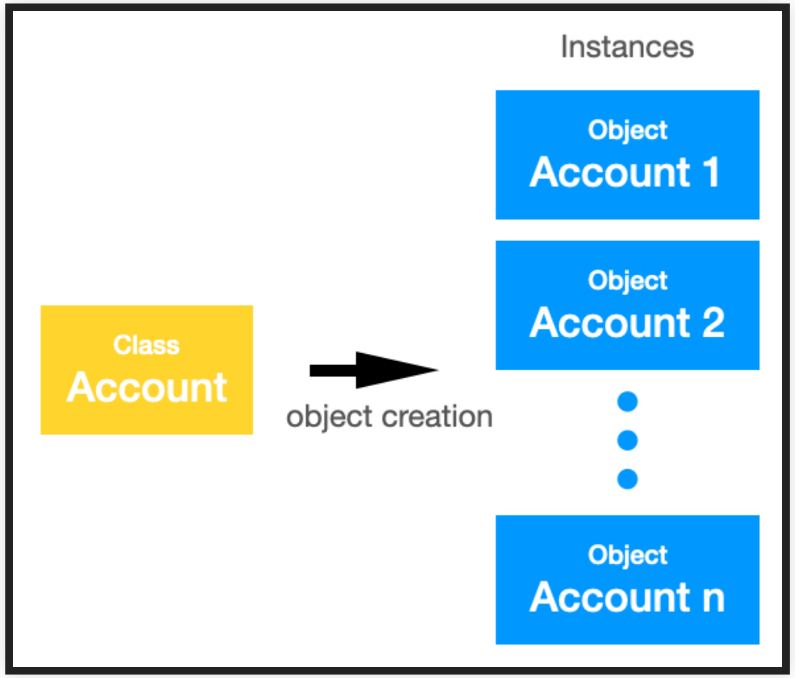
*An **object** consists of **attributes** (variables with current values) and provides services, called **methods**, to other objects. The methods can be used by calling other objects. Each object has a unique ID, which is required for external use.*



The objects of a class differ in their states (=attribute values).

EXAMPLE: CHECKING ACCOUNTS

The checking accounts are realised by account objects, the account class is the blueprint for each account object - i.e the data type for creating the account objects.



What information is in a checking account?

- an account number
- a PIN / secret number
- an account balance

What can you do with a checking account?

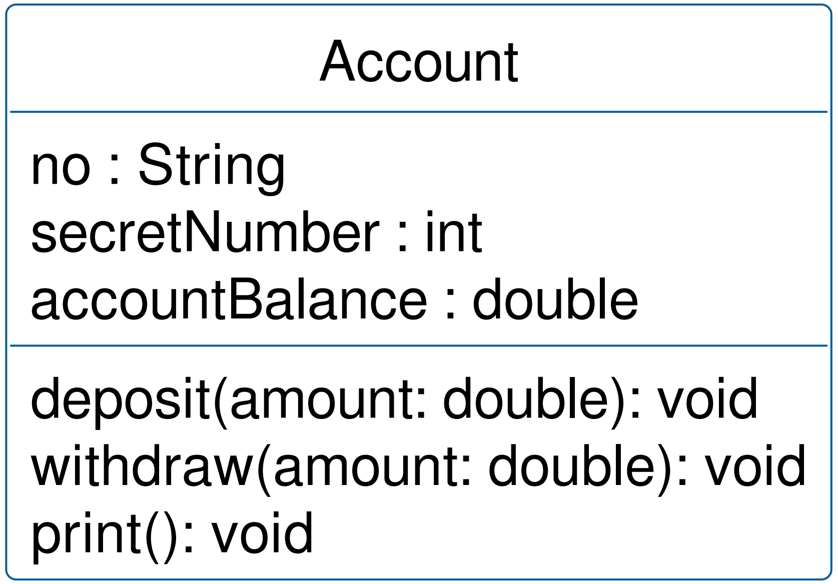
- Deposit money
- Withdraw money
- View account balance

DESCRIBE AN ACCOUNT

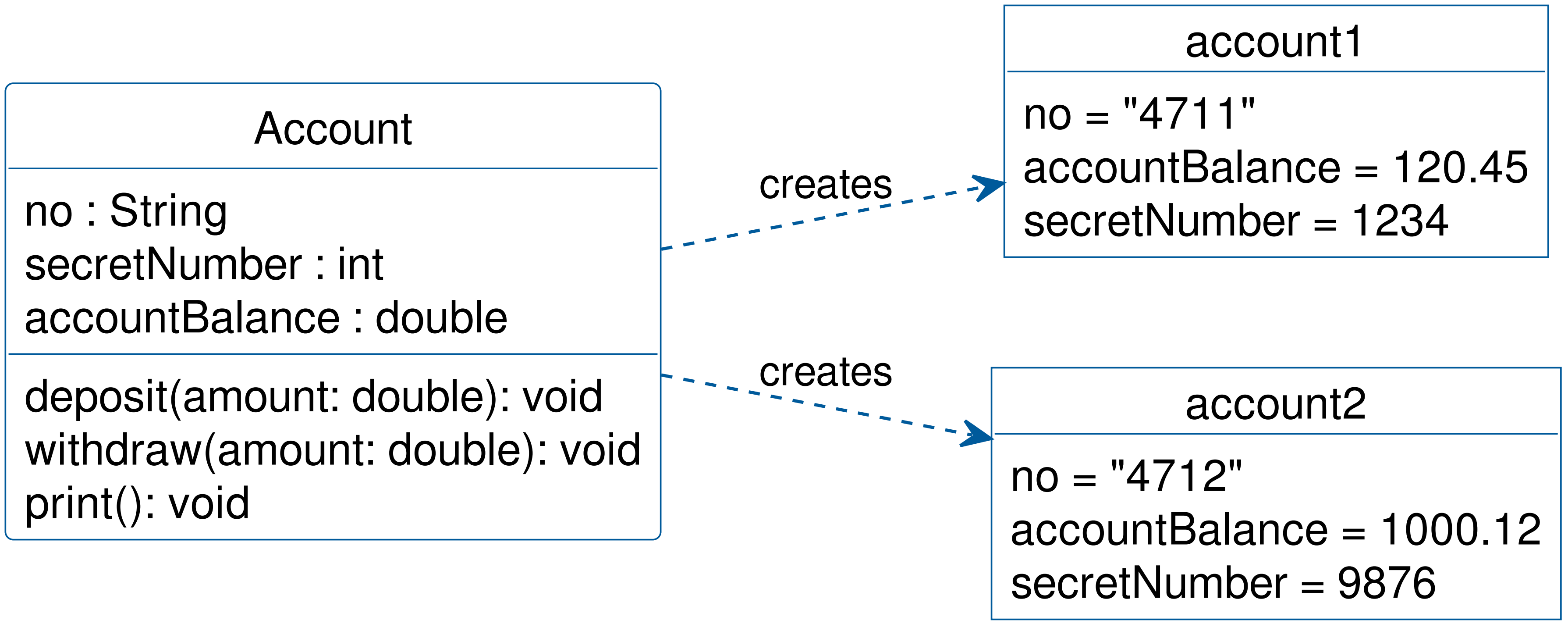
Let's look at two ways of describing an account. One is Java and the other is UML class models.

ACCOUNT IN UML

The following figure shows a class in UML. In this case an account with the three attributes `no`, `secretNumber` and `accountBalance` and the three methods `deposit`, `withdraw` and `print`.



Now, let's create two account objects,
say `account1` and `account2`.



Note: apart from their IDs (not shown) the objects differ in their states.

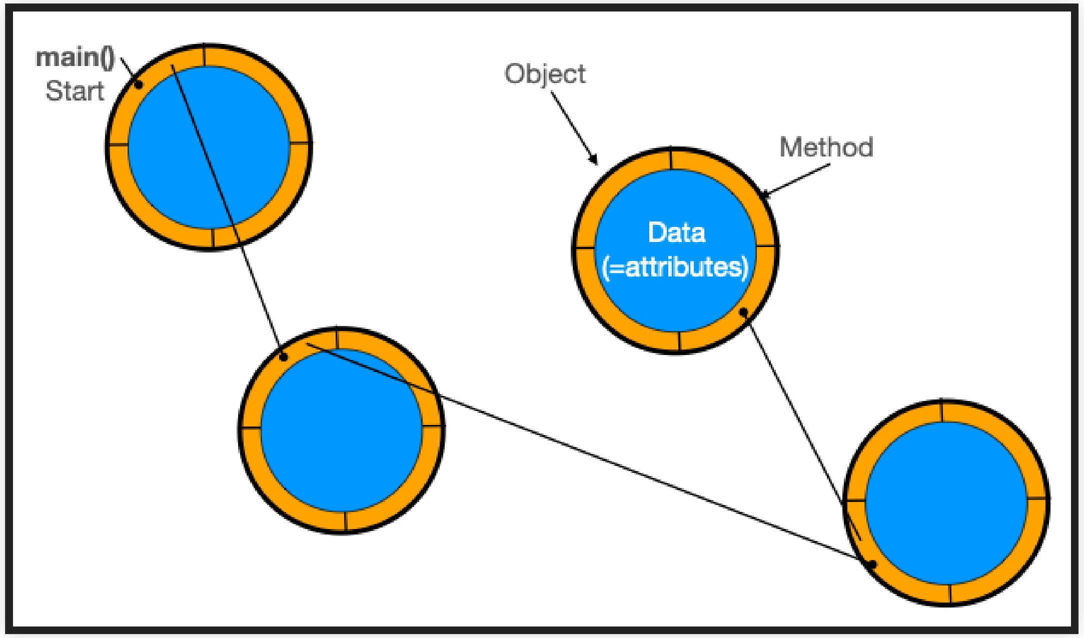
ACCOUNT IN JAVA

If we want to represent this class in Java, we also define the three attributes `no`, `secretNumber` and `accountBalance` as well as the three methods `deposit`, `withdraw` and `print`.

```java
class Account {
  String no;
  int secretNumber;
  double accountBalance;

  void deposit(double amount) {
    accountBalance += amount;
  }
  void withdraw(double amount) {
    accountBalance -= amount;
  }
  void print() {
    System.out.println("Account " + no + ": balance = " + accountBalance);
  }
}
```

Every object-oriented programming program in Java starts with a `main` function, which is called without an associated object (more on that later). In `main()`, objects are typically created and their methods called, which entails further object creations and method calls in order to render the program's functionality.



