



PGR112 – 3: Classes and objects

Object-oriented programming

Marcus Alexander Dahl /
marcusalexander.dahl@kristiania.no

Topics for today

Week 2

- Classes and objects
- Constructor
- Method Overloading
- Fields
- Getters and setters
- Access Modifiers
- `this`

Tips for catching up

- Explore Java as a language, find some code that works and tweak it. Use examples shared, from the repository or lecture slides. Don't just read it, play with it.
- Locate links from the lecture slides, try to find the corresponding topic from the lectures and find tutorials that are specific in covering those topics
- Read the first lecture, index 0, get an overview.

Small reminders

- Arguments are provided when calling the code of methods and functions. Types are not specified, you as the programmer need to provide the expected type of data, specified as parameters in method declarations.

```
System.out.println(arg1, arg2);
```

- Parameters are typed variables which a block of code one can expect as available within that method (or function).

```
void printLine(String line) {  
    System.out.print(line + "\n");  
}
```

Classes

- Classes are the foundation for objects we want to create instances of. We describe our class:
 - What attributes / fields the objects has
 - What methods the objects have (more precisely what objects can do)
- When we create (or initialize) an object from a class, we then use a special method called a constructor, and we call upon this method with a special keyword: **new**
- In common with all objects that were instantiated from the same class, is that they share methods and attributes. The objects differ based upon the observation that their attributes can have different values. Objects have different state based on the difference in data these attributes can store.

Objects

- Objects are created when we instantiate them using the new keyword, and they are created based on a class definition.
- When we create objects, we expect to be able to do something with those objects. We need a way to refer to them though. You can use the following tutorial from [Oracle](#), but as we continue with these slides, our example will be one that differs.
- You can read more about classes and objects [here](#).

Wallet-example

- `Wallet wallet = new Wallet();`

I create an object of type *Wallet*.
The reference to this object is *wallet*.

`wallet.printCoins();`

I call upon the method *printCoins*, by
using the reference *wallet* to access this
method available (publically).

Wallet-class

- This initial example is a simple abstraction of a wallet, which contains a count of (physical) coins within a wallet.

It only keeps a track of the amount of coins, nothing more.

- ```
public class Wallet {
 int coins = 0; // amount of coins initially within the wallet

 public void printCoins() {
 System.out.println(
 "This wallet contains " + this.coins + "coins."
);
 }
}
```



# Wallet-class

```
public class Wallet {
 int coins = 0;

 public void printCoins() {
 System.out.println("You have " + this.coins + " coins.");
 }
}
```

# Constructor

- But where is the constructor in the Wallet-class from the previous slide?
- All classes have a default constructor (even though we not provide code for one). This standard constructor, provided when none is given, has no parameters.

Constructors are identified by:

- Name matches the name of the class (for example, *WalLet*)
  - It can have parameters (it is also a method)
  - It returns the object
  - It is in most cases *public*
  - Constructors are special methods involved in the creation (initializing) of objects. If there is nothing
  - Constructors are most used to set start values for the different attributes an object can have.
- 
- Lets try to use a constructor from the Wallet-example provided...

# Method overloading

- We can by design have more than one constructor, as long as they have a different set of parameters. This goes for all methods within a class, and this is called method overloading. Methods can have the same name in a class, as long as they take different parameters.

- Examples:

```
public void print(String s){}
```

```
public void print(int i){}
```

```
public void print(int i, String s){}
```

```
public String print(String s){}
```

# Constructors summarized

- Constructors is a special method that we use when initializing objects, which are instances of a class.
- We call upon the constructor with the special keyword `new`, with eventual provided start values as arguments sent to the constructor.
- We can have different constructors in the same class (as long as they have different parameters)

# String?

- String is a non-primitive type, one of the very few provided by the Java language. It is also a class.
- We have used a String as such:
  - `String s = "Any text that can fit nicely between two specific characters";`
- We can also create that same String value using a constructor.
  - `String s = new String("Any text that can fit nicely between two specific characters");`
- The String-class do have more constructors available for use, but for now, do not delve into the String-class, we will do so as we progress through the course.

You can read more about it [here](#) though.

# Attributes of objects

- Attributes within objects are often called fields, instance variables or member variables.
  - These are variables that we can access inside the methods within the object.
  - This we can identify in our Wallet-example. The method `printCoins` prints out the value of the `coins` field. The method itself had access to this variable within the object. It had no access modified keyword specified before the type.
  - We can choose how fields and methods are accessible, if a method or field can only be used within the class, or accessible publically through a reference to an instantiated object. You have seen this as `public*` so far.
- Access modifiers\*

# Access modifies

- We have previously seen that we have public methods.  
(`public static void main...`)
- In simple terms, we can explain that by using the public keyword, we open a method or field up for access from outside the object or class. We are not going into details of this now, as we will explore this later in the course during the topic of inheritance.
- We will now take a look at some access methods for fields and methods. In our example today, we saw little of them. No access modifier provided means that the access modifier is default.
- The table below shows the 4 different opportunities we have when deciding which access modifier to use, where we will for now be focusing on private and public.

