Lecture - 06

Declarations: Classes

Arpit Rana

8th Feb 2022

Class Declarations

A class declaration introduces a new reference type.

```
class_modifiers class class_name
                    extends_clause
                    implements_clause // Class header
{ // Class body
    field_declarations
    method_declarations
    constructor_declarations
}
```

- Class header: it can specify the following information
  - An *accessibility modifier* (e.g. `public`)
  - Additional *class modifier* (e.g. `abstract`, `final`)
  - Any class it *extends*
  - Any interface it *implements*

Class Declarations

```
class_modifiers class class_name
                        extends_clause
                        implements_clause // Class header
{ // Class body
      field_declarations
      method_declarations
      constructor_declarations
}
```

- Class body (enclosed in { }): it can contain member declarations
  - Field declarations
  - Method declarations
- Members declared as static belong to the class (called *static* members), while non-static members belong to the objects of the class (called *instance* members).

Class Declarations

```
class_modifiers class class_name
                        extends_clause
                        implements_clause // Class header
{ // Class body
    field_declarations
    method_declarations
    constructor_declarations
}
```

- Class body (enclosed in { }): it can also include the following declarations
  - Constructor declarations
- In the class syntax, the only mandatory parts are the keyword `class`, the *class name*, and the class body braces ({ }).

Class Declarations

***static* and *non-static* context:**

- A *static* context is defined by static methods, static field initializers, and static initializer blocks.
- A *non-static* context is defined by instance methods, non-static field initializers, instance initializer blocks, and constructors.
- One crucial difference between the two contexts is that the static code can refer only to other static members.

Method Declarations

The simplified declaration of methods is as follows.

```
method_modifiers return_type method_name
                    (formal_parameter_list) throws_clause // Method header
{ // Method body
      local_variable_declarations
      statements
}
```

Method header: It specifies the following information (except method name) -

- Scope or *accessibility modifier*
- Additional *method modifiers*
- The *type* of the return value or ***`void`*** if the method does not return a value
- A *formal parameter list*
- Any *exceptions* thrown by the method (specified in *throws* clause)

Method Declarations

The *formal parameter list* is a comma-separated list of parameters for passing information to the method through a *method call*.

Each parameter is a simple variable declaration consisting of its *type* and *name*:

optional_parameter_modifier type parameter_name

- The parameter names are local to the method. The *optional parameter modifier* – `final`
- Recommended to use *@param* tag in a Javadoc comment to document the formal parameters of a method.
- The *signature* of the method comprises the method name and the type of the formal parameters only.
- The method body is a block containing the local variable declarations and the statements of the method. They can be *instance* or *static*.

Statements

Statements in Java can be grouped into various categories.

- Declaration statements
- Control flow statements
- Expression statements
    - Assignments
    - Increment and decrement operators
    - Method calls
    - Object creation expressions with the new operator
- Block ({ }), is a compound statement that can be used to group zero or more local declarations and statements.

Instance Methods and the Object Reference `this`

Instance methods belong to every object of the class and can be invoked only on objects.

- All members defined in the class, both static and non-static, are accessible in the context of an instance method.
- The object on which the method is being invoked (current object) can be referenced in the body of the instance method by the keyword **`this`**.
  - The **`this`** reference can be used to access members of the object.
  - The **`this`** reference can not be modified as it is a final reference.
  - If a method needs to pass the current object to another method, it can do so using the **`this`** reference.
  - The **`this`** reference cannot be used in static context.

Instance Methods and the Object Reference `this`

```
public class Light {
  // Fields:
  int     noOfWatts;      // Wattage
  boolean indicator;      // On or off
  String  location;       // Placement

  // Constructor
  public Light(int noOfWatts, boolean indicator, String site) {
    String location;

    this.noOfWatts = noOfWatts;   // (1) Assignment to field
    indicator = indicator;        // (2) Assignment to parameter
    location = site;              // (3) Assignment to local variable
    this.superfluous();           // (4)
    superfluous();                // equivalent to call at (4)
  }

  public void superfluous() {
    System.out.printf("Current object: %s%n", this); // (5)
  }

  public static void main(String[] args) {
    Light light = new Light(100, true, "loft");
    System.out.println("No. of watts: " + light.noOfWatts);
    System.out.println("Indicator:    " + light.indicator);
    System.out.println("Location:     " + light.location);
  }
}
```

Method Overloading

- Each method has a signature which comprises -
  - The name of the method
  - The types and the order of the parameters in the formal parameter list
- Several method implementations may have the same name, as long as the method signature differ. This practice is called ***method overloading***.
  - e.g. the class `java.lang.Math` contains an overloaded method `min()`, which returns the minimum of two numeric values.

```
public static double min(double a, double b)
public static float min(float a, float b)
public static int min(int a, int b)
public static long min(long a, long b)
```

Method Overloading

- Only methods declared in the same class and those that are inherited by the class can be overloaded.
- At the compile time, the right implementation of an overloaded method is chosen based on the signature of the method call.

Constructors

The main purpose of constructors is to set the initial state of an object, when the object is created by using the **`new`** operator.

```
accessibility_modifier class_name (formal_parameter_list)
                                   throws_clause // Constructor header
{ // Constructor body
    local_variable_declarations
    statements
}
```

Constructors

There are some restrictions on constructors unlike method declarations -

- Modifiers other than an accessibility modifier are not permitted in the constructor header.
- Constructors cannot return a value, and, therefore, do not specify a return type, not even void, in the constructor header.
- The constructor name must be the same as the class name.

Constructors

The main purpose of constructors is to set the initial state of an object, when the object is created by using the **`new`** operator.

```
public class Name {

  Name() {                          // (1) No-argument constructor
    System.out.println("Constructor");
  }

  void Name() {                     // (2) Instance method
    System.out.println("Method");
  }

  public static void main(String[] args) {
    new Name().Name();              // (3) Constructor call followed by method
call
  }
}
```

The Default Constructor

If a class does not specify any constructors, then a default constructor is generated for the class by the compiler.

The default constructor is equivalent to the following implementation:

```
class_name() { super(); }    // No parameters. Calls
superclass constructor.
```

- A default constructor is a no-argument constructor.
- The only action taken by the default constructor is to call the superclass constructor.
- Also, all instance variables in the object are set to the default value of their type.

Overloaded Constructors

Like methods, constructors can be overloaded.

Since the constructors in a class all have the same name as the class, their signatures are differentiated by their parameter lists.

```
class Light {
  // …
  // No-argument constructor:
  Light() {                                                    // (1)
    noOfWatts = 50;
    indicator = true;
    location  = "X";
  }

  // Non-zero argument constructor:
  Light(int noOfWatts, boolean indicator, String location) { // (2)
    this.noOfWatts = noOfWatts;
    this.indicator = indicator;
    this.location  = location;
  }
  //…
}

class Greenhouse {
  // …
  Light moreLight  = new Light(100, true, "Greenhouse");      // (3) OK
  Light firstLight = new Light();                             // (4) OK
}
```

Next lecture -
Declarations: Arrays