For example, if we use this account class in a `main` method, this could result in

```
class Account {
    // attribute and method definitions ...
    static public void main(String[] args) {
        // create account with balance
        Account account1 = new Account(100.0);
        // withdraw 50 EUROS
        account1.withdraw(50.0);
    }
}
```

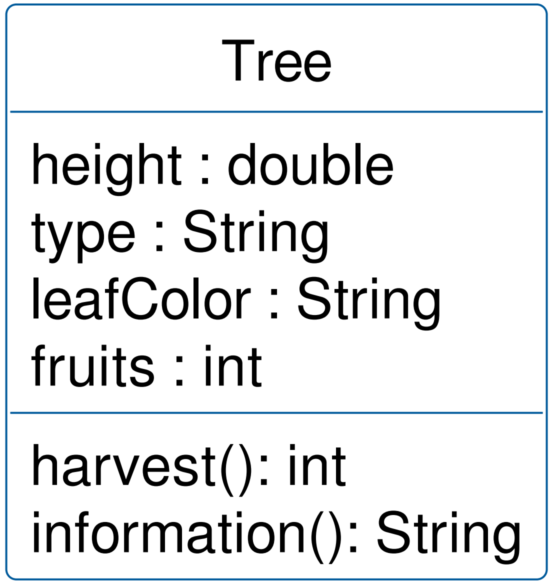
EXERCISE: OO PROGRAMMING MODEL

Which statements are correct?

- ☐ Objects are blueprints
- ☐ Methods are defined within classes
- ☐ Methods are executed with respect to objects
- ☐ Attributes are defined within objects
- ☐ Objekts have unique IDs
- ☐ Methods are defined within objects
- ☐ Attributes are defined within classes
- ☐ Objects in a class typically have different states

EXERCISE: EXPLAIN UML CLASS MODEL

Look at the following UML class model and evaluate the following statements.



- ☐ The class name is Tree
- ☐ The class defines an attribute named leafColor
- ☐ The class defines the method height
- ☐ The attribute does not have a data type
- ☐ The class defines a method named harvest

2.2 CLASS DEFINITION

*A **class** is a user-defined data type that defines attributes and methods that are common to all instances (or objects) of the class. Each class has a unique name in its package (comparable to a folder in the file system). The order of class, method and attribute definitions is arbitrary in Java!*

Attributes *define an objects' variables. All methods can read and change attribute values. As a rule, attributes should not be accessible from outside an object. Access to attributes is to be guaranteed by methods. Within a class, attributes can be used by their names.*

Methods *provide "services" to other objects and determine the behaviour of objects in a class. In Java, a method is called from outside - i.e. by methods of other objects - using the "." operator. Methods can be parameterised in the same way as functions and have a result type (including "void"). Within a class, methods are called by their name.*

```java
class Square {
  double sideLength;

  void setSideLength(double value) {
    sideLength = value;
  }

  double area() {
    return sideLength * sideLength;
  }

  public static void main(String args[]) {
    Square square = new Square();
    square.setSideLength(1.5);
    System.out.println("Area: " + square.area());
  }
}
```

- Class `Square` defining an attribute named `sideLength`
- and a method named `area()`

Objects store data through attributes and local variables

- Objects are created in heap space which means their lifetimes are independent of function call behavior and stack size changes.
- The attributes of an object generally determine the memory requirements of that object. Their lifetime ends when the object is deleted from heap memory by the garbage collector (see later).
- Attributes are automatically initialised by the compiler. Since they are variables of an object (or instance), they are also called `instance variables`. Example: `sideLength`
- Local variables and parameters only exist as long as the block (e.g. method body, loop body or branch) in which they are defined exists. They are created on the stack and are not initialised automatically.

The method main() belongs to the class square as a so-called static method

The main method is a property of the square class, not of the square objects. It is used to start an application. Via the parameter `args`, information can be passed to an application from the command line. Each class can contain at most one main() method, but an application can contain any number of main() methods!

Command Line Example: Starting a class with parameters

We are using a modified version of the square class (see comments):

```
class Square {
    /* same code as in last example */
    public static void main(String args[]) {
        Square square = new Square();
        // use command line parameter as sideLength argument,
        // convert first argument (index 0!) from string to double
        square.setSideLength(Double.valueOf(args[0]));
        System.out.println("Area: " + square.area());
    }
}
```

Command line to start the square class with parameters:

```
> java Square 1.5
> Area: 2.25
```

Variable, attribute and class names should follow a convention.

- Classes begin with capital letters. Name components are separated by capital letters (Camel notation). Example: `AccountManagement` instead of `Account_Management`.
- Local variables, attributes and methods begin with lower case letters. Name parts should be separated by capital letters (Camel notation). Example: `sideLength` instead of `side_length`
- Constants are written with capital letters. Name parts are separated by underscores. Example: `STANDARD_COUNT_NUMBER`

Java programming conventions can be found here, for example:
https://google.github.io/styleguide/javaguide.html

EXERCISE: CLASSES

Which statements are correct?

- ☐ An objects' memory requirement is not related to the attributes of the underlying class
- ☐ Variables and attributes have equal lifetimes
- ☐ Methods have names, parameters and result type
- ☐ Every instance of a class initialises its attributes
- ☐ A class defines which attributes an instance of the class has

EXERCISE: JAVA PROGRAMMING CONVENTIONS

Which names comply with the convention ?

- ☐ Method: enterPin()
- ☐ Method: output_details()
- ☐ Constant: INITIAL_SIZE
- ☐ Attribute: color_code
- ☐ Class name: ACCOUNT
- ☐ Attribute: numberOfDoors
- ☐ Attribute: SecretNumber
- ☐ Class name: Car

EXERCISE: FRACTIONS

Implement a class `Fraction` with two attributes `numerator` and `denominator`. Add the two methods `setNumerator` and `setDenominator`, each of which accepts a parameter of the type `int` and stores them on the corresponding attributes. Add another method `print` in which the attributes `numerator` and `denominator` are output separated by a slash, e.g. `2/4`.

2.3 OBJECT CREATION

Objects are created in Java using the `new` keyword.

```
new Square();
```

Upon execution `new` reserves memory space for the attributes of the object and initialises the attributes. Subsequently, the appropriate constructor method is executed, which was called together with `new`. The Java compiler determines the correct constructor based on the parameter types given.

```java
// create square, save object reference
Square square = new Square();
// set side length
square.setSideLength(1.5);
// compute area using square reference
// save result in variable area
double area = square.area();
// print area
System.out.println(area);
```

After execution `new` returns a reference to the created object, which is usually stored in an attribute or variable in order to be able to call the object later. However, storing the reference is optional.

Also note that the variable `square` contains a reference to the square object (i.e. some number) and **not the object itself**.

SPECIAL CASE: PRIMITIVE DATA TYPES

For efficency reasons regarding performance and memory consumption, instances of primitive data types are not realised as objects.

Let's look at an attribute or a local variable which could be declared using

```
int integerNumber;
```

Execution of the above line causes an automatic reservation of memory space (4 or 8 bytes, depending on the hardware) for an integer at runtime.

For each primitive data type in Java, there is an additional data type of which an instance can be created.

```
Integer i = new Integer(1); // create object containing int value 1
```

This will be covered in more detail in chapters 4 (Wrapper) and 6 (Generics).

CONSTRUCTORS

```java
class Square {
    double sideLength;

    Square(double pSideLength) {
        sideLength = pSideLength;
    }
    // ...
}
```

```java
Square square = new Square(1.5);
```

Constructors are special methods of a class used for object initialisation. A constructor has the same name as the class and can have parameters, but no return value.

CONSTRUCTOR CHAINING

A class may have more than one constructors which can be distinguished by their parameter types.

```java
class Rectangle {
    double height, width;
    // Constructor: initialises the attributes from the parameters
    Rectangle(double value1, double value2) {
        height = value1;
        width = value2;
    }
    // Constructor: uses the 1st CTOR, we get a square
    Rectangle(double value) {
        this(value, value);
    }
    public static void main(String[] args) {
        Rectangle r1 = new Rectangle(3.0, 4.0); // Rectangle: 3.0 x 4.0
        Rectangle r2 = new Rectangle(1.5); // Rectangle: 1.5 x 1.5 (Square)
    }
}
```

The first constructor has two parameters for initialising `height` and `width`. The second constructor will do the same but has only one parameter which will be used to initialise both attributes. To avoid redundant statements the second CTOR calls the first CTOR using the predefined variable "this" (see also section 2.5). The compiler recognises that the first constructor is to be called by matching parameter types.

Calling a CTOR from another is called *constructor chaining*.