[javatpoint.com](http://javatpoint.com)

Bare tilgang i klassen

Tilgang «over alt»

| Access Modifier | within class | within package | outside package by subclass only | outside package |
|-----------------|--------------|----------------|----------------------------------|-----------------|
| Private         | Y            | N              | N                                | N               |
| Default         | Y            | Y              | N                                | N               |
| Protected       | Y            | Y              | Y                                | N               |
| Public          | Y            | Y              | Y                                | Y               |

# Private

- We do want to limit the access to fields and methods (specially towards other objects and possible interactions).
- We should make sure that only data (information) is accessible outside the object when (strictly) necessary.
- It is usual to set fields to be private. As such, we make sure that only the object itself can access this data (within). We give the object better control over its own state.
- Private methods defined within the class, are often helper methods to split up code (to maintain growing code), as private methods are only available to methods defined within the class.
- For example, a complicated method can be public, but split up into several calls to private methods.



# Public

- With public, we give explicit public access. A public field results in an environment where other objects can change this value (from outside).
- It is common to combine private fields with public getters and setters.
  - Getter: A method which return a value for a specific field requested.
  - Setter: A method which takes one value as a parameter (input value), which run a block of code which can replace the value stored within the private variable.

(Having this directly set the field's data using the parameter as it is provided, is pretty much the same as using the public access modifier.)

- Let me demonstrate using the previous example given ...

# IntelliJ and getters/setters

- Let us check it out ...

# Encapsulation

- To quote [tutorialspoint](#):

«**Encapsulation** is one of the four fundamental OOP concepts. The other three are inheritance, polymorphism, and abstraction.

Encapsulation in Java is a mechanism of wrapping the data (variables) and code acting on the data (methods) together as a single unit. In encapsulation, the variables of a class will be hidden from other classes, and can be accessed only through the methods of their current class. Therefore, it is also known as **data hiding**.

To achieve encapsulation in Java –

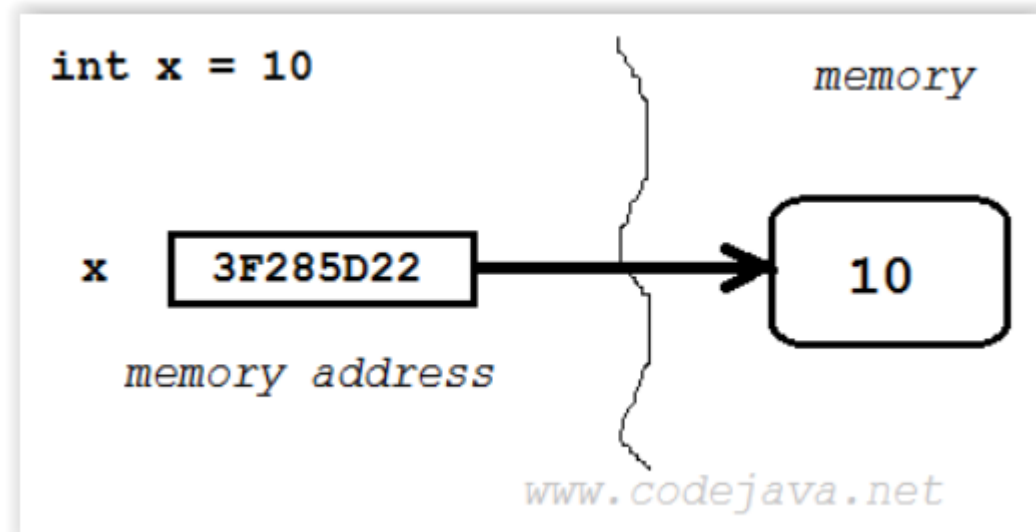
- Declare the variables of a class as private.
- Provide public setter and getter methods to modify and view the variables values.»

# Looking back on last week

- We worked primarily with primitive data types within methods.  
(local variables)
- During this step (session 3) we have learned how to instantiate, creating objects and seeing how we can have variables within an object as fields.

