Chapter 8

Object Communication

Lecture slides for:

*Java Actually: A First Course in Programming*

Khalid Azim Mughal, Torill Hamre, Rolf W. Rasmussen
ISBN: 978-1-844480-418-4
http://www.ii.uib.no/~khalid/jact/

*Permission is hereby granted to use these lecture slides in conjunction with the book.*

*Modified: 26/5/07*
Overview

- Responsibilities and roles
- Defining good abstractions
- Structured programming
- Redundant code
- Communication and cooperation
- Associations
- Ownership
- Encapsulation
- Class design
- Method overloading
- Documenting source code
- Example: CD-collection
- Generating documentation (javadoc)
Responsibilities and Roles

• The properties and operations defined in a class determine:
  – Which abstraction the class represents.
  – The responsibility domain of the class
  – What roles objects of the class fulfill.
  – Which program code ought to part of the class.

• Advantages of good abstractions
  – Easy to understand the program
  – Easy to modify the program
  – Can reduce the size of the program
  – Easy to identify and remember what the different parts of the program do.
  – Promote reuse of code

• Usefulness of good abstractions becomes clear as the size of the program increases.
Creating good abstractions

- What is a good abstraction?
- A good abstraction:
  - Binds properties and operations that belong together
  - Makes it possible to avoid duplication of code
  - Limits the scope of future changes
  - Makes it easy to understand the program:
    - Gives classes and objects easily understandable responsibilities and roles
    - Hides details that are not necessary for the overall understanding of the abstraction
- Determine which methods will be useful to have in order to manipulate the fields of a class.
- Remove and hide methods and fields which are not directly useful in a class.
- Do not lump together many responsibilities. Let the role of each object be as clear as possible.
- Abstraction to represent an object often comes before its implementation.
- The implementation process can provide insights and ideas about new abstractions that can be useful.
Programming methodology: *structured programming*

- Behaviour of objects is realized by applying *structured programming*.

- Actions in a method are described by the following *control structures* only:
  - Sequence (block)
  - Choice (conditional execution)
  - Repetition (loops)

- Results in methods that are simpler
  - to understand,
  - to test and debug,
  - to modify and maintain.
Duplicate program code

- Makes program maintenance difficult.
- Same modifications must be done in several places in the code.
- Implies that program code should be restructured.

Example:

```java
> java Report1
import java.util.Scanner;
public class Report1 {
    public static void main(String[] args) {
        Scanner keyboard = new Scanner(System.in);
        System.out.print("Number of cars: ");
        int NumOfCars = keyboard.nextInt();
        System.out.print("Number of boats: ");
        int NumOfBoats = keyboard.nextInt();
        System.out.print("Number of trains: ");
        int NumOfTrains = keyboard.nextInt();
        System.out.print("Number of motor cycles: ");
        int NumOfMotorCycles = keyboard.nextInt();
        System.out.print(NumOfCars);
```
if (NumOfCars == 1) {
    System.out.print(" car, ");
} else {
    System.out.print(" cars, ");
}

System.out.print(NumOfBoats);
if (NumOfBoats == 1) {
    System.out.print(" boat, ");
} else {
    System.out.print(" boats, ");
}

System.out.print(NumOfTrains);
System.out.print(" train and ");

System.out.print(NumOfMotorCycles);
if (NumOfMotorCycles == 1) {
    System.out.println(" motor cycle.");
} else {
    System.out.println(" motor cycles.");
}
• Total: 38 LOC (Lines Of Code).
• Each new transport type will require approximately 8 new lines and impacts 3 old lines.
• If the prompt is changed, e.g. "How many cars?", then this change must be done for several other lines.
• Easy to make a mistake. For example, copying the code for motor cycles to handle buses, and forgetting to update the conditional expression.
  System.out.print(numOfBuses);
  if (numOfMotorCycles == 1) {
      System.out.println(" bus.");
  } else {
      System.out.println(" buses.");
  }
• How to avoid code duplication:
  – Use loops
  – Extract common behavior into methods
  – Use arrays instead of handling each value separately.
Code with less duplication and better abstraction