There are some limitations with constructor chaining:

- Constructors may only be called by other constructors (or implicitly by `new`).
- The constructor call must be the first statement in a constructor.
- At most one other constructor may be called.

DEFAULT CONSTRUCTOR

If no constructor is defined, the compiler implicitly defines a parameterless default constructor. It reserves memory for the attributes of the object.

```
class Triangle {
    double a, b, c;

    public static void main(String[] args) {
        // The empty constructor is automatically available
        Traingle t1 = new Traingle();
    }
}
```

However, a default constructor may also be created explicitly.

```
class Triangle {
    double a, b, c;

    Triangle() {
        a = 1.0;
        b = 2.0;
        c = 3.0;
    }

    public static void main(String[] args) {
        Traingle t1 = new Traingle();
    }
}
```

EXERCISE: OBJECTCREATION

Have a look at the class Ledger. Extend the `main` method and create an instance. Then call the method `deposit` with the value 100.0 and the method `withdraw` method with the value 50.0. Complete the task by calling `printBalance`.

EXERCISE: CONSTRUCTORCHAINING

Given the class `rectangle` with the attributes `height` and `width`. Add three constructors:

- a default constructor without parameters, which sets `height` and `width` to 1
- a constructor with one parameter that sets `height` and `width` to the given value
- a constructor with two parameters that sets `height` and `width` individually.

Hint: Apply the concept of constructor chaining. There should only be one constructor that actually sets values for the attributes `height` and `width`!_

Now answer the following supplementary questions:

- Explain object creation and constructors (also constructor chaining) in Java using the example.
- Give a suitable constructor that initialises the attributes with suitable parameters
- Add further constructors
- What are the possibilities for object creation?

2.4 OVERLOADING METHODS

In addition to constructors, methods with the same name and different parameter list can also be defined. This is called "method overloading".

```
class Rectangle {
    double height;
    double width;
    // ... Constructor etc.
    void setDimension(double h, double w) {
        height = h;
        width = w;
    }
    void setDimension(double value) {
        setSeiten(l, l);
    }
}
```

- The class `Rectangle` defines a method `setDimension(double, double)`.
- ... and a method `setDimension(double)`.
- Depending on the parameter types used, the compiler decides which method is called.

Selection of the method to be called (1/2)

```
class Rectangle {
    void setDimension(int h, int w) { /*...*/ }
    void setDimension(double h, double w) { /*...*/ }

    public static void main(String[] args) {
        Rectangle r = new Rectangle();
        r.setDimension(1, 1);
        r.setDimension(1.0, 1.0);
        r.setDimension(1.0, 1);
    }
}
```

- The method setDimension is defined with two different signatures: `setDimension(double, double)` and `setDimension(int, int)`.
- The decision which method to call is based on the data types of the supplied parameters

Selection of the method to be called (2/2)

```
class Rectangle {
    void setDimension(int h, int w) { /*...*/ }
    void setDimension(double h, double w) { /*...*/ }

    public static void main(String[] args) {
        Rectangle r = new Rectangle();
        r.setDimension(1, 1);
        r.setDimension(1.0, 1.0);
        r.setDimension(1.0, 1);
    }
}
```

- If there is no exact match, the compiler decides which is the most suitable signature
- Because an `int` can be mapped into a `double`, but a `double` cannot be mapped into an `int` without an explicit cast, this results in...
- ... a call of the method with the signature `setDimension(double, double)`.

EXERCISE: OVERLOADING

Given the class `Filler` with the attribute `level`. Overload the method refill. The new method shall have no parameters. This overloaded method shall call the original method `refill` with the value `1.0`.

2.5 REFERENCES AND „THIS“

A **reference** is a alias name for an object in memory. It is not the memory address! A reference has a specific type (e.g. the corresponding class) and either refers to an object of the designated type or has the predefined value `null`.

```java
Square ref1 = new Square(1.0);
Square ref2;
Square ref3 = new Square(3.0);
```

- The variable `ref1` contains a reference to a square with side length 1
- The variable `ref2` contains no reference
- The variable `ref3` contains a reference to a square with side length 3

***References** can be copied to other variables or attributes.*

```java
// ref2 refers to square with side length 1.0
ref2 = ref1;
// do ref1 and ref2 refer to the same square now ?
if (ref1 == ref2) {
  // Yes!
  System.out.println("equal!");
}
// ref2 now refers to square with side length 3.0
ref2 = ref3;
```

Try out different source code examples on Python Tutor in order to see step-by-step animations of your examples.

Caveat: Python Tutor is using Java 8.

Objects receive an implicit attribute called `this` as a reference to the object itself. For classes, a `this` reference makes no sense!

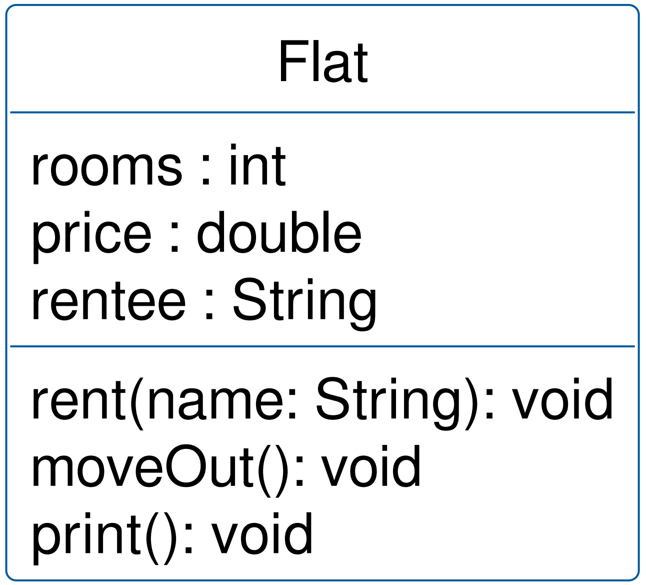
```
class Square {
    double sideLength;
    Square(double sideLength) {
        this.sideLength = sideLength;
    }
    // ...
}
```

- this is the second use of the `this` keyword. It has already been used for constructor chaining.
- note that `this.sideLength` denotes the attribute whereas `sideLength` denotes the parameter!

EXERCISE: FLAT

Use the following UML class model to implement the class `Flat`. Note the following implementation details.

Given the UML class model for a flat with the attributes `rooms`, `price` and `rentee` and the methods `rent`, `moveOut` and `print`.



Implement the class in Java based on the following hints.

- In the `rent` method, assign the name to the `rentee` attribute.
- Empty the `rentee` attribute in the moveOut method by setting it to `""`.
- Finally, the method print is to be implemented. This is to generate an output with System.out.println: **This flat has `<rooms>` rooms, costs `<price>` Eur and is currently rented by `<rentee>`.**
- Add a constructor which first asks for the number of rooms and then the price as parameters and assigns them to the class attributes, rentee should be initialised with `""`.
- Add a Main method and create an object from the class. Then let Richard move in with the `rent` method and call the `print` method.

EXERCISE: THIS REFERENCE

Explain what happens in method `doSomething` of class `Example`:

```
public class Example {
    double x = 5;
    double doSomething(double x) {
        return x + this.x;
    }
}
```

Which statement is correct? doSomething …

- ☐ … returns the sum of both the parameter and the attribute
- ☐ … does not use the parameter
- ☐ … does not use the attribute
- ☐ … returns a result that equals x*2

EXERCISE: JAVA REFERENCES

Which statements are correct?

- ☐ References can be passed to other variables by assignment operators.
- ☐ References can be stored in variables
- ☐ References are used to reference a class.
- ☐ References cannot be passed to other variables, a copy of the object is always passed.
- ☐ References are used to reference an object.

EXERCISE: THIS REFERENCE (2)

Which statements are correct?

☐ There is also a `this` reference within the Main method.

☐ Not every object has the attribute `this`.

☐ `this` is an implicit attribute of an object referencing itself.

☐ The `this` reference can be assigned to a variable.

2.6 PASSING PARAMETERS

Methods can have several parameters. With regard to the data types of the parameters, a distinction is made analogous to attributes:

- Primitive types
- Object references: The respective parameter allows to access an object if the value of the parameter is not `null`.

In Java, parameters are always passed **by value**. The transfer of a memory address (reference with "&" operator) like in "C" does not exist!

Accordingly, with parameters of primitive data types, the variable in the caller's scope cannot be changed by the callee.

However, if an object reference is passed, the object can be used via its reference and may even be changed by calling corresponding methods (e.g. setter methods - see 2.13).

EXAMPLE 1

```
// ... in class Square ...
void adaptLength(Square s) {
    s.sideLength = sideLength;
}

//... in main ...
Square a = new Square(1);
Square b = new Square(3);
a.adaptLength(b);
```

The attribute sideLength of the 2nd square is set to the value of the sideLength attribute of the 1st square.

EXAMPLE 2

```
// ... in class Square ...
void copySquare(Square s) {
    s = this;
}

//... in main ...
Square a = new Square(1);
Square b = new Square(3);
a.copySquare(b);
```

The value of the parameter `p` is changed, it now refers to the 1st square. Outside of the method, i.e. in the caller's scope, this has no effect!

2.7 OBJECT DISSOLUTION

New Java objects can be created using the keyword `new`.

```java
Square a = new Square(1);
Square b = new Square(3);
```

When and how are objects removed from memory?

THE JAVA GARBAGE COLLECTOR

Unlike in "C" or "C++", Java objects cannot be explicitly deleted. Instead, the so-called garbage collector (GC) recognises objects that are no longer referenced and removes them from memory. The GC is part of the Java runtime system and is periodically activated in the background.

