1 Constructing Objects

Objects of a class are created by calling the constructor of the class with the \texttt{new} operator.

- Constructors are methods of the class and they always have the same name with the class itself.
- A class can have more than one constructors. Each one of them must have a different number and/or type of parameters.

\textit{Syntax:}

\begin{verbatim}
new ClassName(parameters)
\end{verbatim}

Note, that if no constructor is explicitly defined for a class, then the compiler will synthesise a default no-argument constructor, which can be used to create objects of the class. This will happen, only if no constructor is defined at all for a class. A zero-argument constructor will not be created by default, if a programmer defines a constructor with one or more arguments.

1.1 Constructor Example: The Rectangle Library class

The \texttt{Rectangle} class is defined in the Java library (package \texttt{java.awt}).

- A \texttt{Rectangle} object contains a set of numbers, specifying an area in a coordinate space by its top-left point \((x, y)\), its width, and its height.

\begin{verbatim}
Rectangle r1 = new Rectangle(5, 10, 20, 30);
\end{verbatim}
In the above code:

1. The constructor of the Rectangle class accepting four ints as parameters is called, to create a Rectangle object.
2. The created object has a top left corner at (5, 10), and has a width equal to 30 and a height equal to 30.
3. The created object is returned, and it is stored (its address) in variable r1.

To find out what constructors are available for a class, the documentation for the class must be consulted or the code for the class be examined.

An alternative constructor of the Rectangle class could be called to create an object:

```java
Rectangle r2 = new Rectangle();
```

The no-arguments constructor, creates a Rectangle object whose top-left corner is at (0, 0) in the coordinate space, and whose width and height are both zero.

## 2 Packages

Classes (including library classes) belong to a package. Thus, a package is a collection of classes.

- To be able to use a class belonging to a package in a program, the class must first be imported in the program. There are two ways to do that:

  1. Import all the classes of the package into the program, e.g.

      ```java
      import java.awt.*;
      ```

  2. Import only the class that needs to be used, e.g.

      ```java
      import java.awt.Rectangle;
      ```

Note that the approach of importing all the classes of a package into a program, does not incur more overhead than the approach of importing a single class. This is because the Java Virtual Machine will actually load a class only when it is needed.

**Example**

```java
import java.awt.Rectangle;

public class MoveTester {
    public static void main(String[] args) {
        Rectangle box = new Rectangle(5, 10, 20, 30);

        // Move the rectangle
        box.translate(15, 25);
    }
}
```
// Print information about the moved rectangle
System.out.println("After moving, the top-left corner is:");
System.out.println(box.getX() + ", " + box.getY());
}
}

Note that in the last statement of the above program, the + operator is applied into strings. The operator creates a new String object which contains the concatenation of the characters in its String arguments.

3 Implementing Classes

Implement a class which simulates a bank account. The following operations must be available (abstraction mechanism for describing a real bank account):

- Deposit money.
- Withdraw money.
- Get the current balance.

/**
 * A bank account has a balance that can be changed by deposits and withdrawals.
 */
public class BankAccount {
  private double balance;

  /**
   * Constructs a bank account with a zero balance.
   */
  public BankAccount() {
    balance = 0;
  }

  /**
   * Constructs a bank account with a given balance.
   * @param initialBalance the initial balance
   */
  public BankAccount(double initialBalance) {
    balance = initialBalance;
  }

  /**
   * Returns the current balance.
   */
  public double getBalance() {
    return balance;
  }

  /**
   * Deposits money into the account.
   * @param amount the amount to deposit
   */
  public void deposit(double amount) {
    balance += amount;
  }

  /**
   * Withdraws money from the account.
   * @param amount the amount to withdraw
   */
  public void withdraw(double amount) {
    if (amount <= balance) {
      balance -= amount;
    } else {
      System.out.println("Insufficient funds.");
    }
  }
}

3
Deposits money into the bank account.
@param amount the amount to deposit
*/
public void deposit(double amount)
{
    double newBalance = balance + amount;
    balance = newBalance;
}

/**
   Withdraws money from the bank account.
   @param amount the amount to withdraw
*/
public void withdraw(double amount)
{
    double newBalance = balance - amount;
    balance = newBalance;
}

/**
   Gets the current balance of the bank account.
   @return the current balance
*/
public double getBalance()
{
    return balance;
}

4 Testing a Class

The following code illustrates how the implemented BankAccount class functionality can be tested. This is done by using another class which creates BankAccount objects and calls methods on them.

