Software and Programming I

Object-Oriented Programming in Java

Roman Kontchakov

Birkbeck, University of London
Outline

- Object-Oriented Programming
- Public Interface of a Class
- Instance Variables
- Instance Methods and Constructors
  - Sections 8.1 – 8.6 (7.1 – 7.5 in 1/e)
Java Compilation and JRE

source
HelloWorld.java

compiler
javac

bytecode
HelloWorld.class

public static void main(String[] args) {
    ...
}

running program
Virtual Machine (VM)
Java Program: Example

```java
public class PrintHelloWorld {
    // each program is a class
    public static void main(String arg[]) {
        for (int i = 1; i <= 10; i++)
            System.out.println("Hello, World!");
    }
}
```
// Method descriptor #15 ([Ljava/lang/String;)V
// Stack: 2, Locals: 2
public static void main(java.lang.String[] arg);
    0  iconst_1
    1  istore_1 [i]
    2  goto 16
    5  getstatic java.lang.System.out : java.io.PrintStream [16]
    8  ldc <String "Hello, World!"> [22]
   13  iinc 1 1 [i]
   16  iload_1 [i]
   17  bipush 10
   19  if_icmple 5
   22  return

Local variable table:
    [pc: 0, pc: 23] local: arg index: 0 type: java.lang.String[]
    [pc: 2, pc: 22] local: i index: 1 type: int

Stack map table: number of frames 2
    [pc: 5, append: {int}]
    [pc: 16, same]
Object-Oriented Programming

- Tasks are solved by collaborating objects
- Each object has its own set of data, together with a set of methods that act upon the data

- A class describes a set of objects with the same behaviour (i.e., methods)

- Objects are constructed with the new operator:

  `Scanner in = new Scanner(System.in);`
Classes and Objects

**Class** represents a set of objects that share a common structure and a common behaviour.

**Objects** are instances of classes.

Encapsulation

- Every class has a **public interface**: a collection of methods through which the objects of the class can be manipulated.

- Encapsulation is the act of providing a public interface and **hiding** implementation details.

- Encapsulation enables **changes in the implementation** without affecting users of the class.
Encapsulation

Encapsulation hides the details of the implementation of an object

Example: Cash Register

- A cashier who rings up a sale presses a key to start the sale; then rings up each item. A display shows the amount owed as well as the total number of items purchased.

- We want the following methods on a cash register object:
  - add the price of an item
  - get the total amount owed and the count of items purchased
  - clear the cash register to start a new sale
A simulated cash register

public class CashRegister {
    /* private data */
    ...
    /* methods (public interface) */
    public void addItem(double price) { /* */ }
    public double getTotal() { /* */ }
    public int getCount() { /* */ }
    public void clear() { /* */ }
}
Using the CashRegister

- constructing an object

```java
CashRegister register1 = new CashRegister();
```

- invoking methods

```java
register1.addItem(1.95);
register1.addItem(2.99);
System.out.println(register1.getTotal() + " " + register1.getCount());
```
Instance Variables

- An object holds **instance variables** that are accessed by its methods

```java
public class CashRegister {
    private int itemCount;
    private double totalPrice;
    // the rest of the class
}
```
Instance Variables

- Every instance of a class has its own set of instance variables.

<table>
<thead>
<tr>
<th>register1: CashRegister</th>
<th>register2: CashRegister</th>
</tr>
</thead>
<tbody>
<tr>
<td>itemCount = 1</td>
<td>itemCount = 3</td>
</tr>
<tr>
<td>totalPrice = 1.95</td>
<td>totalPrice = 7.67</td>
</tr>
</tbody>
</table>

Values of the instance variables determine the state of the object.

- private instance variables can only be accessed by the methods of the same class.

```
register1.itemCount = 0; // ERROR
```
**Object: State, Behaviour & Identity**

**NB:** equal ≠ identical

Instance Methods

```java
public void addItem (double price) {
    ...
}
```

- all instances variables should be private
- most methods should be public

it is useful to classify the methods as

accessors and mutators
Instance Methods: Accessors

- An **accessor** method just queries the object for some information without changing it.

```java
1 public class CashRegister {
2     // ...
3     public double getTotal() {
4         return totalPrice;
5     }
6     public int getCount() {
7         return itemCount;
8     }
9 }
```
A mutator method changes the object on which it operates.

```java
public class CashRegister {
    // ...
    public void addItem(double price) {
        itemCount++;
        totalPrice += price;
    }
}
```
Constructors

- A constructor initialises the instance variables of an object.
- The name of the constructor is the name of the class (and no return type, not even `void`).

```java
1 public class CashRegister {
2     // ...
3     public CashRegister() {
4         itemCount = 0;
5         totalPrice = 0;
6     }
7 }
```
Constructors (cont.)

- By default,
  - numbers are initialized as 0,
  - Booleans as false, and
  - object references as null

  what about String s;?

- If no constructor is provided,
  a constructor with no parameters is generated
class CashRegister {
    /* private data (instance variables) */
    private int itemCount;
    private double totalPrice;
    /* methods (public interface) */
    public void addItem(double price) {
        itemCount++; totalPrice += price;
    }
    public void clear() {
        itemCount = 0; totalPrice = 0;
    }
    public double getTotal() { return totalPrice; }
    public int getCount() { return itemCount; }
}
class CashRegister {

/* methods (public interface) */
public void addItem(double price) {
}
public void clear() {
}
public double getTotal() {
}
public int getCount() {
}

NB: the rest are implementation details, which may be changed without affecting the users of the class
```java
public class CashRegisterTest {
    public static void main(String[] args) {
        CashRegister reg1 = new CashRegister();
        reg1.addItem(2.95);
        reg1.addItem(1.99);
        System.out.println(reg1.getCount());
        System.out.println((reg1.getCount() == 2) ? "OK" : "FAIL");
        System.out.printf("%.2f\n", reg1.getTotal());
        System.out.println((reg1.getTotal() == 4.94) ? "OK" : "FAIL");
    }
}
```
Java Compilation

source
CashRegister.java
CashRegisterTest.java

compiler
javac
CashRegister.class
CashRegisterTest.class

bytecode

Virtual Machine
java

public static void main(String[] args) {
...
}

NB: statements must be inside methods!
Encapsulation: Why?

- What if we want to support a method `void undo()` (cancels the last item)?

- What if we want to implement `CashRegister` using the fixed-point arithmetic (so that `12.92` is `1292`)?

- Instance variables are “hidden” by declaring them `private`, but they are not hidden very well at all...
Take Home Messages

- Encapsulation enables changes in the implementation
- Public interface: a collection of public methods
- Methods: accessors and mutators
- Every instance of a class has its own set of instance variables
- All instance variables should be private
- A constructor initialises the instance variables