Basic operation of the GC

With every object `o` a counter `refs` is managed by the garbage collector. Depending on the event, GC performs the following operations:

- Object Creation
  `o.refs = 1`
- a variable is assigned a reference to `o`
  `o.refs += 1`
- a variable referencing `o` is assigned another reference or `null`
  `o.refs -= 1`
- a variable referencing `o` is dissolved because its scope is not valid anymore
  `o.refs -= 1`
- if `o.refs == 0` holds, `o` can be dissolved and memory is deallocated

Basic operation of the GC: Example

```
{
  Square r = new Square(1.);
  Square s = r;
  r = null;
}
```

- scope is entered
- object is created, reference is stored in "r"; refs == 1
- reference to object is copied; refs == 2
- reference is set to "null"; refs == 1
- scope is left, variables are dissolved; refs == 0 => GC will remove object

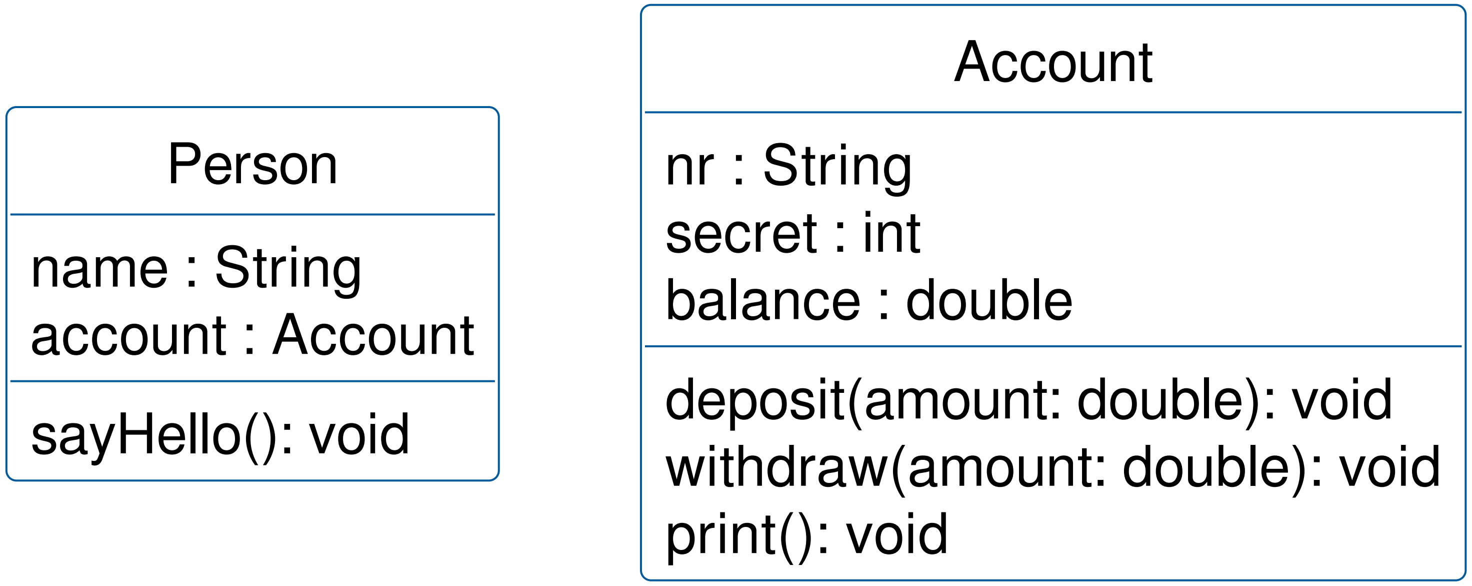
EXERCISE: EXPLAIN GC

Given the following statements of a method.
When may which rectangle object be dissolved?

```
Rectangle r1 = new Rectangle(2, 4), r2, r3;
r3 = new Rectangle(1, 3);
r2 = r3;
r3 = null;
r1 = null;
r2 = null;
```

2.8 ACCESS CONTROL

Let's assume a person object has an associated account.
Which account properties may be accessed from the person objects' methods?



Question: Every Person instance has an attribute of type Account. Which attributes and methods can be accessed from the print() method?

ENCAPSULATION PRINCIPLE IN OBJECT-ORIENTED PROGRAMMING (OOP)

Encapsulation is one of the fundamental concepts in OOP. Encapsulation provides a way to hide the internal state of an object and restrict access to it from outside the class. It is often described using the concept of access modifiers to control the visibility and access to the class members.

To enforce the encapsulation principle, object attributes should never be used directly from the outside, but only via methods. This approach reduces dependencies between classes and thus reduces side effects and testing efforts when changes are made.

Java offers the following access modifiers:

Level	Java keyword	UML	meaning
public	public	+	no restriction
Package	(none)	~	accessible from classes within the same package
Inheritance	protected	#	accessible by objects of the same class or a subclass (see chapter 3) as well as in the same package.
Private	private	-	accessible within an object or by objects of the same class

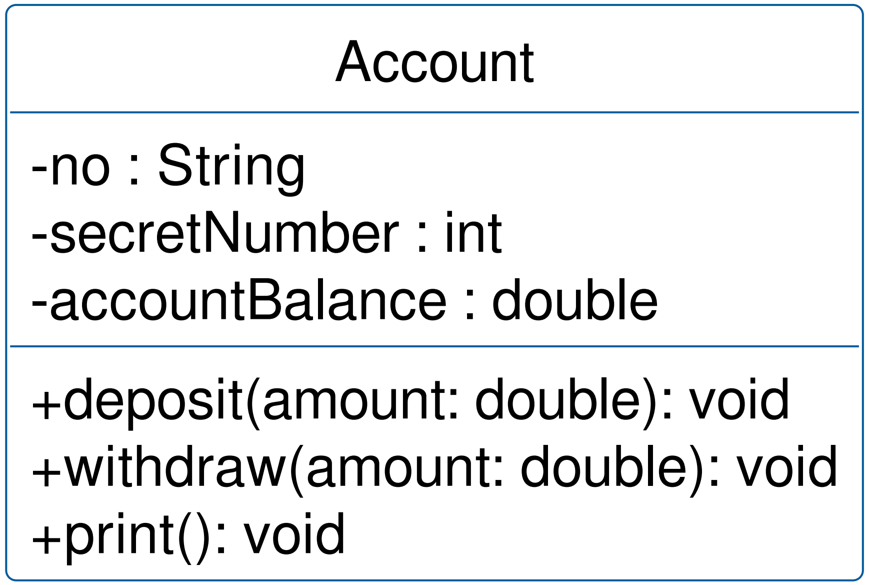
In the case of classes, reducing access is also possible. No marking means that the class can only be used in the same package.

```
class Square {
  // ...
}
```

If you add `public`, the class can be used by all objects.

```
public class Square {
  // ...
}
```

Access Levels can also be specified with UML



Corresponding Java Class Definition:

```java
public class Account {
  private String nr;
  private int secret;
  private double balance;

  public void deposit(double amount) { … }
  public void withdraw(double amount) { … }
  public void print() { … }
}
```

Let's create an object from the class and use attributes and methods:

```java
Account account = new Account();
account.deposit(100.0); // deposit() is public -> OK
account.secret = 1234;  // secret is private -> compiler error!
```

EXERCISE: ACCESS LEVELS

Given the following Java class `Ledger` - add access levels!

Unzip ledger-visibility.zip and import the folder as "Existing Maven Project".

Add access levels so that the attributes `pin` and `balance` are not visible from any other class, the attribute `id` within the inheritance structure, the methods `deposite` and `withdraw` only within the package and `printBalance` from everywhere. The class itself should also be usable from everywhere.

```java
class Ledger {
  String id;
  double balance;
  int pin;

  void deposit(double value) {
    balance += value;
  }
  void withdraw(double value) {
    balance -= value;
  }
  void printBalance() {
    System.out.println("Ledger " + id + ": Balance = " + balance);
  }
}
```

2.9 ARRAYS

An array in Java is a container that contains a **fixed** number of values of the same type.

The concrete data type for the array is specified in the declaration and cannot be changed afterwards.

Arrays are **objects**, i.e. they must be created with `new`.

Array elements are automatically initialised **at runtime** like attributes, i.e. elements of primitive types are assigned their default value and object references are initialised with `null`.

As a consequence, two steps are necessary to create and initialise an array:

1. Create the array object:

```
<type>[] myArray = new <type>[<size>];
... or ...
<type> myArray[] = new <type>[<size>];
```

2. Set each array element:

```
myArray[<index>] = <value>;
```

Example Array

```java
int[] intArray = new int[2];
intArray[0] = 14;
intArray[1] = intArray[0] + 1;
int length = intArray.length;
```

- Creates a new array of type `int` with two elements which are both set to `0`
- Sets the first element at index `0` to 14
- Sets the second element to the value of the first element plus 1
- Returns the length of the array, here `2`

More about arrays

- Valid index values are in the range `0, ...,intArray.length-1`
- The length of an initialised array cannot be changed
- Invalid index values lead to an exception (IndexOutOfBounds) at runtime
- A fast implementation of complex array operations is available in the class Arrays (see Java-API) .