/**
   A class to test the BankAccount class.
*/
public class BankAccountTester {

    /**
       Tests the methods of the BankAccount class.
       @param args not used
    */
    public static void main(String[] args) {
        BankAccount harrysChecking = new BankAccount();
        harrysChecking.deposit(2000);
        harrysChecking.withdraw(500);
        System.out.println(harrysChecking.getBalance());
    }
}
BankAccount harrysSavings = new BankAccount(100);
harrysSavings.withdraw(30);
harrysSavings.withdraw(10);
harrysSavings.deposit(20);
    double balance = harrysSavings.getBalance();
    System.out.println("Savings account balance: " + balance);
}

When the above program is run, it displays:

1500.0
Savings account balance: 80.0

5 Categories of Variables - Initialisation

There are four types of variables according to where in the code they are encountered:

- **Instance fields**: these belong to an object (e.g. the `balance` variable in class `BankAccount`). They cannot exist outside an object. The instance fields stay alive until no method uses the object any longer (in which case the object become a candidate for recycling by the garbage collector).

  If an instance field is not initialised then:
  - boolean variables are initialised to `false`.
  - Other primitive types are initialised to `0`.
  - Objects are initialised to `null`.

- **Static fields**: these belong to a class. They are going to be described in a later lecture.

- **Local variables**: these belong to a method. They go out of scope (seize to exist and they cannot be used) when the method exits. For example, `newBalance` in the `deposit()` method of class `BankAccount`.

  Local variables must be initialised explicitly in the code (otherwise they do not receive a default initialisation value).

- **Parameter variables**: these belong to a method and they are declared in the argument list of the method. They go out of scope (seize to exist and they cannot be used) when the method exits.

  Parameter variables are initialised with the value (for primitives) or address (for objects) of the actual argument which is passed to them.
6 The Java API Documentation

The Java API (Application Programming Interface) lists every class and method in the Java library.

- It can be found in:
  
  http://java.sun.com/j2se/1.5.0/docs/api/index.html

- Familiarisation is needed with the API usage, as nobody can remember the details of all the classes.

7 The Javadoc utility

The `javadoc` utility automatically generates documentation in HTML format, for classes, their methods and fields.

- To make this possible, comments in implemented classes must follow a special format, as in the `BankAccount` class example:
  
  - Each `/** ... */` comment followed by a class, method or field, will be used to produce the HTML documentation.
  - Special tags such as `@author`, `@param`, `@return`, `throws` must be placed within the `javadoc` comment, in order to document an appropriate feature (such as who is the author, what are the parameters of the method, what the method returns and what exceptions are thrown).

- To run the `javadoc` utility:

  
  javadoc ClassName.java

  A list of HTML files will be generated, which can be viewed using a HTML browser.

8 String vs StringBuffer

The `String` class is immutable, i.e. the contents of a `String` object cannot be modified (it is not possible to change, remove or add a character)!

There are two solutions to this problem:

1. Create a new `String` object which contains the desired characters:

   ```java
   String s1 = "A mispeled word";
   s1 = "A misspelled word"; // creates a new String object and assign to s1
   ```

2. Use the mutable `StringBuffer` class to create a `StringBuffer` object. Modify the contents (characters of the object by calling an appropriate method:

   ```java
   StringBuffer s2 = new StringBuffer("A mispeled word");
   s2.insert(4, 's'); // insert an 's' at position 4
   s2.insert(9, 'l'); // insert an 'l' at position 9
   ```
Objects are using “Call by Reference”

Objects are passed by reference to methods, while primitive types are passed by value. This is shown in the program below.

```java
public class CallingMechanismTester {
    public void increase(int i) {
        i++;
    }

    public void modify(StringBuffer stb) {
        stb.append(" there!");
    }

    public static void main(String[] args) {
        int i = 10;
        // create an object before calling an instance method on it
        CallingMechanismTester tester = new CallingMechanismTester();

        tester.increase(i);
        System.out.println("After calling increase(), i is: "+ i);

        StringBuffer s = new StringBuffer("Hello");
        tester.modify(s);
        System.out.println("After calling modify(), s is: "+ s);
    }
}
```

When the above code is run, it displays:

```
After calling increase(), i is: 10
After calling modify(), s is: Hello there!
```