3 Classes, Objects, Inheritance

Example:

```java
class Person {
    String firstName;
    String lastName;
    long taxIdent; // must be unique!
}
```

// Manual initialization, easy to make a mistake (e.g. what about 'taxIdent'?)
Person p1 = new Person();
p1.firstName = "Max";
p1.lastName = "Mustermann";
p1.taxIdent = 12345;

Person p2 = new Person();
p2.firstName = "Fabelhaft";
p2.lastName = "Fabelhaft";
p2.taxIdent = 12345; // oops!

// Complete and consistent.
Person p1 = new Person("Max", "Mustermann", 12345);
Person p2 = new Person("Fabelhaft", "Fabelhaft", 67890);
```

Why do we need constructors?

- Ensure complete and consistent initialization after object creation
- Access (non-default) superclass constructors:
  Construct object according to definition of superclass, then add specifics
- Provide additional constructors for varying use-cases

```java
class Bicycle {
    int cadence;
    int speed;
    int gear;

    Bicycle(int c, int s, int g) {
        cadence = c;
        speed = s;
        gear = g;
    }

    Bicycle(int g) {
        cadence = 0;
        speed = 0;
        gear = g;
    }
}
```

```java
class Tandem extends Bicycle {
    int numberOfDrivers;

    Tandem(int c, int s, int g, int n) {
        super(c, s, g);
        numberOfDrivers = n;
    }
}
```
3 Classes, Objects, Inheritance

Example:

class Person {
    String firstName;
    String lastName;
    long taxIdent; // must be unique!

    Person(String fName, String lName, long lIdent) {
        firstName = fName;
        lastName = lName;
        // Use the given tax identifier 'lIdent' only if we can make sure it is unique:
        if (isUniqueTaxIdentifier(lIdent)) {
            taxIdent = lIdent;
        } else {
            System.err.println("Not unique!");
        }
    }

    Person(String fName, String lName) {
        firstName = fName;
        lastName = lName;
        // A unique tax identifier is created as a side-effect of this constructor:
        taxIdent = createUniqueTaxIdentifier();
    }

    // Complete, consistent, convenient
    Person p1 = new Person("Max", "Busermann", 12345); // first constructor is called
    Person p2 = new Person("Fabienne", "Takelauf"); // second constructor is called
}

Parameters

- **parameter list**: Passing parameters to methods or constructors

```java
int doSomething(int primitiveParameter1,
                double primitiveParameter2,
                SomeClass referenceParameter) {
    int someInt = 17 + 9;
    primitiveParameter1 = 0;
    referenceParameter = null;
    return someInt;
}
```

- Passing primitive type parameters: **Call By Value**
  Changes to parameter have no effect outside of method or constructor

```java
int x = 1;
SomeClass someObject = new SomeClass();
int y = doSomething(x, 2.345, someObject); // At this point, x still has value 1.
```

- Passing reference type parameters: **ALSO Call By Value**
  Changes to parameter have no effect outside of method or constructor

```java
int x = 1;
SomeClass someObject = new SomeClass();
int y = doSomething(x, 2.345, someObject); // At this point, someObject still references // the same object (someObject != null).
```
3 Classes, Objects, Inheritance

Parameters

- However, passing reference type parameters can be used to modify objects or arrays with a lasting effect:

```java
void doSomethingElse(int[] refParameter) {
    for (int i = 0; i < refParameter.length; i++) {
        refParameter[i] = 47;
    }
}
```

// Somewhere else...
int[] someArray = {1, 2, 3, 4, 5};
doSomethingElse(someArray);
for (int i = 0; i < someArray.length; i++) {
    System.out.print("#" + i + ": " + someArray[i]);
}

output will be: #0: 47 #1: 47 #2: 47 #3: 47 #4: 47

Why is this so?

- **Remember**: Reference type variables point to an object of the reference type.
- **Call By Value means**: When method or constructor is called, copies of corresponding variables' values are passed.
- Once method returns: Copies are destroyed.
- Reference type variables may be used to manipulate something OUTSIDE the method (or constructor).

→ "side-effect"

---

The special value **null**: null points to "nothing"

Bicycle bike1 = new Bicycle();
Bicycle sameBike = bike1;