IMPORTANT ARRAY METHODS

- `int[] Arrays.copyOf(int[] original, int newLength)`
  Copies the elements of `original` into a new array with the specified length
- `int[] Arrays.copyOfRange(int[] original, int from, int to)`
  Copies the elements of original starting from `from` to `to` into a new array
- `void Arrays.fill(int[] arr, int val)`
  Sets each field in the array to the value of `val`
- `void Arrays.sort(int[] arr)`
  Sorts the elements in the array in ascending order
- `int Arrays.binarySearch(int[] arr, int key)`
  Searches an element in an array, for this the array must be sorted. Returns the index.

various data types (not only `int`) are possible

WORKING WITH JAVA ARRAYS

```
int[] intArray1;
int[] intArray2 = new int[10];
intArray1 = intArray2;
int intArray3[] = new int[] { 1, 2, 3 };

for (int i = 0; i < intArray1.length; i++)
  intArray1[i] = i;
```

- array reference with value "null"
- Array reference points to array object with 10 int values, all =0
- Both references point to the same array, no copying!
- Static initialization of array elements with 1, 2, 3
- Loop over array size (length attribute) and set values

```java
int sum = 0;
for (int value : intArray1)
  sum += value;

sum = 0;
for (int i = 0; i < intArray1.length; ++i)
  sum += intArray1[i];
```

- calculate sum with new for-each loop
  - Syntax: `for (<element type> <variable> : <array reference>)`
  - Please be aware: all elements of the array are always processed, no changes to the array possible!
- Alternative with standard for loop

ARRAYS WITH DIMENSION > 1

To declare an array with a dimension greater than 1, just use additional brackets `[]`. For example, to declare a 2 x 2 matrix you could code:

```
int[][] intMatrix = new int[2][2];
```

Please

Example 1: 2 x 2 matrix with fixed dimensions

```
double[][] doubleMatrix = new double[2][2];
for (int row = 0; row < doubleMatrix.length; ++row) {
    for (int column = 0; column < doubleMatrix[row].length; ++column) {
        doubleMatrix[row][column] = Math.random();
    }
}
```

- Declare the two-dimensional array, the length of both dimensions is specified
- Loop over the first dimension
- Loop over the second dimension
- Assign a random value to the corresponding field

Example 2: 2 x 2 matrix with dynamic sizing

```
double[][] doubleMatrix = new double[2][];
for (int row = 0; row < doubleMatrix.length; ++row) {
    doubleMatrix[row] = new double[2];
    for (int column = 0; column < doubleMatrix[row].length; ++column) {
        doubleMatrix[row][column] = Math.random();
    }
}
```

- Declaration of the two-dimensional array, the length of the first dimension is specified
- Loop over the first dimension
- Create a new one-dimensional array and assign it to the current field
- Loop over the second dimension
- Assign a random value to the corresponding field

Multidimensional arrays can also be initialized statically when declared; a 2 x 3 matrix of square references can be initialized as follows:

```java
Square[][] squareMatrix = new Square[][] {
  { new Square(1.0), new Square(1.0), new Square(3.0) },
  { new Square(4.0), new Square(5.0), new Square(6.0) }
};
```

How can we iterate over all the elements of an array?
We use the `length` property for every dimension of the array:

```
for (int row = 0; row < squareMatrix.length; ++row)
    for (int column = 0; column < quadratMatrix[row].length; ++column)
        System.out.println(quadratMatrix[row][column].toString());
```

- Size of first dimension: `squareMatrix.length`
- Size of second dimension: `squareMatrix[row].length`
- Read element at [row;column]: `squareMatrix[row][column]`

SORTING ARRAYS

The class `java.util.Arrays` defines a method `sort` for sorting arrays in ascending order.

Sorting an `int`-Array:

```
int[] intArray = new int[] { 31, 22, 13 };
java.util.Arrays.sort(intArray1); // order now: 13, 22, 31
```

Please be aware: To sort arrays with elements of user-defined data types, the data type (e.g. Square) must implement the Comparable interface (see Chapter 6)!_

EXERCISE: ARRAYS (1)

Which statements are correct?

- ☐ The length of an array is specified with the declaration of the variable.
- ☐ The data type of an array can be changed.
- ☐ Access to the index -1 returns the last element
- ☐ Access to the index `array.length` - 1 returns the last element.

EXERCISE: ARRAYS (2)

Which array variables are correctly declared (and initialised, if applicable)?

- ☐ char f = new char[];
- ☐ int[] intFeld2;
- ☐ Square Squares[4];
- ☐ Square[] squares;
- ☐ int intFeld[];
- ☐ char[] f2 = new char[3];

EXERCISE: CHAR-ARRAY

Given an input via the console. Convert the input into lower case letters, iterate over each character of the input and count the letters used. Use an int array with 26 fields for this. Then output all characters that occurred at least once.

Unzip char-array.zip and import the folder as "Existing Maven Project".

EXERCISE: MATRIX

Define and assign a 3x3 int matrix using the given values and output the sum of all matrix cells.

Unzip matrix-array.zip and import the folder as "Existing Maven Project".

2.10 STRINGS

Strings are not primitive data types. Strings are immutable objects of the class String in Java.

Java provides a predefined solution for processing strings - the class `java.lang.String`. The `String` class, like all classes from `java.lang`, is imported by default and can be instantiated as follows:

```java
String s1 = "Hello";
// empty String
String s2 = new String();
// see first statement - the compiler will convert it to this:
String s3 = new String("Hello");
```

Strings cannot be changed after they have been created, they are *Immutable*. I.e. every change causes the creation of a new string object.

```
String msg = "Hello"; // create 1st string
msg += " ";           // create 2nd and 3rd string for " " and result
msg += "World";       // create 4th and fifth string for "World" and result
```

Please be aware: If you want to proceed differently for reasons of efficiency, the StringBuilder class, for example, is a good choice.

Definition: An object is considered `immutable` if the state of the object can no longer be changed after it has been created.

Please be aware: String constants are internally held only once in memory, so "new" is used for s3 in the following example to create a new string object:

```java
String s1 = "Hello";
String s2 = "Hello";
String s3 = new String("Hello");
System.out.println(s1 == s2); // true
System.out.println(s1 == s3); // false
```

IMPORTANT METHODS OF THE STRING CLASS

Given two string references s1 and s2.

- `s1.compareTo(s2)`
  s1 < s2 yields <0, s1 equal s2 yields 0, s1 > s2 yields >0
- `s1.equals(s2)`
  yields true, if s1 and s2 contain the same characters (in the same order)
- `s1.length()`
  yields number of characters in s1
- `s1.indexOf(s2)`
  yields the index of the first character of 1st occorence of s2 in s1 or -1 if not found
- `s1.lastIndexOf(s2)`
  like indexOf() but for the last occurence of s2
- `s1.charAt(int i)`
  yields the character at index `i`
- `s1.trim()`
  yields a string with whitespace characters removed from both ends of s1

- `s1.substring(int i1, int i2)`
  yields the substring with characters from i1 to i2-1
- `String substring(int i1)`
  yields the substring with characters from i1 to the end of s1
- `s1.replace(char x, char y)`
  yields a copy of s1 with all characters x replaced by y
- `s1.replace(String x, String y)`
  yields a copy of s1 with all strings x replaced by y
- `s1.startsWith(String s2)`
  yields true if s1 starts with s2
- `s1.endsWith(String s2)`
  yields true if s1 ends with s2
- `s1.contains(String s2)`
  yields true if s1 contains s2

EXAMPLES

```java
String s1 = "My";
String s2 = "example";
// Concatenation using "+" operator
String s3 = s1 + " " + s2; // "My example"
s3 += " number " + 1; // "My example number 1"
System.out.println(s3);
// using substrings
System.out.print(s1.substring(11, 17)); // "number"
System.out.print(s3.substring(11)); // "number 1"
// search string "ample"
int pos = s3.indexOf("ample"); // pos == 5
// read 4th character
char c = s3.charAt(3); // c == 'e'
```

THE SPECIAL METHOD TOSTRING

In Java, each object has a `toString()` method that can create a string representation of the object. This ensures, for example, that any object can be output to the console. For self-defined classes, the toString() method should be overridden to obtain the required string representation.

Example: String representation for squares

```java
public class Square {
    private double sideLength;

    public String toString() {
        return "Square(side length = " + sideLength + ")";
    }

    // ... definition of area() method left out

    static public void main(String[] args) {
        Square s = new Square(1.5);
        double area = q.area();
        System.out.printf("area = %f\n", area);
        // s.toString() will be called to convert s to string!
        System.out.println(s); // output: "Square(side length = 1.5);
    }
}
```

EXERCISE: STRINGS

Read a URL as input from the console and extract components such as schema, user, password, host, port, path, query and fragment.

Unzip string-example.zip and import the folder as "Existing Maven Project".

URL structure

```
       |------------------ Schema specific part ------------------|

 https://max:muster@www.example.com:8080/index.html?p1=A&p2=B#ressource
 \___/   \_/ \____/ \_____________/ \__/\_________/ \_______/ \_______/
   |      |    |           |         |       |          |         |
Schema+   | Password     Host      Port    Path       Query    Fragment
       User
```

Read in any URL as a string and extract the components. Use string methods as described in this chapter. Then output the extracted components using `System.out.println`.

Optional components are user, password, host, port, path, query and fragment.

Output `Error` if something is wrong with the input.

