02: Object Orientation

Network Oriented Software

Stefan Huber Harry Schmuck Maximilian Tschuchnig Eduard Hirsch
ITS, FH Salzburg

Summer 2021

Section 1

Classes and Objects in Java

OOP in Java

We assume you know these four aspects of object-oriented programming (OOP):

- Abstraction
  Hiding details and complexities of inner workings, exposing simplicity
- Encapsulation
  Cohesion of data and code, data hiding, forming has-a relationships
- Inheritance
  Hierarchical subtyping, forming is-a relationships
- Polymorphism
  Type-depending behavior

Classes in Java

OOP languages organize code in classes.

- In Java, all code is in classes; it is purely OOP.
- Each class `A` is implemented in a source file `A.java`
- The keyword `class` followed by the class name defines a class.

```java
// A.java
class A {
}
```

Class members

A class comprises

- Properties implemented as fields (member variables)
- Functionality implemented as methods (member functions)

```
class Person {
  private String name;

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

Each of them has one of four access specifiers that specify visibility:

- `private` Can only be accessed within the class
- `protected`: See later
- Default: See later
- `public`: Can be accessed from everywhere

Object creation

An instance of a class is called object.

- In Java, the operator `new` creates class instances.

```
1 Person p = new Person();
```

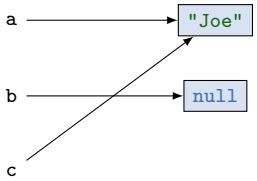
Variable versus object

The variable is *not* the object:

- We say that the variable binds to the object.
- Multiple variables may bind to the same object.

```
1 Person a = new Person();
2 Person b = new Person();
3 Person c = a;                 // Binds to the same object as 'a'.
4 c.setName("Joe");             // The Person behind 'a' and 'c' is called "Joe" now
```

After the above code we have three variables bound to two objects.



Parameter passing

Parameters behave like variables and they are passed by value.

- But think of the *variable per se* being passed by value, not the object it binds to!

```
public void f(Person p, Person q) {
  // Now both paramters (as variables) bind to the object q. But the objects
  // have not changed.
  p = q;
  // Here we actually change the object that p binds to.
  p.setName("Joe");
}

public void test() {
  Person alice = new Person();
  Person bob = new Person();
  // Now bob's name will become "Joe", but alice is not touched.
  f(alice, bob);
}
```

So Java variables are more like C++ pointer variables, but without having access to the actual addresses.

Inheritance

A class can extend (derive from, inherit from) one class, its superclass.

- In Java, the keyword `extends` is used.
- The derived class is also called subclass. The superclass is also called base class.

```
1 class Student extends Person {
2 }
```

A class inherits all properties and functionality from its superclass.

- The access specifiers of the superclass are taken over.
- A class can access protected members of its superclass.

There is no multiple inheritance in Java.

- Also, if a class does not extend any class then it implicitly extends the class `Object`.
- Conclusion: The class hierarchy in Java is a tree with `Object` at its root.

Inheritance – an “is-a” relation

When `Student` extends `Person` then for the type system a `Student` *is a* `Person`.

- A `Person` variable can bind to a `Student` object.
- Hence, we can pass a `Student` argument for a `Person` parameter.
- The *Liskov substitution principle* takes is-a verbatim.

```
1 // A Student is a Person
2 Person p = new Student();
```

Static versus dynamic type:

- The variable `p` has the static type `Person`.
- But since it binds to a `Student` object it has the dynamic type `Student`.

Characteristics of objects and relationships between classes

Three characteristics define an object:

- State: What properties does an object posses?
- Behavior: What can an object do?
- Identity: What distinguishes two objects from each other?

Three basic relationships exist between classes:

- Inheritance: The is-a relationship between subtype and supertype.
- Association: The has-a relationship between a class and its field. Associations may be categorized into Aggregations (class functions also in absence of the association) or Compositions (class can not exist without).
- Dependence: The uses-a relationship when methods of one class modify or uses (instances of) other classes.

Default field initialization

- Booleans are initialized to `false`.
- Numeric fields are initialized to zero.
- Objects are initialized to `null`.

It is poor style to rely on default initialization:

- Explicitly initialize fields instead!

Constructors

The `new` operator for object creation calls a constructor:

- Like a method whose name is the class name and no return type.
- Primarily used to initialize the object's state.

