### 5 Classes, Objects, Inheritance

- **Class definition (general form):**
  ```java
  modifier class MyClass extends MySuperClass
  implements YourInterface1, ..., YourInterfaceN
  {
    // fields, constructors, methods
  }
  ``

- **(Access) modifier** for classes:
  - Certain combinations of [public, protected, private, static, final, abstract]
  ```java
  public class Mountainbike extends Bicycle {
    public Mountainbike(int startHeight, int startCadence, int startSpeed, int gear) {
      super(startCadence, startSpeed, startGear);
      seatHeight = startHeight;
    }
  }
  ```

- **Example:**
  ```java
  public class Bicycle {
    double result = 1.0;
    public double calculate(double help) {
      result = result + (help / help);
      return result;
    }
  }
  ```

- **Code snippet:**
  ```java
  long factorial(int n) {
    long result = 1;
    while (n != 0) {
      result = result * n;
      n = n - 1;
    }
    return result;
  }
  ```
### Classes, Objects, Inheritance

**Class Bicycle**

```java
public class Bicycle {
    public int cadence = 0;
    public int speed = 0;
    public int gear = 1;

    public MountainBike(int startHeight, int startCadence, int startSpeed, int startGear) {
        super(startCadence, startSpeed, startGear);
        seatHeight = startHeight;
    }

    public void applyBrakes() {
        speed = 0;
    }

    public void setCadence(int newCadence) {
        cadence = newCadence;
    }

    public void setHeight(int newValue) {
        seatHeight = newValue;
    }
}
```

**Field Declaration (general form):**

```
modifier type name;
```

**Access modifier (for fields):**

Certain combinations of `{public, protected, private, static, final}`

**Type:** Any primitive or reference type

---

### Method Declaration (general form):

```
modifier typeOfReturnValue name (parameter*) throwsClause {
    statement*;
}
```

**Access modifier (for methods):**

Certain combinations of `{public, protected, private, static, final, abstract}`

- **typeOfReturnValue:** Any primitive or reference type
- **parameter*:** (later)
- **throwsClause*:** (later)
- **statement*:** statement(s) to execute

---

**Method Declaration (general form):**

```
modifier typeOfReturnValue name (parameter*) throwsClause {
    statement*;
}
```

**Access modifier (for methods):**

Certain combinations of `{public, protected, private, static, final, abstract}`

- **typeOfReturnValue:** Any primitive or reference type
- **parameter*:** (later)
- **throwsClause*:** (later)
- **statement*:** statement(s) to execute
**5 Classes, Objects, Inheritance**

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;
    }
}
```

Example:

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

```java
class Person {
    String firstName;
    String lastName;
    long taxIDent;
}
```
Example:

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

    Person(String firstName, String lastName, long taxIdent) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    // Use the given tax identifier 'tIdent' only if we can make sure it is unique:
    if (!isUniqueTaxIdentifier(tIdent)) {
        taxIdent = tIdent;
    } else {
        System.err.println("Not unique!");
    }
}
```

Example:

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

    Person(String firstName, String lastName, long taxIdent) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    // Use the given tax identifier 'tIdent' only if we can make sure it is unique:
    if (!isUniqueTaxIdentifier(tIdent)) {
        taxIdent = tIdent;
    } else {
        System.err.println("Not unique!");
    }
}
```

Example:

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

    Person(String firstName, String lastName, long taxIdent) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    // Use the given tax identifier 'tIdent' only if we can make sure it is unique:
    if (!isUniqueTaxIdentifier(tIdent)) {
        taxIdent = tIdent;
    } else {
        System.err.println("Not unique!");
    }
}
```

Example:

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

    Person(String firstName, String lastName, long taxIdent) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    // Use the given tax identifier 'tIdent' only if we can make sure it is unique:
    if (!isUniqueTaxIdentifier(tIdent)) {
        taxIdent = tIdent;
    } else {
        System.err.println("Not unique!");
    }
}
```

```java
Person p1 = new Person("Max", "Mustermann", 123456);
Person p2 = new Person("Fabienne", "Fabelhaft", 67890);
```
Example:

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

    Person(String fName, String lName, long tIdent) {
        firstName = fName;
        lastName = lName;
    }

    // Use the given tax identifier 'tIdent' only if we can make sure it is unique:
    if (!isUniqueTaxIdentifier(tIdent)) {
        taxIdent = tIdent;
    } 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", "Mustermann", 123456); // first constructor is called
Person p2 = new Person("Fabienne", "Fabelhaft"); // second constructor is called