Use the following URLs to test your solution

- https://max:muster@www.example.com:8080/index.html?p1=A&p2=B#ressource
- https://heise.de
- http://online-lectures-cs.thi.de/gdp2-ss2021-inf/chapter02.html#/1/1
- file:///irgend/ein/pfad.txt
- https://lmgtfy.com?q=String+Java
- https://media.giphy.com/media/LmNwrBhejkK9EFP504/giphy.gif

Hints

- Above all, the methods `split`, `indexOf` and `substring` are needed.
- First extract the schema
- Then check whether there is a user, a password is optional and may only be specified if a user is present
- It may help to work from back to front after schema and user/password are extracted

2.11 CHECKING FOR EQUALITY

String Example Revisited

```
String s1 = "Hello";
String s2 = "Hello";     // same String Object reused
String s3 = new String("Hello"); // force creation of a new string
System.out.println(s1 == s2); // true
System.out.println(s1 == s3); // false
```

How can we realize "s1 equals s3" based on the content? The character strings are actually the same!

There are two types of equality checking in Java:

1. Identity or referential equality:
   Given two references, ref1 and ref2, both not equal `null`
   if (ref1 == ref2) → ref1 and ref2 point to the same object
2. Equality of value or equality of content
   The equals() method is predefined for all objects and can be used to check the equality of content:
   if (ref1.equals(ref2)) → ref1 and ref2 point to objects with equal values

Example (1/3)

```java
String s1 = "Hello";
String s2 = new String("Hello");

// check for referential equality
if (s1 == s2)
    System.out.println("same string object!");

// check for equality of content
if (s1.equals(s2))
    System.out.println("s1 and s2 contain the same string!");

// Output: s1 and s2 contain the same string!
```

Example (2/3)

```java
String s1 = "Hello";
String s2 = "hello"; // notice the lowercase 'h'

// check for referential equality
if (s1 == s2)
    System.out.println("same string object!");

// check for equality of content
if (s1.equals(s2))
    System.out.println("s1 and s2 contain the same string!");

// no output produced!
```

Example (3/3)

```java
String s1 = "Hello";
String s2 = s1;

// check for referential equality
if (s1 == s2)
    System.out.println("same string object!");

// check for equality of content
if (s1.equals(s2))
    System.out.println("s1 and s2 contain the same string!");

// Output:
// same string object!
// s1 and s2 contain the same string!
```

EQUALITY FOR USER DEFINED CLASSES

or comparisons between your own classes, the equals() method can be overridden.
Please be aware: The default implementation checks the two references for equality.

Example: comparing squares:
Two squares are equal when their side lengths are the same.

```
class Square {
    private int sideLength;

    //... more definitions

    public boolean equals(Object o) {
        return sideLength == ((Square) o).sideLength;
    }
}
```

> *Note that it is generally necessary to override the* `hashCode` *method whenever this method is overridden, so as to maintain the general contract for the* `hashCode` *method, which states that equal objects must have equal hash codes.*

See chapter 6 for more information on hash maps. An object's hash code is used to store the object in a hash map data structure.

Example hashCode implementation for Square:

```java
class Square {
    private int sideLength;

    //... more definitions

    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + sideLength;
        return result;
    }
}
```

2.12 STATIC ELEMENTS

You already know two static elements:

```java
// 1. main method for starting an application
static public void main (String args[]) { ... }

// 2. Output to console, out is static attribute of class System
System.out.println("Hello, world!");
```

In certain cases, one would like to define attributes or methods for all objects of a class. Such elements can be definded as `static` in Java. To access a static element, only the class name is needed. You don't have to create a class instance!

Please be aware: Access of non-static elements from static methods results in a compiler error!

What happens in class X?

```
public class X {
    private static int counter;

    public X() {
        counter++;
    }
}
```

The counter contains the number of instances of class `X`.

While attributes of a class are initialised during the creation of an object and thus via the instructions in the constructor, the question arises of how to deal with static attributes. For this purpose, Java provides a so-called static block, which is used to define instructions that are executed the first time the class is loaded, for example when an instance is created for the first time.

```java
public class X {
  private static int counter;
  static {
      counter = 10;
  }
  public X() {
    counter++;
  }
}
```

2.13 GETTER AND SETTER METHODS

Attributes are usually not defined as public to prevent uncontrolled access by other objects. For controlled access, two methods are defined for each attribute (if necessary!), one for read access and one for write

The following naming convention ensures uniform usage:

Given an attribute named `test` of type T (e.g. T := int).

```
private T test;
```

Getter for `test`:

```
public T getTest() {
    return test;
}
```

Setter for `test`:

```
public void setTest(T value) {
    test = value;
}
```

In our Square example we could have one getter and one setter:

```java
public class Square {
    // Attribute for side length
    private double sideLength;

    public double getSideLength() {
        return sideLength;
    }
    public void setSideLength(double sideLength) {
        this.sideLength = sideLength;
    }
}
```

Attributes of the type Boolean are considered a special case. In Java, the convention suggests that the getter for attributes of the type Boolean use the prefix "is" instead of "get".

```java
public class Account {
    // Attribute for locking an account
    private boolean locked;

    public boolean isLocked() {
        return locked;
    }
    public void setLocked(boolean locked) {
        this.locked = locked;
    }
}
```

RETURN OF THE CURRENT INSTANCE IN THE SETTER

It is possible for the setter to return the called object as a result. This is known as `builder pattern` or `fluent` and enables the chaining of calls, e.g. to configure an object.

```java
public class Account {
    private boolean locked;
    double balance;

    public Account setLocked(boolean locked) {
        this.locked = locked;
        return this;
    }
    public Konto setBalance(double balance) {
        this.balance = balance;
        return this;
    }
}
```

```java
// usage:
Account myAccount = new Account().setLocked(false).setBalance(1234.0);
```

2.14 PACKAGES

How can large Java projects with many classes be structured?

In order to not get lost in a large Java project, classes are organised in packages. A package is much like a folder in a file system. A package can contain classes but also other packages. Thus, a Java project usually has a tree structure.

Example1: Java runtime library

The Java runtime library, for example, contains the package java.lang for standard classes such as the class String, the complete path for using the String class is:

```
java.lang.String test = new java.lang.String("Hello, World!");
```

Example 2: println

The complete call of the println method is:

```
java.lang.System.out.println("Hello, World!");
```

ADDING A JAVA CLASS TO A PACKAGE

In the first statement of a public class you may define the package for that class (and other private classes of that file) using the keyword `package`.

```
// all follwing classes are in package2 which is in package1
public class Test { ... }
```

This means that there must be a folder `package1` containing a folder `package2`. The file `Test.java` is then located in path `package1/package2/Test.java`.

IMPORTING CLASSES

Following the package statement, classes can be listed which are to be imported. Imported classes may be used without a full package path. Instead, it is sufficient to specify the class name.

```
import java.util.Date; // import Date class for handling calendar dates
import java.util.*;    // import all classes from java.util

// instead of java.util.Date now = new java.util.Date(); we can write:
Date now = new Date();
```

STATIC IMPORTS

In Java, you can also access static attributes and methods without prefixing the class name by statically importing the corresponding class:

```
import static java.lang.Math.*;
import static java.lang.System.*;

class MathTester {
    public static void main(String[] args) {
        System.out.println(Math.sin(3.14)); // long version
        out.println(sin(3.14));             // short version
    }
}
```

In the example above, the last line can be written very short because `out` is a static attribute of the class System and `sin` is a static method of the class Math and both properties are public.

2.15 EXERCISE

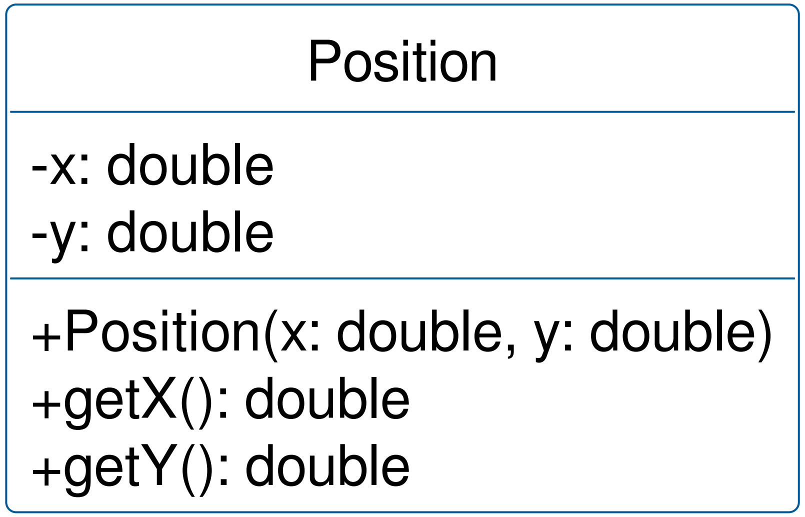
CLASS POSITION

In the following exercise, a Position class is realised. This class is intended to offer basic algorithmic operations and simplify calculation with 2D coordinates.

Unzip position-class.zip and import the folder as "Existing Maven Project".

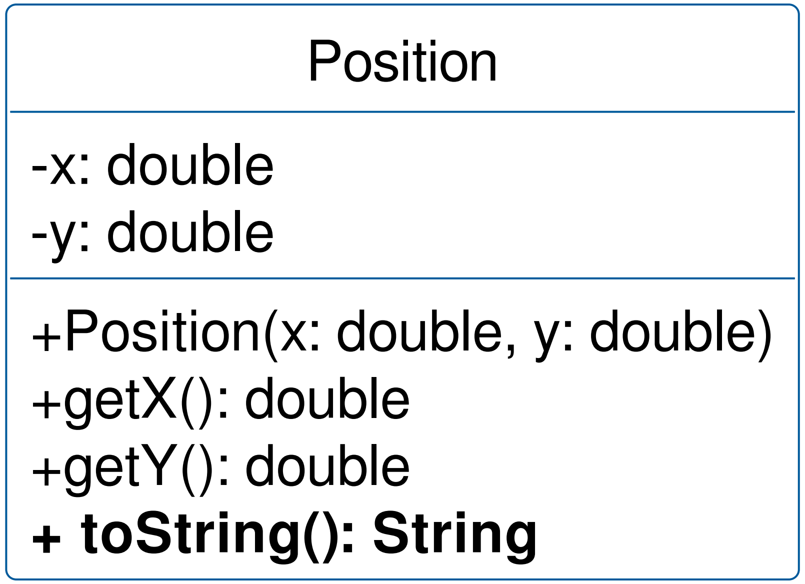
EXERCISE STEP 1: FROM UML TO JAVA

Start with the class `Position` and define the corresponding attributes, getters and a constructor.