- Introduce a new abstraction to simplify the code: NumberOFTimes.

```java
import java.util.Scanner;
class NumberOFTimes {
    String singular, plural;
    int numOfTimes;

    NumberOFTimes(String singular, String plural) {
        this.singular = singular;
        this.plural = plural;
        Scanner keyboard = new Scanner(System.in);
        System.out.print("Number of times "+ plural + ": ");
        numOfTimes = keyboard.nextInt();
    }
    void print() {
        if (numOfTimes == 1) {
            System.out.print("1 "+ singular);
        } else {
            System.out.print(numOfTimes + " "+ plural);
        }
    }
}
```
// Client that uses NumberOfTimes class.
public class Report2 {
    public static void main(String[] args) {
        NumberOfTimes[] items = {
            new NumberOfTimes("car", "cars"),
            new NumberOfTimes("boat", "boats"),
            new NumberOfTimes("train", "trains"),
            new NumberOfTimes("motor cycle", "motor cycles")
        };

        for (int i = 0; i < items.length; ++i) {
            items[i].print();
            int remaining = items.length - 1 - i;
            switch (remaining) {
                case 1: System.out.print(" and "); break;
                case 0: System.out.println("."); break;
                default: System.out.print("", ");
            }
        }
    }
}

- Total: 30 LOC
- Exactly same behaviour, and to introduce a new type costs hardly one line of code and has no impact on the existing code.
- To introduce a new prompt only requires to replace one line of code in the NumberOfTimes class:
  System.out.print("How many " + plural + "? ");
Communication and cooperation

• Method calls in Java are used to:
  – Make objects cooperate
  – Query objects about information
  – Give information to other objects
  – Delegate work to other objects
  – Move information among objects using parameters and return values.

• In order to call a method on an object, we need a reference to the object.
• How to obtain a reference to an object:
  – Create a new object with the new operator. This is not particularly useful if you want to refer to an existing object.
  – Ask another object for a reference.
  – Store the reference in a local variable. The reference is not accessible once the method finishes executing.
  – Store the reference in a field variable. This creates a long-lasting association between the object who has the field and the object that the field refers to.
## Associations: UML notation

<table>
<thead>
<tr>
<th>Multiplicity</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Only one object can fulfil the association. A Wing object is attached to exactly one Plane object.</td>
</tr>
<tr>
<td>0..1</td>
<td>Zero or one object can fulfil the association. A Plane object has 0 or 1 Person object that is a pilot.</td>
</tr>
<tr>
<td>0..*</td>
<td>Any number of objects can fulfil the association.</td>
</tr>
<tr>
<td>n</td>
<td>Only $n$ objects can fulfil the association, where $n &gt; 1$. A Plane object has exactly 2 Wing objects.</td>
</tr>
<tr>
<td>0..n</td>
<td>Up to and including $n$ objects can fulfil the association, where $n &gt; 1$. A Plane object can carry zero or maximum $n$ Person objects that are passengers.</td>
</tr>
</tbody>
</table>
### Example of Relationships

```java
class Person {
    String name;
    // ...
}

class Wing {
    void adjustFlaps(float lift) {
        // ...
    }
    // ...
}

class Plane {
    Wing leftWing;
    Wing rightWing;
    Person pilot;
    // ...
}
```

- A plane has (owns) two wings.
- A wing is attached to a plane.
- A plane has one person (at a time) who is the pilot.
- A person can be a pilot of a plane (at a time).
Object Ownership

- Owner of a object has often/usually:
  - a field variable with a reference value that refers to the object it owns.
  - the main responsibility to maintain a reference to the object.
  - control over the life time of the object.
Encapsulation

*Abstraction* focuses on the *visible behaviour* of an object, while *encapsulation* focuses on *hiding* the *implementation* that offers this behavior.

A class has 2 *views*:

- *Contract* that corresponds to our abstraction of the behaviour that is common for all objects of the class.
  
  - Consists of names of methods and documentation on how *clients* (i.e. other objects) can use objects of the class, and how each method ought to be used.
  
  - Purpose: allows objects to communicate with each other.

- *Implementation* corresponds to representation of the abstraction and *mechanisms* that provide the desired behavior.
  
  - Consists of field variables and Java statements that provided the behaviour when executed.
Hiding Information

- Encapsulation leads to separation of contract and implementation.
  - Objects as black boxes where the implementation is hidden.
    *Contract:*
    - **What** a class offers - clients use the behavior of the class.
    *Implementation:*
    - **How** a class implements its behavior, not a concern of the client.

- Hiding in Java:
  - Classes can be hidden.
  - Individual fields and methods in a class can be hidden.
  - Access modifiers can be used in the declaration to specify encapsulation.

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>Class, field or method is accessible everywhere</td>
</tr>
<tr>
<td>private</td>
<td>Fields and methods are only accessible to the code in the class in which they are declared. Classes that are &quot;top-level&quot; cannot be private.</td>
</tr>
</tbody>
</table>
Data Abstraction: *Hiding information*

- Hide implementation details of a class:
  - Use access modifiers to hide fields and methods.
  - Allows changes in the class implementation with minimal impact on clients.
  - Leads to fewer dependencies between classes.
  - Increases reuse of code.
  - Results in program parts that are *black boxes*. 
Class Design

- Use *standard* format for class declarations.