```java
memory [simplified model]

<table>
<thead>
<tr>
<th>cell nr</th>
<th>cell name</th>
<th>cell content</th>
</tr>
</thead>
<tbody>
<tr>
<td>1149</td>
<td>someArray1</td>
<td>&lt;1150&gt;</td>
</tr>
<tr>
<td>1150</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1151</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1152</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>1327</td>
<td>refParameter</td>
<td>&lt;1150&gt;</td>
</tr>
</tbody>
</table>
```

---

```java
memory [simplified model]

<table>
<thead>
<tr>
<th>cell nr</th>
<th>cell name</th>
<th>cell content</th>
</tr>
</thead>
<tbody>
<tr>
<td>1149</td>
<td>bike1</td>
<td>&lt;1150&gt;</td>
</tr>
<tr>
<td>1150</td>
<td>bike1.cadence</td>
<td>0</td>
</tr>
<tr>
<td>1151</td>
<td>bike1.speed</td>
<td>0</td>
</tr>
<tr>
<td>1152</td>
<td>bike1.gear</td>
<td>1</td>
</tr>
<tr>
<td>1327</td>
<td>sameBike</td>
<td>&lt;1150&gt;</td>
</tr>
</tbody>
</table>
```
3 Classes, Objects, Inheritance

Returning values

Methods may return a value [corresponding to declared return type, which may also be void):

\[
\text{long faculty(int n) \{ long result = 1; for (int i = 2; i \leq n; i++) \{ result = result \times i; \} return result; \}}
\]

// Somewhere else...
\[
\text{long x = faculty(5); System.out.println("Faculty of 5 is \( + x + \) ").;}
\]

General form: \text{return expression;}

Returns the value of expression

---

The special value \textbf{null}:

- null points to "nothing"

Bicycle bike1 = new Bicycle();
Bicycle sameBike = bike1;
sameBike = null; // Has no effect on bike1.
3 Classes, Objects, Inheritance

Returning values

• Aside from primitive types, references can be returned as well:

```java
Bicycle goGetABike() {
    if (checkForSufficientFunds()) {
        return new Bicycle();
    } else {
        return null;
    }
}

// Call the method from somewhere else...
Bicycle bike = goGetABike();
```

• Corresponding objects/arrays are not "destroyed" (Remember: Reference type variables hold references to the objects, not the objects themselves!)

Calling methods

• Methods can be called from inside and outside a class:

```java
public class Bicycle {
    public int cadence = 0;

    public void changeCadence(int newCadence) {
        cadence = newCadence;
        // also: this.cadence
    }

    public void someOtherMethod() {
        changeCadence(5);
        // also: this.changeCadence
    }
}

public static void main(String[] args) {
    Bicycle bike = new Bicycle();
    bike.changeCadence(10);
    // bike.cadence == 10
    bike.someOtherMethod();
    // bike.cadence == 5;
}
```

• If needed, objects may refer to themselves as this

Access Modifiers & Packages

• Access modifiers:
  - public: Can be accessed / invoked by anybody
  - private: Can only be accessed / invoked from within same class
  - protected: Can only be accessed / invoked from within same class and its subclasses
  - <no modifier>: Can be accessed / invoked from within same package

<table>
<thead>
<tr>
<th>Class</th>
<th>Package</th>
<th>Subclasses</th>
<th>World</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>protected</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>no modifier</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>private</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
</tbody>
</table>
3 Classes, Objects, Inheritance

Calling methods

- Methods can be called from inside and outside a class:

```java
public class Bicycle {
    public int cadence = 0;
    
    public void changeCadence(int newCadence) {
        cadence = newCadence; // also: this.cadence
    }
    
    public void someOtherMethod() { // also: this.changeCadence
        changeCadence(5);
    }
    
    public static void main(String[] args) {
        Bicycle bike = new Bicycle();
        bike.changeCadence(10); // bike.cadence == 10;
        bike.someOtherMethod(); // bike.cadence == 5;
    }
}
```

- If needed, objects may refer to themselves as this