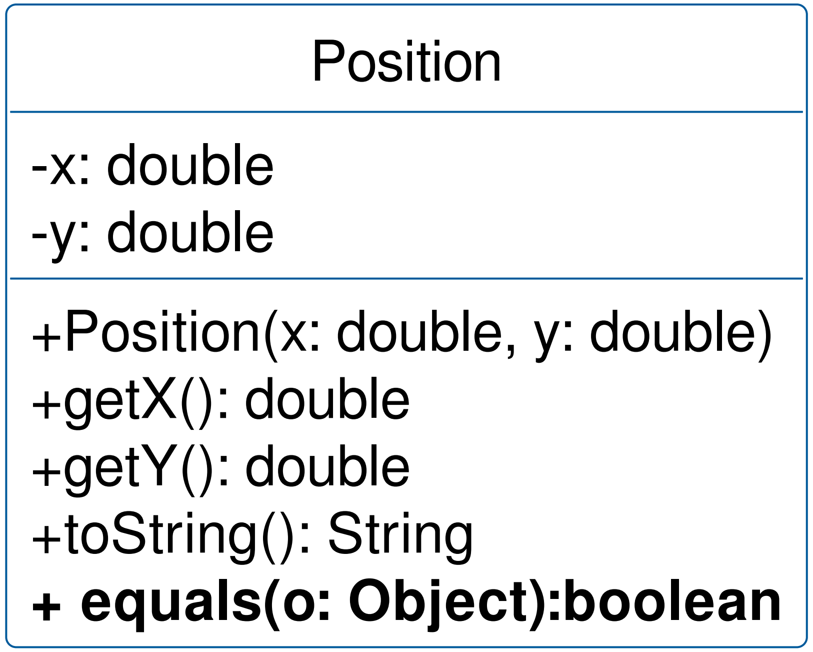
EXERCISE STEP 2: TOSTRING

Add a `toString` method and implement it. `toString()` should return a string containing the values for `x` und `y` separated by a `/`.



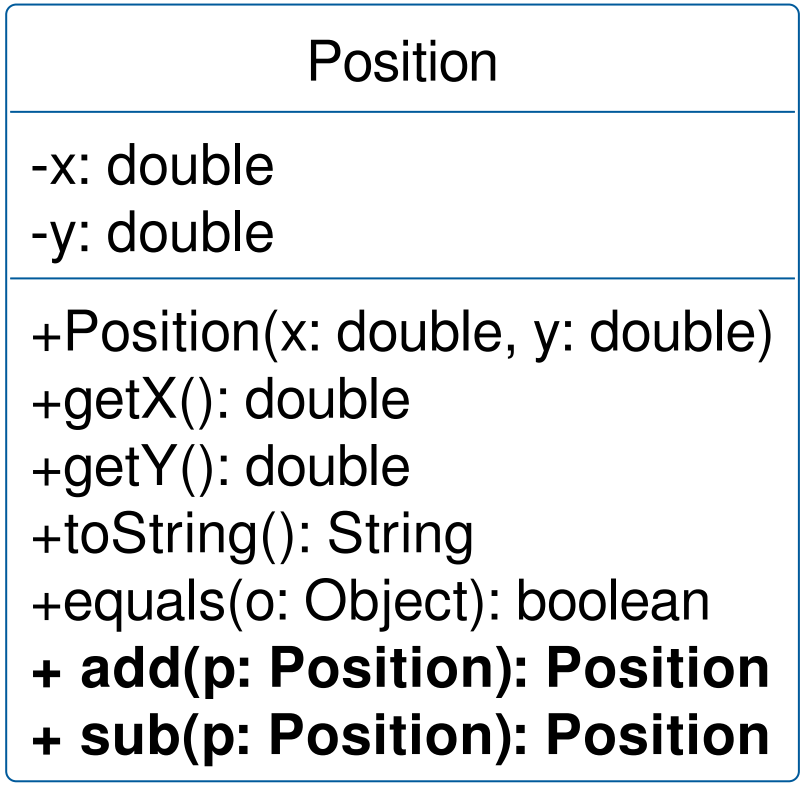
EXERCISE STEP 3: EQUALS

Add the method `equals` and implement it. Check if the parameter passed is not zero, cast the parameter to `position` and check if `x` and `y` are identical.



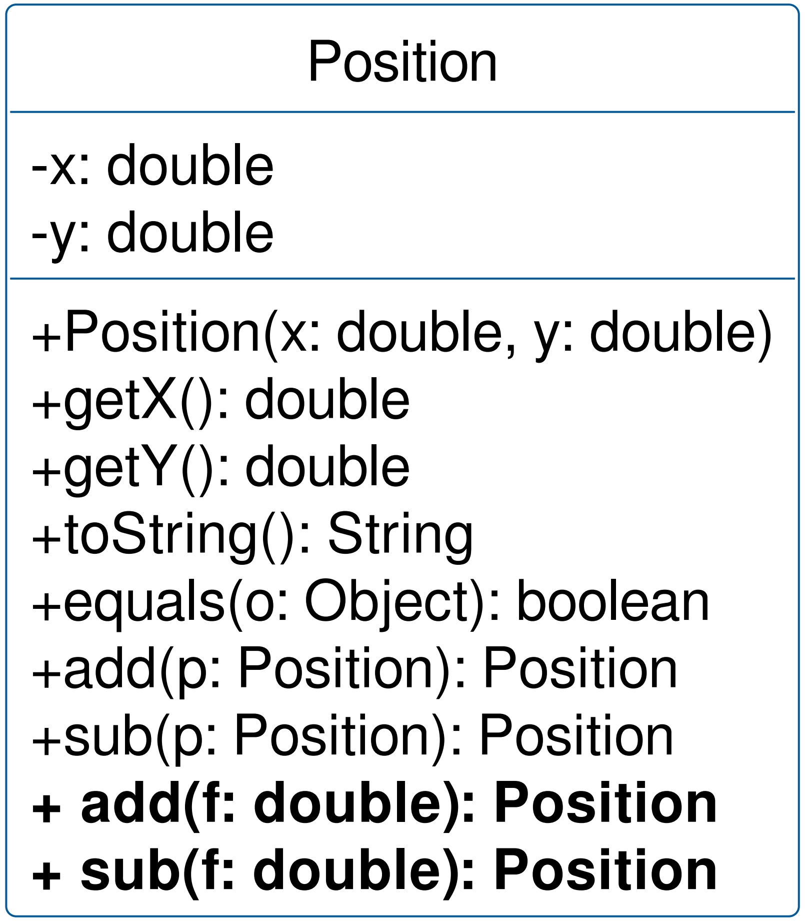
EXERCISE STEP 4: ADDING AND SUBRTRACTING

Java does not offer the possibility to overwrite operators - if you want to realise arithmetic operations between your own classes, you have to add the corresponding methods. Add the methods `add` and `sub`, calculate the new values for `x` and `y`. Finally return a new instance to enable chaining of operators.


Position
-x: double
-y: double
+Position(x: double, y: double)
+getX(): double
+getY(): double
+toString(): String
+equals(o: Object): boolean
+ add(p: Position): Position
+ sub(p: Position): Position

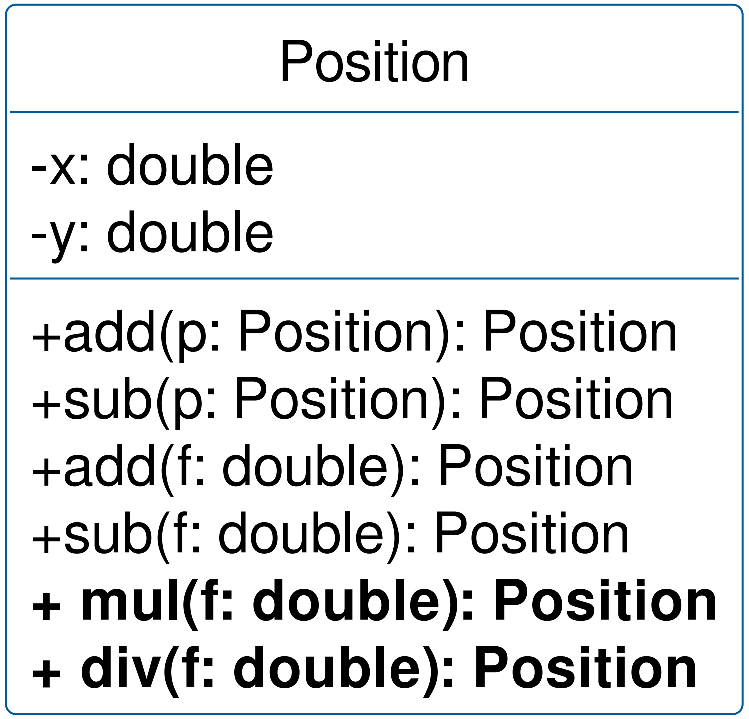
EXERCISE STEP 5: OVERLOADING ADD AND SUBTRACT

It is also conceivable that a single value is added or subtracted for both position constituents at the same time. Overload the methods accordingly.


