Object Oriented Programming

Java

Part 3: Methods

Methods (1)

Sintax

```
Modifier* Type Id ( [ TypeP IdP [, TypeP IdP]* ] ) {
        [ Local_variable | Statement ]*
   }
```

- Modifier: modifier (visibility, among others)
- Type: return type of the method
- Id: method name
- TypeP: type of the parameters of the method
- IdP: name of the parameters of the method
- { [Local_variable | Statement]* }: body of the method

Methods (2)

- **Modifiers**:
  - Visibility:
    - **public**: accessible anywhere the class is accessible.
    - **private**: accessible only in the class itself.
    - **protected**: accessible in subclasses of the class, in classes in the same package and in the class itself.
  - **abstract**: method without body.
  - **static**: class method, invoked on behalf of the entire class.
  - **final**: cannot be overridden in subclasses.

Methods (3)

- If the visibility is omitted, the method is accessible in the class and in classes in the same package only.
- With the exception of visibility modifiers, a method may have more than one modifier. However, it cannot be at the same time **abstract** and **final**.
- A static method can only access static fields and static methods.

Methods (4)

- The return type of a method is mandatory and it can be:
  - primitive type (`boolean`, `char`, `byte`, `short`, `int`, `long`, `float` and `double`)
  - references (classes and interfaced defined by Java, for instance, class `String`, and classes and interfaced defined by the programmer)
  - **`void`**
- The result is returned to the caller by the **`return`** statement.

Methods (5)

- A method might have zero, one, or more parameters:
  - Possible types for the parameters:
    - primitive types (`boolean`, `char`, `byte`, `short`, `int`, `long`, `float` and `double`)
    - references (classes and interfaced defined by Java, for instance, class `String`, and classes and interfaced defined by the programmer)

Methods (6)

- In Java, all parameter methods are **passed by value**:
  - The values of the parameters are always copies of the values passed by argument.
  - The method can change its parameters' value without affecting values in the code that invoked the method.
  - When the parameter is an object reference, it is the object reference, not the object itself, that is passed by value.
    - One can change which object a parameter refers to inside the method without affecting the reference that was passed.
    - But if one change any fields of the object or invoke methods that change the object's state, the object is changed for every part of the program that holds a reference to it.
  - A parameter can be defined as **final**, meaning that the value of the parameter will not change while the method is executing.
    - When the parameter is a reference, the final only applies to the reference, and not the object itself.

Methods (9)

```
public class Demo {
   static void foo(int arg1) {
      arg1 = 10;
   }
   public static void main(String args[]){
      int arg = 5;
      foo(arg);
      System.out.println("arg = " + arg);
   }
}
```

In the terminal it is printed `arg = 5`

Methods (10)

```
public class Xpto {
   public int var = 1;
}
```

```
public class Demo {
   public static void foo(Xpto x) {
      x.var += 10;
   }
   public static void main(String args[]){
      Xpto arg = new Xpto();
      foo(arg);
      System.out.println(“arg.var = ” + arg.var);
   }
}
```

In the terminal it is printed `arg.var = 11`

Methods (11)

```
public static void printAnimals(final int i, final String[] s) {
   for (; i<s.length; i++)
       System.out.println(s[i]);
}
```

```
final parameter i may not be assigned
   for (; i<strings.length; i++)
1 error
```

```
public static void printAnimals(final int i, final String[] s) {
   s[i]="Snake";
   for (int j=i; j<s.length; j++)
       System.out.println(s[j]);
}
```

```
printAnimals(1,new String[]{"Lion","Tiger”,"Bear"});
```

prints in the terminal

```
Snake
Lion
```

Methods (12)

- A method is invoked with the dot operator (“`.`”) via references in the form `reference.method(params)`.
- The `reference` is an identifier of:
  - an object, in a non-static method.
  - a class, in a **static** method.

Methods (13)

- Sequential invocation:
  - A method may return an object, from which another method can be invoked. There are two ways to perform sequential invocation:
    1. Store the object in a variable to invoke the method from it.

```
Classe var = obj.method1();
var.method2();
```

    2. Invoke it directly.

```
obj.method1().method2();
```

Methods (14)

- Inside a non-static method, the object on which the method was invoked is referenced by **`this`**.
- There is no `this` reference inside static methods.
- Usually, the `this` reference is used to pass the object on which the method was called as argument to other methods.

Methods (15)

```
public void deposit(float value){
   balance += value;
}
```

```
public void deposit(float value){
   this.balance += value;
}
```

- The `this` reference is needed in the presence of a **field hidden** by a local variable or a parameter.

```
public void deposit(float balance){
   this.balance += balance;
}
```

Methods (16)

- The signature of a method is given by its name and number and type of its parameters.
- Two methods can have the same name if they have different number or type of its parameters (and thus different signatures).
- This feature is called **overloading**.
- The compiler uses the number and type of the arguments to find the best match from the available overloads.
- Overloading is typically used when a method (or constructor) can accept the same information presented in different forms, or when it can use some parameters with default values (and so they are not needed to be supplied).

