Java Essentials

Object-Oriented Design, IV1350

Contents

Contents

1 Arrays and Lists

Array

An array is appropriate if the number of elements is *fixed and known*.

```
int[] myArray = new int[5];
```

List

- It is better to use a `java.util.List` if the number of elements is *not both fixed and known*.

```
import java.util.ArrayList;
import java.util.List;

public class Lists {
    public static void main(String[] args) {
        List myList = new ArrayList();
        myList.add("Hej");
```

```
 8          myList.add(3);
 9      }
10  }
```

- A **List** can contain objects of any class, this example stores a **String** (line 7) and an **Integer** (line 8).

Generic List

- The list content can be restricted to objects of one specific class.
  - Adding **<String>** on line six specifies that the list may only contain **String** objects.
  - Adding **<>** on line seven specifies that this holds also for the created **ArrayList**.

```
 1  import java.util.ArrayList;
 2  import java.util.List;
 3
 4  public class Lists {
 5      public static void main(String[] args) {
 6          List<String> myList =
 7              new ArrayList<>();
 8          myList.add("Hej");
 9          myList.add("Hopp");
10      }
11  }
```

Generic List (Cont'd)

- A generic list can be iterated using a for-each loop, see lines 11-13.

```
 1  import java.util.ArrayList;
 2  import java.util.List;
 3
 4  public class Lists {
 5      public static void main(String[] args) {
 6          List<String> myList =
 7              new ArrayList<>();
 8          myList.add("Hej");
 9          myList.add("Hopp");
10
11          for(String value : myList) {
12              System.out.println(value);
13          }
14      }
15  }
```

2 Objects

What is an Object?

- The goal of object-oriented programming is to declare classes that *group data and methods* operating on that data.

- A class represents an *abstraction*, for example *person*. An object of the class represents a specific *instance* of the class, for example the person *you*.

Code Example

- Create project in NetBeans
- Create class

Use `static` Very Restrictively

- Static fields are *shared by all objects* of the class.
- If for example the account balance was static, all accounts would have the same balance. Such a program would be useless.
- Since fields can not be static, neither can methods since *static methods can only access static fields*.
- Static fields and methods are normally *not used at all*, except in few very special cases.

Creating New Objects

Whenever we want to create a *new account*, we create a *new object* of the **`Account`** class. This is done with the operator **`new`**.

```
Account acct = new Account(1234567, 100);
```

Code Example

- Create an **`Account`** object and use it
- Demonstrate the debugger

3 Constructors

Providing Initial Values

- The constructor is used to *provide initial values* to newly created objects.

```
 1  public class Account {
 2    private long acctNo;
 3    private int balance;
 4
 5    public Account(long acctNo, int balance) {
 6      this.acctNo = acctNo;
 7      this.balance = balance;
 8    }
 9    //The methods are not showed.
10  }
```

- The values passed to the constructor are saved in the object's fields on lines 6 and 7.
- Sending parameters to a constructor is just like sending parameters to a method.

Calling the Constructor

```
Account acct = new Account(1234567, 100);
```

- The constructor is invoked when an new object is created.
- Parameters are passed to the constructor just the same way parameters are passed when an ordinary method is called.

The Variable `this`

The variable **`this`** always refers to the current object.

```
1  public class Account {
2    private long acctNo;
3    private int balance;
4
5  public Account(long acctNo, int balance) {
6      this.acctNo = acctNo;
7      this.balance = balance;
8  }
```

- Lines 6 and 7 illustrate the use of **`this`**.
- **`this.balance`** on line 7 refers to the field declared on line 3.
- **`balance`** on line 7 refers to the constructor parameter declared on line 5.
- These are two different variables.

More Than One Constructor

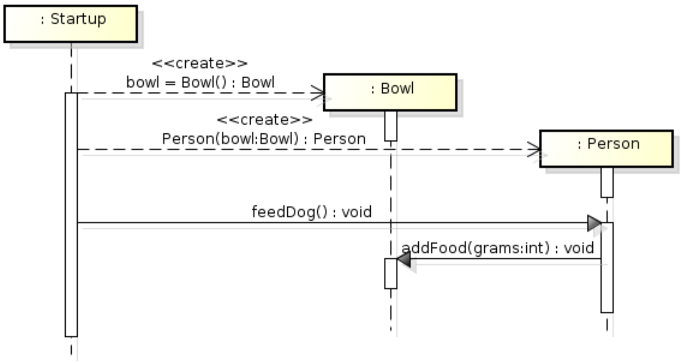
```
1   public class Account {
2       private long acctNo;
3       private int balance;
4
5       public Account(long acctNo) {
6           this(acctNo, 0);
7       }
8
9       public Account(long acctNo, int balance) {
10          this.acctNo = acctNo;
11          this.balance = balance;
12      }
13  }
```

- We need more constructors if we do not always provide the same set of initialization parameters.
- The constructor on lines 5-7 is used when no initial balance is specified.
- Calls constructor on lines 9-11, with **`balance`** = `0`.

4 References

A Reference Is a Value

- The **`new`** operator *returns a reference* to the newly created object.



- A reference can, *like any other value*, be stored in variables, sent to methods, sent to constructors, etc.

- Whenever the **`new`** operator is used, a new object with a new reference is created. Many bugs arise because *wrong reference* is used.