```
class Person {
  private String name;      // Initialized to null by default

  Person() {
    name = "";
  }
  Person(String name) {
    this.name = name;       // this is an implicit parameter to all methods: the instance
  }
}
```

There can be more than one constructor.

- The no-argument constructor has zero parameters.
- Java knows method overloading. The overload resolution is based on argument types:

```
Person p = new Person("Joe");
```

Implicit no-argument constructor

If no constructor is defined then there is an implicit no-argument constructor:

- It essentially does nothing, i.e., all fields are initialized with their default values.
- If there is at least one constructor then there is no implicit no-argument constructor.

Constructors calling constructors

- If the first statement in a constructor is `this()` then the constructor calls another constructor of the same class.
- This way, code duplication in constructors can be reduced. (C++ 11 knows that too.)

```
1  class Person {
2    private String name;
3    private int age;
4
5    Person() {
6      this("");
7    }
8    Person(String name) {
9      this(name, 0);
10   }
11   Person(String name, int age) {
12     this.name = name;
13     this.age = age;
14   }
15 }
```

Constructor of the superclass

- If the first statement in a constructor is `super()` then the constructor of the superclass is called with the supplied arguments.
- If the subclass does not explicitly call a superclass constructor then the no-argument constructor of the parent is called. If it does not have one – also no implicit one – the compiler reports an error.

Field initialization

In general, a field can be initialized in the following way:

- At the field declaration.
- In an initialization block, but this is very uncommon.
- In a constructor.

```
class Person {
  private String name = "";
  private int dayofbirth;

  { // Initialization block
    int ms = System.currentTimeMillis();
    dayofbirth = ms / 1000 / 60 / 60 / 24;
  }

  Person() {
  }

  Person(String name) {
    this.name = name;
  }

  Person(String name, int dateofbirth) {
    this.name = name;
    this.dateofbirth = dateofbirth;
  }
}
```

Object destruction

There are no destructors to free allocated memory. But sometimes we still need to release resources when an object is destructed.

- Java knows the finalize methods, which is called before the garbage collector destroys the object.
- But there is no guarantee when the garbage collector does so.

Packages

In Java it is possible to group classes in packages.

- Package names (except the standard Java ones) start usually with a domain name in reverse, e.g. `at.ac.fhsalzburg.nos`. It is a naming convention in Java to use lower case characters for package names only.
- With the help of packages, the uniqueness of class names can be guaranteed, as classes with the same name in different packages can still be differentiated. It is similar to namespaces in C++.

```
1 package com.example;
2
3 class A {
4 }
```

- Packages can be nested into each other, for example `java.util` is nested in `java`. For the compiler, there is no relationship between nested packages.
- The class `com.example.A` resides in the file `com/example/A.java`, so nested packages indicate nested directories. The `package` instruction in which package classes are defined.

Class importation

- A class can use all classes that are in its own package as well all public classes from others.
- This can either be done by adding the full package name in front of every class name, or by using the import statement.
- This is like using namespaces in C++ and not like `#include` statements.

```
1 import com.example.*;            // Import all classes from package com.example
2 import com.example.nos.Person;   // Import this specific class
```

Access Modifiers

The table of access modifiers and their visibility:

Modifier	Class	Package	Derived class	'Outside'
public	Y	Y	Y	Y
protected	Y	Y	Y	N
default	Y	Y	N	N
private	Y	N	N	N

Notice: Default means "package protected", so declare your fields private to ensure encapsulation!

Also classes have access modifiers, and only public classes can be accessed outside the package.

```
1 public class Person {
2 }
```

Final fields

The value of fields defined as `final` cannot be change after initialization (cf. const in C++).

- Fields declared as final must be initialized upon object construction.
- A final field of a class type cannot re-bind to different object, but the object can be modified!
- Hence, useful for fields whose type is immutable (e.g. String) or primitive.

```
1 class Person {
2   final Person mother;
3   final Person father;
4
5   Person (Person mother, Person father) {
6     this.mother = mother;
7     this.father = father;
8   }
9 }
```

Static fields

- A static field is like a “class field” rather than an “object field”.
- It exists exactly once, even if no or multiple object have been created.
- Static variables are rather rare, static constants more common.

```
class Physics {
  final static double VACUUM_PERMEABILITY = 1.256e-6; // H/m
  final static double SPEED_OF_LIGHT = 300e6;         // m/s
}
class Person {
  static int totalCount = 0;
  final int id;

  Person () {
    totalCount++;
    id = totalCount;
  }
}
```

Static methods

- Static methods do not operate on an object; they can be called by `classname.method()`
- They can access static fields and static methods.
- They cannot access `this` or object fields.
- The `main` method does not operate on an object, which is why it is a static method.

Class design guidelines

- Always keep fields private.
- Always initialize fields.
- Not each fields always needs individual getter and setter methods.
- If a class has too many fields with a basic type then maybe the class should be split.
- If a class violates the *single-responsibility principle* then break it up.