The `main` method (1)

- The JVM interpreter executed always the **`main` method** of the class indicated in the command line:
  - Modifiers: **public static**
  - Return: **void**
  - Parameters: **String[] args**
- An application can have any number of `main` methods (because each class in an application can have one). The `main` method to execute is specified each time the program is run.

The `main` method (2)

```
Class Echo {
   public static void main(String[] args) {
      for(int i=0; i<args.length; i++)
         System.out.print(args[i]+ " ");
      System.out.println();
   }
}
```

```
> javac Echo.java
> java Echo I am here
> I am here
```

The `main` method (3)

- The program terminates when it is executed the last instruction in the `main`.
- If one needs to anticipate the termination, the following method should be invoked:

  System.exit(int status);

  where the parameter `status` identifies the terminating code (success - 0 in Linux, 1 in Windows).

Local variables

- Declaring a local variable is similar to declaring a field, but **final** is the only modifier applicable.

  Modifier* Type Id [= Expr] [, Id = Expr]* ;
- Local variables need to be initialized before being used, in their declaration or before they are used.
  - Unlike fields, there are no default initialization of local variables.
- Like fields, when the initialization of a local variable declared as **final** is not made in its declaration, this local variable is termed **blank final**.

Statements (1)

- A **block** groups zero or more statements.
- A block is delimited by the parenthesis `{` and `}`.
- A local variable exists only as long as the block containing its declaration is executing.
- A local variable can be declared in any point inside a block, but always before being used.

Statements (2)

Assignement

Var = Expr [, Var = Expr]*;

- Var: local variable
- Expr: expression

- Recall that:
  - The assignment Var = Var op Expr is equivalent to Var op= Expr.

Statements (3)

- The assignment of references provides two distinct references to the same object.

```
Account c1 = new Account(), c2;
c1.deposit (1000);
c2 = c1; // c2 and c1 are references to the same object
System.out.println("Balance of c1 = " + c1.balance());
System.out.println("Balance of c2 = " + c2.balance());
c1.deposit(100);
c2.deposit(200);
System.out.println("Balance of c1 = " + c1.balance());
System.out.println("Balance of c2 = " + c2.balance());
```

In the terminal is printed

```
Balance of c1 = 1000
balance of c2 = 1000
Balance of c1 = 1300
Balance of c2 = 1300
```

Statements (4)

Conditional execution

`if` (Expr-Bool) statement1 [`else` statement2]

- Expr_Bool: Boolean expression

Statements (5)

```
char c;
/* identifies char category */
if (c>=‘0’ && c<=‘9’)
   System.out.println(”Digit!");
else if ((c>=‘a’ && c<=‘z’) || (c>=‘A’ && c<=‘Z’))
   System.out.println(“Character!”);
else
   System.out.println(”Other!");
```

Statements (6)

Value selection

```
switch (Expr) {
        case literal: statement1
        case literal: statement2
        ...
        default: statementN
    }
```

- Value of the expression Expr (`char`, `byte`, `short` or `int`, or corresponding wrapper class, or enum) is compared with literals.
- The **`break`** statement is use to terminate the processing of a particular case within the switch statement.

Statements (7)

```
int i = 3;
switch (i) {
    case 3: System.out.print("3, ");
    case 2: System.out.print("2, ");
    case 1: System.out.print("1, ");
    case 0: System.out.println("Boom!");
    default: System.out.println(”A number, please!");
}
```

In the terminal is printed

```
3, 2, 1, Boom!
A number, please!
```

Statements (8)

```
int i = 3;
switch (i) {
    case 3: System.out.print("3, ");
    case 2: System.out.print("2, ");
    case 1: System.out.print("1, "); break;
    case 0: System.out.println("Boom!");
    default: System.out.println(”A number, please!");
}
```

In the terminal is printed `3, 2, 1,`

Statements (9)

```
int i = 4;
switch (i) {
    case 3: System.out.print("3, ");
    case 2: System.out.print("2, ");
    case 1: System.out.print("1, ");
    case 0: System.out.println("Boom!");
    default: System.out.println(”A number, please!");
}
```

In the terminal is printed `A number, please!`

Statements (10)

Conditional cycle

`while` (Expr-Bool) body-statements

- The body-statements are executed while the Expr-Bool is evaluated to `true`.