- A class with too many properties or too many responsibilities/behavior should be split into several classes.

- Use meaningful names for classes and methods that reflect their purpose.

- Field variables:
  - Usually declared private.
  - Ought to be initialized in a constructor.
  - Not always necessary with *selector*/mutator methods.

- Always initialize local variables of a method.

- Avoid too many primitive data types in a class. It is a sign to rethink their purpose.
Method Overloading

- Signature of a method consists of its name and formal parameter list.
- Which method is executed depends on, among other things, its signature.
- When a method is called, there must be a method with a signature that matches the signature of the call. For overloaded methods, this is determined at compile time.
- Constructor overloading is also a form of method overloading.
- Changing the return value alone is not sufficient to overload a method.

```java
class Light {
    private int numOfWatts;
    public double cost(int numOfHours) {
        double KWH_price = 0.35; // cents pr. kiloWatt hour
        double price = ((double)numOfWatts * numOfHours/ 1000)* KWH_price;
        return price;
    }
    public double cost(int numOfHours, double KWH_price) {
        double price = ((double)numOfWatts * numOfHours/ 1000)* KWH_price;
        return price;
    }
}
```
Documenting the source code

- **Main goal:** A program should be easy to read and understand.
- **What is important to document:**
  - This is not always obvious.
  - Methods: what they do is more important than how they are implemented.
  - Fields: what is their purpose
  - Classes:
    - What abstraction the class represents and what responsibilities it has.
    - What role the objects of the class fulfill and what their typical usage.
  - Algorithms that are used (pseudo code)
  - Overall description of how the different parts of the program fit together.
- **How much documentation is enough?**
  - The bigger and more complex a program, the more important the need for good documentation.
  - Choosing meaningful names and writing code that is easy to understand decreases the need for documentation.
  - Documentation should be useful. i.e. have a purpose.
Which method is easy to understand?

// Compute the average of numbers in an array.
public class Average {
    /** This method calculates the average by traversing over the array. */
    static double compute(double[] array) {
        double value = 0;
        for (int index = 0; index < array.length; ++index) {
            value += array[index]; // Add the current value from the array
        }
        return value/array.length;
    }
    static double average(double[] values) {
        double sum = 0;
        int numOfValues = values.length;
        for (int i = 0; i < numOfValues; ++i) {
            sum += values[i];
        }
        return sum/numOfValues;
    }
    public static void main(String[] args) {
        double[] values = { 1.0, 3.2, 4.1, 7.8, 9.3, 11.4, 8.5 }; // Add the array values
        System.out.println(compute(values));
        System.out.println(average(values));
    }
}
Javadoc comments

/** This is a javadoc comment. */

- Javadoc comments in the source code can be used by the javadoc tool to generate documentation.
- Information provided using *markup tags* in Javadoc comments is extracted by the javadoc tool to generate the documentation.
- Javadoc comments can be used to document:
  - Classes
  - Fields
  - Methods

<table>
<thead>
<tr>
<th>Markup Tags</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>@param &lt;parameterName&gt; &lt;description&gt;</td>
<td>Description of a formal parameter of a method.</td>
</tr>
<tr>
<td>@return &lt;description&gt;</td>
<td>Description of the return value from a method.</td>
</tr>
</tbody>
</table>
Example: CD Collection

- Using several classes.
- Using constructors.
- Creating objects, including arrays.
- Calling methods.
- Using access modifiers.
- Using javadoc.
/**
 * Administration of a CD collection.
 * Uses objects of the classes CDCollection and CD.
 */
public class CDAdmin {
    public static void main (String[] args) {
        CDCollection music = new CDCollection(4);
        music.printCDCollection();

        music.insertCD (new CD(115.50, 4));
        music.insertCD (new CD(200,12));
        music.insertCD (new CD(99.99, 5));

        music.printCDCollection();

        music.insertCD (new CD(150.99, 12));
        music.insertCD (new CD(123.50, 15));

        music.printCDCollection();
    }
}
/** This class represents a collection of CDs. */
public class CDCollection {
/** Index of the last CD inserted. */
private int indexOfLastCD;
/** Array that holds the CDs. */
private CD[] CDArray;
/** Create a CD collections. 
 * @param maxNumOfCDs Maximum number of CDs that can be stored in the 
 * collection. */
public CDCollection (int maxNumOfCDs) {
    indexOfLastCD = -1;
    CDArray = new CD[maxNumOfCDs];
}
/** Insert a CD. 
 * @param newCD CD to be inserted. */
public void insertCD (CD newCD) {
    if (!isFull()) {
        ++indexOfLastCD;
        CDArray[indexOfLastCD] = newCD;
        System.out.println ("New CD inserted.");
    } else {
        System.out.println ("Collection is full.");
    }
}
/** @return Number of CDs in the collection at the moment. */
public int getNumOfCDs() { return (indexOfLastCD +1); }