Example:

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

    Person(String fName, String lName, long tIdent) {
        firstName = fName;
        lastName = lName;
    }

    // Use the given tax identifier 'tIdent' only if we can make sure it is unique:
    if (!isUniqueTaxIdentifier(tIdent)) {
        taxIdent = tIdent;
    } 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", "Mustermann", 123456); // first constructor is called
Person p2 = new Person("Fabienne", "Fabelhaft"); // second constructor is called
Classes, Objects, Inheritance

Example:

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

    Person(String fName, String lName, long tIdent) {
        firstName = fName;
        lastName = lName;
        // Use the given tax identifier 'tIdent' only if we are
        // sure that it is unique. If not, use one created as
        // a side-effect
        taxIdent = createUniqueTaxIdentifier(tIdent);
    }

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

```java
Person p1 = new Person("Max", "Müller", 12345);  // first constructor is called
Person p2 = new Person("Peter", "Petersen");  // 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).
  ```
5. Classes, Objects, Inheritance

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 **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).
```

- 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\" + i + ": " + someArray[i]);
}
```

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

```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\" + i + ": " + someArray[i]);
}
```

```
output will be: #0: 47 #1: 47 #2: 47 #3: 47 #4: 47
```
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...
```java
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

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...
```java
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
The special value **null**:

- null points to "nothing"

```java
Bicycle bike1 = new Bicycle();
Bicycle sameBike = bike1;
sameBike = null;  // Has no effect on bike1.
```

---

Returning values

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

```java
long faculty(int n) {
    long result = 1;
    for (int i = 2; i <= n; i++) {
        result = result * i;
    }
    return result;
}
```

// Somewhere else...
long x = faculty(5);
System.out.println("Faculty of 5 is " + x + ");

- General form: **return expression**;

  Returns the value of **expression**
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!)

---

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!)
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!)

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`

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`

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`
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 (final, static)

- Access modifiers:
  - `static`: field or method bound to class instead of object
class-method, class-variable as opposed to instance-method, instance-variable
  - `final`:
    - fields: cannot be changed (constants)
    - methods: cannot be overridden (later)
    - classes: cannot be subclassed

```java
final class MyClass {
    static int sameValueForAllInstances = 3;
    final int constantValue = 5;
    static final int constantValueForAllInstances = 7;
    static void methodOne() { /* ... */ }
    final void methodTwo() { /* ... */ }
    static final void methodThree() { /* ... */ }
}
```

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

- Packages:
  - Encapsulate a set of classes and interfaces
  - Hierarchical organization
  - Declaration: package myfirstpackage;
  - Examples: java.math, de.tum.wzw
Access Modifiers (final, static)

- **static**: field or method bound to class instead of object class-method, class-variable as opposed to instance-method, instance-variable
- **final**:
  - fields: cannot be changed (constants)
  - methods: cannot be overridden (later)
  - classes: cannot be subclassed

```java
final class MyClass {
    static int sameValueForAllInstances = 3;
    final int constantValue = 5;
    final static int constantValueForAllInstances = 7;

    static void methodOne() { /* ... */ }
    final void methodTwo() { /* ... */ }
    static final void methodThree() { /* ... */ }
}
```

Access Modifiers (final, static)

- **static**: field or method bound to class instead of object class-method, class-variable as opposed to instance-method, instance-variable
- **final**:
  - fields: cannot be changed (constants)
  - methods: cannot be overridden (later)
  - classes: cannot be subclassed

```java
final class MyClass {
    static int sameValueForAllInstances = 3;
    final int constantValue = 5;
    final static int constantValueForAllInstances = 7;

    static void methodOne() { /* ... */ }
    final void methodTwo() { /* ... */ }
    static final void methodThree() { /* ... */ }
}
```
Overriding, Hiding

**Overriding methods**

- **Why?**
  - Let **subclasses** provide a more specialized version of an instance-method

- **How?**
  - Subclass defines an instance-method with **same signature** (name plus number and types of parameters) as defined by super-class

```java
class Bicycle {
    int speed;
    public void speedUp(int increment) {
        speed = speed + increment;
        System.out.println("superclass instance-method");
    }
}

class MountainBike extends Bicycle {
    public void speedUp(int increment) {
        speed = speed + 2 * increment;
        System.out.println("subclass instance-method");
    }
}

MountainBike mb = new MountainBike();
mb.speedUp(10); // mb.speed == 20
```