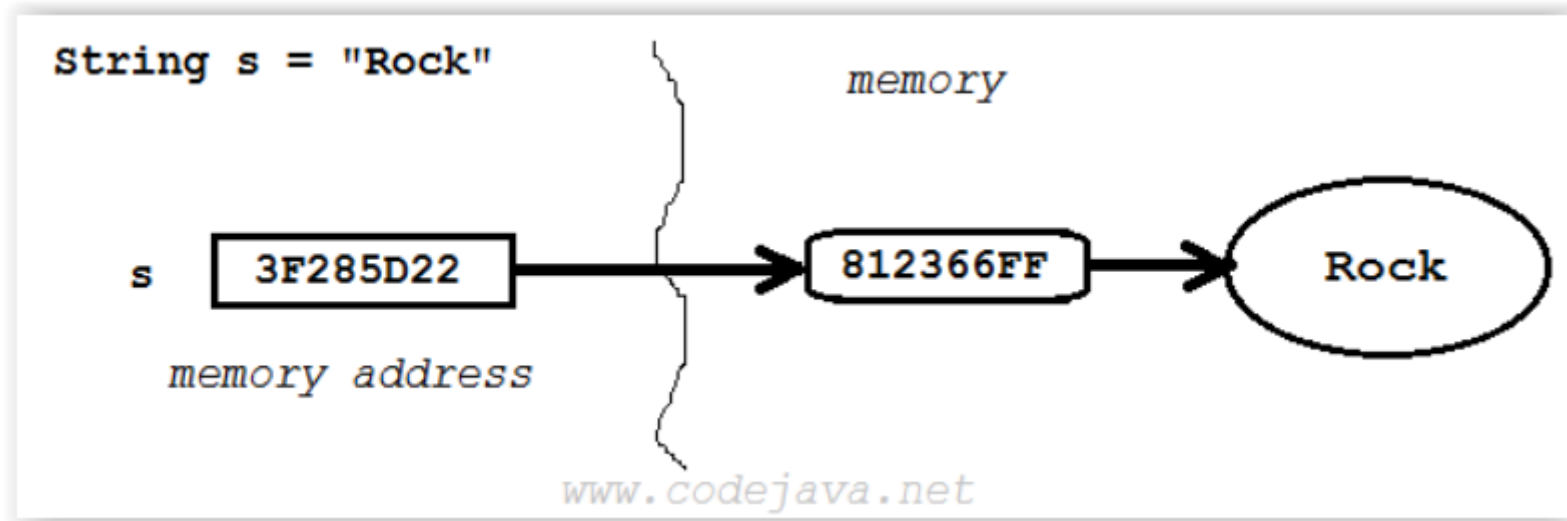
# Variables

- Primitive variables (int, char boolean etc.)



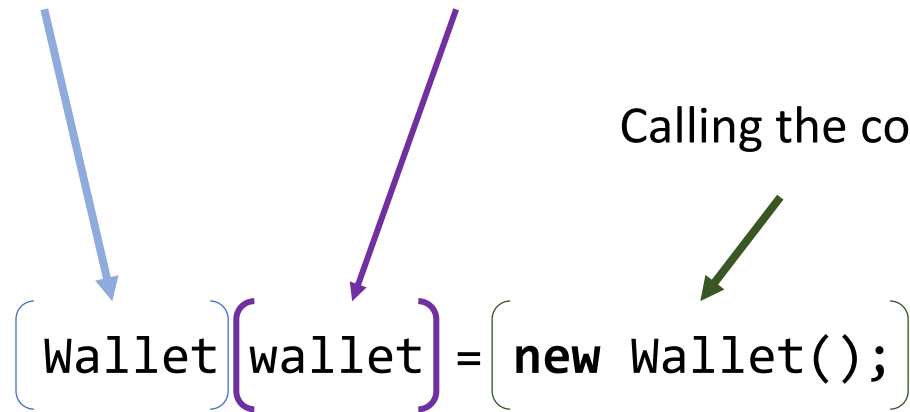
# Reference variable

- Reference variables – refers to an object



# Taking a look at our example

- The type of data this reference points to in memory



The diagram shows the code `Wallet wallet = new Wallet();` with three arrows pointing to different parts: a blue arrow to `Wallet`, a purple arrow to `wallet`, and a green arrow to `new Wallet()`.

```
Wallet wallet = new Wallet();
```

Calling the constructor – an object is instantiated and returned

We can use the reference `wallet` to access what is available in that object, for example:

```
wallet.printCoins();
```

Reference to the object

# Fields – In summary

- Fields are variables that belongs to an object. The fields and their different values all make up the (current) state of the object.
- Fields do expect a start value when the constructor has finished initializing, creating an object that is returned.
- We can device to not set values using the constructor, and those fields when then have specified default values.
- We should hide fields, as in setting the access modifier to private. Then only the object itself can access the fields.
- We can offer both fields read- and write-access, by providing code for setter and getters.



# Getter

- A method which return the value of a field requested from an object.

```
private String name; // field
```

```
public String getName() {
 return this.name;
}
```

# Setter

- A method which changes the value of a field, using the provided input.

```
private String name; // field

public void setName(String value) {
 name = value;
}
```

# *this*

- What if the name of a parameter within a method, for example the setter-method, is the same as a field within the object?

```
private String name; // field

public void setName(String name) {
 // name = name; ?
}
```

We can use *this* to reference the object itself:

```
private String name;

private void setName(String name) {
 this.name = name;
}
```

# Before we end

- Goals for this session:
  - I can make classes with fields and methods
  - I can make constructors, and know how they work
  - I can create (instantiate) objects
  - I can use a reference variable to access an object (and what that entails)
  - I can create getters and setters and understand the purpose of this (encapsulation)
  - I know that an object can use this to refer to itself.

Remember, do not just read code, play with it.  
Good luck with the tasks!

Remember, there is help available all week, use Mattermost or GitHub.