/** @return true if the collection is full. */
public boolean isFull() { return getNumOfCDs() >= CDArray.length; }

/** @return true if the collection is empty. */
public boolean isEmpty() { return (getNumOfCDs() <= 0); }

/** Print information about the collection. */
public void printCDCollection() {
    System.out.println("*******************************");
    System.out.println("Number of CDs: " + getNumOfCDs());
    System.out.printf("Value of collection: £ %.2f%n", totalValueOfCollection());
    System.out.printf("Average value of a CD: £ %.2f%n", averageValueOfCD());
    System.out.println("*******************************");
}
/** @return Total value of the CDs in the collection. */
public double totalValueOfCollection () {
    if (isEmpty()) return 0.0;
    double totalValue = 0.0;
    for (int i = 0; i <= indexOfLastCD; i++) {
        totalValue += CDArray[i].getPrice();
    }
    return totalValue;
}

/** @return Average price of a CD. */
public double averageValueOfCD () {
    if (isEmpty()) return 0.0;
    return totalValueOfCollection() / getNumOfCDs();
}
} // End of CDCollection
/** This class represents a CD. */
public class CD {

/** Price of the CD. */
private double CDPrice;
/** Number of tracks on the CD. */
private int numOfTracks;

/** Create a CD.
 * @param price Price of the CD.
 * @param tracks Number of tracks on the CD.
 */
public CD(double price, int tracks) {
    CDPrice = price;
    numOfTracks = tracks;
}

/** @return Price of the CD. */
public double getPrice() { return CDPrice;}
/** @return Number of tracks on the CD. */
public int getNumOfTracks() { return numOfTracks;}"
Output from the program:
*******************************
Number of CDs: 0
Value of collection: £ 0.00
Average value of a CD: £ 0.00
*******************************
New CD inserted.
New CD inserted.
New CD inserted.
*******************************
Number of CDs: 3
Value of collection: £ 415.49
Average value of a CD: £ 138.50
*******************************
New CD inserted.
Collection is full.
*******************************
Number of CDs: 4
Value of collection: £ 566.48
Average value of a CD: £ 141.62
Javadoc

1. javadoc <names of Java files>
2. javadoc -private <names of Java files>

1. Generate documentation for public, package and protected declarations (classes, interfaces, methods, fields).
2. Generate documentation for all declarations (classes, interfaces, methods, fields).

- Example of usage:
  javadoc CD.java CDCollection.java CDAdmin.java
  javadoc *.java
  javadoc -private *.java
### Field Summary

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>private</td>
<td>CD[]</td>
<td>CDArray Array that holds the CDs.</td>
</tr>
<tr>
<td>private</td>
<td>int</td>
<td>indexOfLastCD Index of the last Cd inserted.</td>
</tr>
</tbody>
</table>

### Constructor Summary

- **CDCollection(int maxNumOfCDs)**
  - Create a CD collections.

### Method Summary

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>double</td>
<td>averageValueOfCD()</td>
<td></td>
</tr>
<tr>
<td>int</td>
<td>getNumOfCDs()</td>
<td></td>
</tr>
<tr>
<td>void</td>
<td>insertCD(CD newCD)</td>
<td>Insert a CD.</td>
</tr>
<tr>
<td>boolean</td>
<td>isEmpty()</td>
<td></td>
</tr>
<tr>
<td>boolean</td>
<td>isFull()</td>
<td></td>
</tr>
<tr>
<td>void</td>
<td>printCDCollection()</td>
<td>Print information about the collection.</td>
</tr>
<tr>
<td>double</td>
<td>totalValueOfCollection()</td>
<td></td>
</tr>
</tbody>
</table>
Experience

• Ability to create good abstractions, write good code and document the source code correctly comes with experience.

• Ways to gain experience:
  – Write source code.
  – Compile code and find errors.
  – Read code other people have written.
  – Experiment with existing code.
  – Maintain code over a long period.

• Experience helps you to appreciate good code that is properly documented.

• As always: Practice makes perfect.