`do` body-statements `while` (Expr-Bool)

- The test may be executed only after body-statements are executed.
- In both conditional cycles:
  - Inside the body-statements the program may transfer control to the evaluation of Expr-Bool with **`continue`**.
  - A **`break`** may be used to exit from the cycle.

Statements (11)

Iterative cycle

`for` (initialization; Expr-Bool; update)

  body-statements

- The for loop is used to iterate a variable over a range of values until some logical end to that range is reached.
- The initialization is executed before entering the cycle.
- If the Boolean expression Expr-Bool is evaluated to `true` the body-statements are executed.
- After executing body-statements the update is executed and then the loop-expression Expr-Bool is reevaluated.
- The cycle repeated until Expr-Bool is found to be `false`.

Statements (12)

- Loop variables may be declared directly in the initialization.
- The initialization and update parts of the loop may be comma-separated list of expressions.
- All the expressions in the for construct are optional.

  By default, the Expr-Bool expression is evaluated `true`.

  –The statements **`for(;;)`** and **`while(true)`** are equivalent.
- Inside the body-statements the program may transfer control to the evaluation the update expression followed by the evaluation of Expr-Bool with **`continue`**.
- A **`break`** may be used to exit from the cycle.

Statements (13)

```
/* Print even numbers until 20 */
for(int i=0, j=0; i+j<=20; i++, j++)
   System.out.print(i+j + " ");
System.out.println();
```

In the terminal is printed

```
0 2 4 6 8 10 12 14 16 18 20
```

Statements (14)

Iterative cycle (variant): ***for-each loop***

`for` (Type loop-var: set-expr) body-statements

- The set-expr must evaluate to an object that defines a set of values (an array or an object that implements the `java.lang.Iterable` interface, as collections provided by Java) over which we intend to iterate through.
- Each time though the loop loop-var takes on the next value from the set, and body-statements are executed.
- This continues until no more values remain in the set.

Statements (15)

```
static double mean(int[] values) {
   double sum= 0.0;
   for (int val : values)
      sum+= val;
   return sum/ values;
}
```

```
int[] values={20,19,18,17,16,15,14,13,12};
System.out.println(”The mean is" + mean(values));
```

In the terminal is printed `The mean is 16.0`

Statements (16)

- The `break` statement may be used to exit from any block.
- There are two forms of the `break` statement:
  - **Unlabeled**: **`break;`**
  - **Labeled**: **`break` label`;`**
- An unlabeled `break` terminates the innermost `switch`, `for`, `while` or `do` statements, and so it can only be used in those contexts.
- A labeled `break` can terminate any labeled statement.

Statements (17)

```java
public boolean updateValue(float[][] matrix, float val) {
   int i, j;
   boolean found = false;
findval:
   for (i=0; i<matrix.length; i++) {
      for (j=0; j<matrix[i].length; j++) {
         if (matrix[i][j]==val) {
            found = true;
            break findval;
         }
      }
   }
   if (!found)
      return false;
   //update matrix[i][j] as wished
   return true;
}
```

Statements (18)

```
public boolean updateValue(float[][] matrix, float val) {
   int i, j;
findval:
   {
      for (i=0; i<matrix.length; i++) {
         for (j=0; j<matrix[i].length; j++) {
            if (matrix[i][j]==val)
               break findval;
         }
      }
      //if we reach this then we have not found val
      return false;
   }
   //update matrix[i][j] as wished
   return true;
}
```

Statements (19)

- The **`continue`** statement can be used only within a loop (`for`, `while` or `do`) and transfers control to the end of the loop's body to continue on with the loop.
- This causes the following expression to be the next thing evaluated:
  - The Expr-Bool, in the case of `while` and `do`.
  - The update followed by Expr-Bool, in the case of a basic `for`.
  - The next value in the set of elements, if there is one, in the case of an enhanced for-each loop.

Statements (20)

- The `continue` statement also has two forms:
  - **Unlabeled: `continue;`**
  - **Labeled: `continue` label`;`**
- In the unlabeled form, `continue` transfers control to the end of the innermost loop's body.
- The labeled form transfers control to the end of the loop with the label. The label must belong to a loop statement.

Statements (21)

```
static void duplicateSymmMatrix(int[][] matrix) {
   int dim = matrix.length;
column:
   for (int i=0; i<dim; i++) {
      for (int j=0; j<dim; j++) {
         matrix[i][j]=matrix[j][i]=matrix[i][j]*2;
         if (i==j)
            continue column;
      }
   }
}
```

Statements (22)

- A **`return`** statement terminated execution of a method and returns to the invoker.
  - If the method returns no value (`void`), use simply `return;`
  - If the method has a return type, the `return` must include an expression of a type that could be assigned to the return type.
- A `return` may also be used to exit a constructor; in this case, as constructors have no return type, use only `return;`