Position
-x: double
-y: double
+Position(x: double, y: double)
+getX(): double
+getY(): double
+toString(): String
+equals(o: Object): boolean
+add(p: Position): Position
+sub(p: Position): Position
+ add(f: double): Position
+ sub(f: double): Position

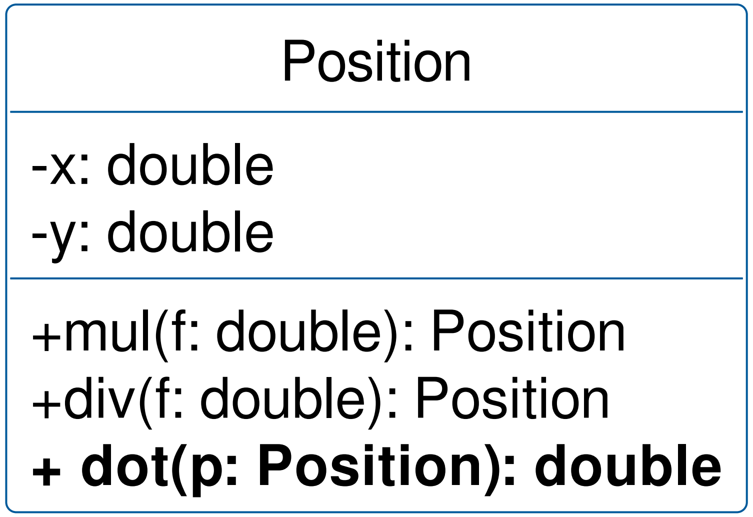
EXERCISE STEP 6: MULTIPLY AND DIVIDE

Add two operations multiplying and dividing both position constituents.


Position
-x: double
-y: double
+add(p: Position): Position
+sub(p: Position): Position
+add(f: double): Position
+sub(f: double): Position
+ mul(f: double): Position
+ div(f: double): Position

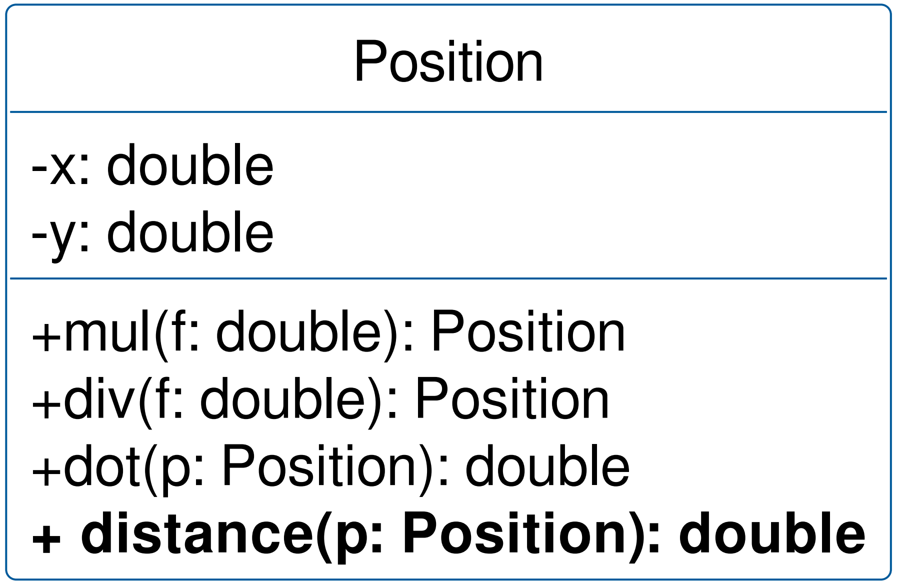
EXERCISE STEP 7: SCALAR PRODUCT

Let's extend the class `position` to include a method for calculating the scalar product: `dot = x1 * x2 + y1 * y2`.


Position
-x: double
-y: double
+mul(f: double): Position
+div(f: double): Position
+ dot(p: Position): double

EXERCISE STEP 8: DISTANCE BETWEEN TWO POSITIONS

Let's extend the class `Position` to include a method for calculating the distance between the current position and another. To do this, you must subtract the position passed from the current one and then add up the square of each dimension and finally take the root.


Position
-x: double
-y: double
+mul(f: double): Position
+div(f: double): Position
+dot(p: Position): double
+ distance(p: Position): double

Hint: Use Math.sqrt(double) to calculate the square root of a double value.

EXERCISE STEP 9: POSITIONS FOR A POLYGON

Create a main method, use the following positions.

```
Position p1 = new Position(10, 10);
Position p2 = new Position(12, 18);
Position p3 = new Position(17, 16);
Position p4 = new Position(14, 8);
Position p5 = new Position(12, 7);
```

Put these positions into an array, then iterate over each element and calculate the centre of gravity of the points.

QUESTIONS: CENTER OF GRAVITY

What is the correct center of gravity?

- ☐ 13.4 / 10.9
- ☐ 13.0 / 11.8
- ☐ 11.1 / 13.2
- ☐ 14.2 / 12.1

QUESTIONS: PERIMETER OF A POLYGON

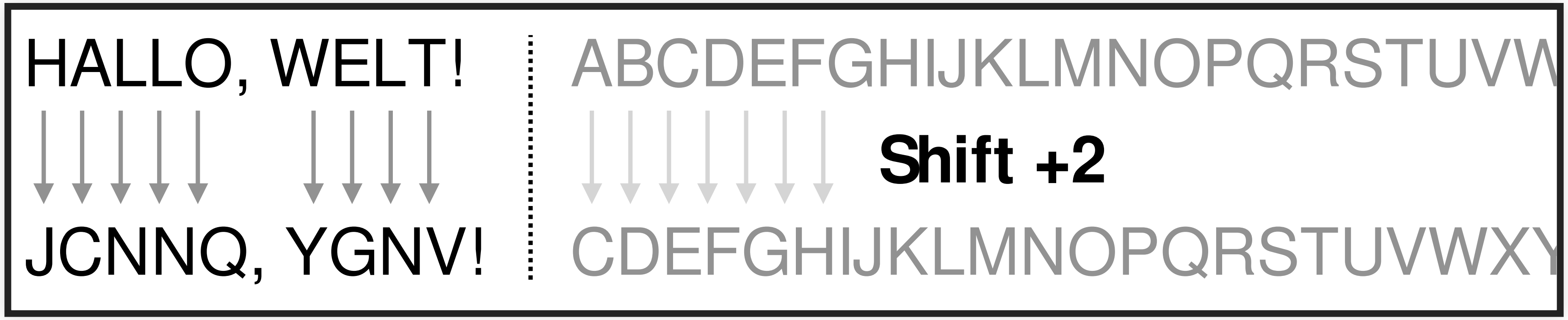
What is the correct perimeter for our previous example?

- ☐ approx. 31 units
- ☐ approx. 28 units
- ☐ approx. 29 units
- ☐ approx. 27 units
- ☐ approx. 30 units
- ☐ approx. 26 units

2.16 EXERCISE (2)

CAESAR CIPHER

The goal of this exercise is to implement the Caesar Cipher.

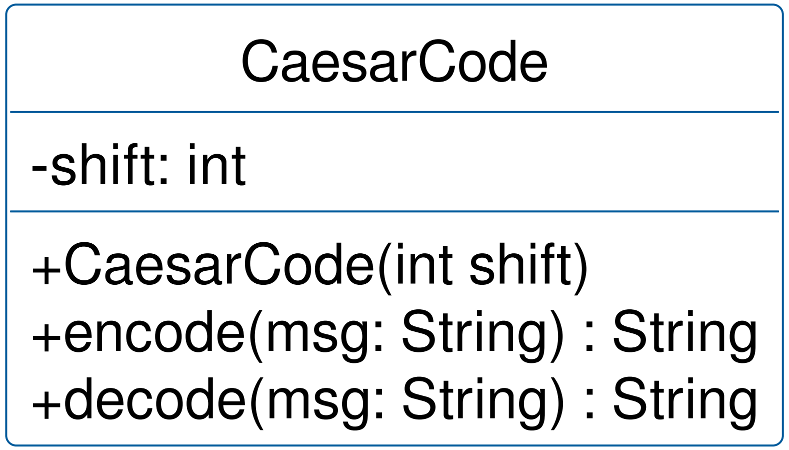


The Caesar cipher is a simple symmetric encryption method in which the letters of a message are shifted by a certain value. With a shift of 2, `A` *becomes* `C`*.*

Unzip caesar-code.zip and import the folder as "Existing Maven Project".

EXERCISE: VERSCHLÜSSELUNG

Implement your Caesar encryption algorithm in a class `CaesarCode` with two methods `encode` and `decode`. Use the following UML class model as a guide.



Hints for your implementation

- Move lowercase letters to the corresponding lowercase letter and uppercase letters to the corresponding uppercase letter. A shift of 3 transforms from **a** to **d** and from **A** to **D**.
- Ignore all other characters, including special characters like ä, ö or ü
- Decrypt by applying the shift in the opposite way. A shift of 3 transforms from **d** to **a** and from **D** to **A**.
- It is possible to do int-calculations and comparisons with characters in Java:

```
char c = 'a' + 1; // c == 'b'

if (c >= 'a' && c <= 'z') {
    // c contains lowercase letter
}
```

EXERCISE: APPLICATION

Expand the Main method. Process the prepared input and use it with your implementation.