**Output will be:** subclass instance-method
Overriding, Hiding

- **Overriding methods**
  - Let subclasses provide a more specialized version of an instance-method
  - Subclass defines an instance-method with same signature (name plus number and types of parameters) as defined by superclass

```java
class Bicycle {
    int speed;
    public void speedUp(int increment) {
        speed = speed + increment;
        System.out.println("superclass instance-method");
    }
}
class MountainBike extends Bicycle {
    public void speedUp(int increment) {
        speed = speed + 2 * increment;
        System.out.println("subclass instance-method");
    }
}
MountainBike mb = new MountainBike();
mb.speedUp(10); // mb.speed == 20
```

output will be: **subclass instance-method**

Overriding, Hiding

- **Hiding class-methods**
  - Let subclasses provide a more specialized version of a class-method
  - Subclass defines a class-method with same signature (name plus number and types of parameters) as defined by superclass

```java
class Bicycle {
    public static void myClassMethod(int someInt) {
        System.out.println("superclass class-method");
    }
}
class MountainBike extends Bicycle {
    public static void myClassMethod(int someInt) {
        System.out.println("subclass class-method");
    }
}
Bicycle.myClassMethod(10); // "superclass class-method"
MountainBike.myClassMethod(10); // "subclass class-method"
```

Overriding, Hiding

- **Polymorphism**: subclass objects may be assigned to superclass variables

  ```java
  MountainBike mountainBike = new MountainBike();
  Bicycle bicycle = mountainBike;
  ```

  → **Essential feature** of object oriented software

- Only methods and fields defined by the the superclass "portion" of the object may be accessed; and the overridden ("right") methods are called

  ```java
  bicycle.gear = 3; // Ok, gear defined in class Bicycle
  bicycle.speed = 20; // ERROR! speed is not a field in class Bicycle
  mountainBike.setHeight = 20; // Ok
  mountainBike.speedUp(5); // Overridden method in subclass MountainBike is used
  bicycle.speedUp(10); // Overridden method in subclass MountainBike is used
  MountainBike.myClassMethod(99); // "subclass class-method"
  Bicycle.myClassMethod(99); // "superclass class-method"
  ```
Polymorphism

- **Polymorphism**: subclass objects may be assigned to superclass variables
  
  ```java
  MountainBike mountainBike = new MountainBike();
  Bicycle bicycle = mountainBike;
  ```

  → Essential feature of object-oriented software

  - Only methods and fields defined by the superclass "portion" of the object may be accessed; and the overridden ("right") methods are called
  
  ```java
  bicycle.gear = 3; // Ok, gear defined in class Bicycle
  bicycle.seatHeight = 20; // ERROR! seatHeight is not a field in class Bicycle
  mountainBike.setHeight = 20; // Ok
  ```

  MountainBike.speedUp(5); // Overridden method in subclass MountainBike is used
  Bicycle.speedUp(10); // Overridden method in subclass MountainBike is used

  MountainBike.myClassMethod(99); // "subclass class-method"
  Bicycle.myClassMethod(99); // "superclass class-method"
Overriding, Hiding

- **Overriding methods**
  - Let subclasses provide a more specialized version of an instance-method
  - Subclass defines an instance-method with same signature (name plus number and types of parameters) as defined by superclass

```java
class Bicycle {
    int speed;
    public void speedUp(int increment) {
        speed += increment;
        System.out.println("superclass instance-method");
    }
}

class MountainBike extends Bicycle {
    public void speedUp(int increment) {
        super(2 * increment); // call overridden method from superclass
        System.out.println("subclass instance-method");
    }
}

MountainBike mb = new MountainBike();
mb.speedUp(10); // mb.speed == 20
```

Output will be:
- superclass instance-method
- subclass instance-method

Polymorphism

- **Polymorphism**: subclass objects may be assigned to superclass variables
  ```java
  MountainBike mountainBike = new MountainBike();
  Bicycle bicycle = mountainBike;
  ```

  → **Essential feature** of object oriented software

- Only methods and fields defined by the the superclass "portion" of the object may be accessed; and the 'overridden ("right") methods are called

  ```java
  bicycle.gear = 3; // OK, gear defined in class Bicycle
  bicycle.seatHeight = 20; // ERROR! seatHeight is not a field in class Bicycle
  mountainBike.speedUp(5); // OK
  ```

  ```java
  mountainBike.speedUp(5); // Overridden method in subclass MountainBike is used
  bicycle.speedUp(10); // Overridden method in subclass MountainBike is used
  MountainBike.myClassMethod(99); // "subclass class-method"
  Bicycle.myClassMethod(99); // "superclass class-method"
  ```