Code Example

- Passing references

- *It is impossible to follow the course without understanding the following example.*

5 Exceptions

Exception Changes Execution

- Method throwing exception is interrupted.
- Execution continues in **`catch`** block in calling method.

```
public class Main {
  public static void main(String[] args) {
    try {
      ClassThatThrowsException ctte = new ClassThatThrowsException();
      System.out.println("before call to methodThatThrowsException");
      ctte.methodThatThrowsException(true);
      System.out.println("after call to methodThatThrowsException");
    } catch (Exception e) {
      System.out.println("in catch block");
    }
  }
}

public class ClassThatThrowsException {
  public void methodThatThrowsException(boolean throwException) throws Exception {
    System.out.println("Before throw");
    if (throwException) {
      throw new Exception("Information about the exception");
    }
    System.out.println("After throw");
  }
}
```

The code above prints the following.

```
before call to methodThatThrowsException
Before throw
in catch block
```

Code Example

- The `catch` block need *not be in calling method*.
- Can be placed further up in the method call stack.
- Illustrated in the following code example.

Runtime Exceptions

- All examples so far have been *checked exceptions*.
- There are also *runtime exceptions*, which inherits the class `java.lang.RuntimeException`.
- Runtime exceptions do not have to be specified in a `throws` clause.

6 Javadoc

Javadoc

- Javadoc is used to generate *html pages* with code documentation.
- It is *strongly recommended* to write Javadoc for *all* declarations (classes, interfaces, methods, fields etc) that are not private.
- A Javadoc comment is written between `/**` and `*/`.
- The tags `@param` and `@return` are used to document method *parameters* and *return values*.

Code Example

Write Javadoc comments and generate html pages.

7 Annotations

Annotations

- Annotations provide information about a piece of source code for the compiler, JVM or something else.
- Usually used for properties unrelated to the functionality of the source code, for example to configure security, networking or multithreading.
- Starts with the at sign, @, for example @`SomeAnnotation`.
- May take parameters, for example @`SomeAnnotation(someString = ``abc'', someBoolean = true)`

8 Interfaces

Interface Is a Contract

- An *interface is a contract*. A class implementing the interface must fulfill the contract specified by the interface.
- The contract is specified as a *set of methods*. The implementing class must provide implementations for those methods.
- The methods must do what is intended in the interface. This should be documented in the interface.
- All declarations in an interface are always *public*.

Interface Example

The following interface defines the contract *Write the specified string to the log*.

```java
public interface Logger {
    /**
     * Writes the specified message to the log.
     * @param message This string is written
     *                to the log.
     */
    void log(String message);
}
```

The interface is implemented by the following class.

```java
public class FileLogger implements Logger {
...
    public void log(String message) {
        //write to file
    }
}
```

The `@Override` Annotation

- The `@Override` annotation specifies that the *annotated method should be inherited* from a superclass or interface.
- A *compiler error* will result if the method is not inherited.
- *Always use* `@Override` for inherited methods since it eliminates the risk of accidentally specifying a new method.
- For example accidentally naming the method `logg` instead of `log` in the implementing class in the previous example.

9 Inheritance

Inheritance

Everything in the superclass that is not private *is also present* in the the subclass.

```
public class Superclass {
    public void methodInSuperclass() {
        System.out.println(
        "Printed from methodInSuperclass");
    }
}

public class Subclass extends Superclass {
    public static void main(String[] args) {
        Subclass subclass = new Subclass();
        subclass.methodInSuperclass();
    }
}
```

The program above prints the following.

```
Printed from methodInSuperclass
```

Override (Omdefiniera)

- A method in the subclass with the *same signature* as the method in the superclass will *override* the superclass' method.
- A method's signature consists of its name and parameter list.
- Overriding means that the *overriding method will be executed* instead of the overridden.
- Do not confuse with overloading (överlagra), which is to have methods with same name but different signatures, due to different parameter lists. This has nothing to do with inheritance.

Override Example

```
public class Superclass {
    public void overriddenMethod() {
        System.out.println("Printed from overriddenMethod" +
                           " in superclass");
    }
}

public class Subclass extends Superclass {
    @Override
    public void overriddenMethod() {
        System.out.println("Printed from overriddenMethod" +
                           " in subclass");
    }

    public static void main(String[] args) {
        Subclass subclass = new Subclass();
        subclass.overriddenMethod();
    }
}
```

The program above prints the following.

```
Printed from overriddenMethod in subclass
```

To Call the Superclass

super is a *reference to the superclass.*

```
public class Superclass {
    public void overridenMethod() {
        System.out.println("Printed from Superclass");
    }
}

public class Subclass extends Superclass {
    public void overridenMethod() {
        System.out.println("Printed from Subclass");
        super.overridenMethod();
    }

    public static void main(String[] args) {
        Subclass subclass = new Subclass();
        subclass.overridenMethod();
    }
}
```

The program above prints the following.

```
Printed from Subclass
Printed from Superclass
```

The *assigned instance* is executed, not the declared type.

```
public class Superclass {
    public void overriddenMethod() {
        System.out.println("Printed from overriddenMethod" +
                           " in superclass");
    }
}

public class Subclass extends Superclass {
    @Override
    public void overriddenMethod() {
        System.out.println("Printed from overriddenMethod" +
                           " in subclass");
    }

    public static void main(String[] args) {
        Subclass subclass = new Subclass();
        subclass.overriddenMethod();
        Superclass superclass = new Subclass();
        superclass.overriddenMethod();
    }
}
```

The program above prints the following.

```
Printed from overriddenMethod in subclass
Printed from overriddenMethod in subclass
```