Object Oriented Programming

Java

Part 2: Classes and objects

Classes (1)

Sintax

Modifier* class Ident

[extends IdentC] [implements IdentI [,IdentI]*] {

[Fields | Methods]*

}

- Modifier: modifier (visibility, among others)
- Ident: class name
- extends IdentC: specialization of the superclass
- implements IdentI: implementation of interfaces

Classes (2)

- **Class modifiers**:
  - **public**: a public class is publicly accessible
    - Anyone can declare references to objects of the class or access its public members.
  - **abstract**: an abstract class is considered incomplete and no instances of the class may be created.
  - **final**: a final class cannot be subclassed.
- Without the modifier **public**, a class is only accessible within its own package.
- A class declaration can be preceded by several modifiers. However, cannot be both **abstract** and **final**.

Classes (3)

```
public class Account{
        /* fields */
        /* methods */
}
```

Fields (1)

Sintax

Modifier* Type Ident [= Expr] [, Ident = Expr]* ;

- Modifier: modifier (visibility, among others)
- Ident: field name
- Type: field type
- Expr: field initialization

Fields (2)

- Field possible types:
  - Primitives:
    - `boolean`
    - `char`
    - `byte`
    - `short`
    - `int`
    - `long`
    - `float`
    - `double`
  - References: classes and interfaces defined by Java, for instance, the class `String`, and classes and interfaces defined by the programmer.

Fields (3)

- **Field 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.
  - **static**: class variable.
  - **final**: constant field, whose value cannot be changed after initialized.
  - **transient**: field which is not going to be serialized.
- If visibility is omitted, the field is accessible in the classes of the same package.
- With the exception of visibility modifiers, a field may contain more than one modifier.

Fields (4)

- **Principle of encapsulation and data hiding**:
  - Fields should not be accessible outside the object where they belong; instead, they should only be updated via methods (setters).
  - Fields' visibility should be **private** or **protected** (or package). The **public** modifier should be avoided.

Fields (5)

- **Initialization**:
  - Expr might be a constant, another field, a call to a method, or an expression combing those.
  - By default, a newly created object is given an initial state, where the fields are initialized with their default values depending on their types:
    - **boolean** – false
    - **char** – '\u0000'
    - **byte**, **short**, **int**, **long** – 0
    - **float**, **double** – +0.0
    - Reference to an object – null
  - A field can be (explicitly) initialized:
    - Directly when it is declared in the class.
    - When the class is loaded to the JVM (in the case of **static** fields), or in the creation of the object (in the case of non-static fields).

Fields (6)

- A constant has the modifiers **static final**.

```
public static final double PI = 3.141592;
```

- A **final** field needs to be explicitly initialized. One that does not have an initializer is termed **blank final**.
  - Blank finals are used when simple initialization is not appropriate.
  - Blank finals must be initialized once the class has been loaded to the JVM (in the case of a static field) or once an object has been fully constructed (for non-static fields).
  - The compiler will ensure that this is done!

Fields (7)

```
public class Account {
        /* fields */
        private static long nbNextAccount = 0;
        protected long nbAccount; // account number
        protected String owner;   // account owner
        protected float balance;  // actual balance
        /* methods */
}
```

Fields (8)

- A field in a class is accessed via the dot operator (“`.`”) in the form `reference.field`.
- The `reference` is an ideintifier of:
  - an object, for a non-static field.
  - a class, for a **static** field.

```
System.out.println(Account.nbNextAccount);
```

Objects

Sintax

Ident = new IdentClass ([Expr [, Expr]*]);

- Ident: reference to the new object
- IdentClass: type of the objet to create
- Expr: constructor parameters

Garbage collector

- In Java, an object is created with the **new** operator, but the object is never deleted explicitly.
- The **garbage collector** manages memory and objects that cannot be used any longer have their space automatically reclaimed without programmer intervention.
- If the programmer no longer needs an object it should cease referring to it:
  1. With local variables in methods this can be as simple as returning from the method.
  2. More durable variables, such as object fields, must be set to null.

Constructors (1)

- A **constructor** is a block of statements that are executed to initialize an object before the reference to it is returned by `new`.
  - They have the same name as the class.
  - Like methods, they take zero or more arguments.
  - Unlike methods, they have no return type, not even `void`.
  - They are commonly used to initialize the values of the fields, when more than simple initialization is needed.
- A class may contain more than one constructor.
  - The type and the number of arguments being passed to the constructor determine the constructor to be used.

Constructors (2)

```
public class Account {
        /* Fields */
        private static long nbNextAccount = 0;
        protected long nbAccount; // account number
        protected String owner;   // account owner
        protected float balance;  // actual balance
        /* Constructors */
        Account(String s) {
                nbAccount = nbNextAccount++;
                owner = s;
                balance = 100; //minimum amount to open an account
        }
        Account(String s, float q) {
                nbAccount = nbNextAccount++;
                owner = s;
                balance = q;
        }
        /* Methods */
}
```

Constructors (3)

- If no constructor is provided (and only in this case), Java provides a **default no-arg constructor** (no-arg=no arguments).
- A **copy constructor** takes an argument of the current object type and constructs the new object to be a copy of the passed in object.
  - Usually, this is simple a matter of assigning the same values to all fields, but sometimes the semantics of the class dictate more sophisticated actions.

```
/* copy constructor */
Account(Account c) {
        nbAccount = c.nbAccount;
        owner= c.owner;
        balance = c.balance;
}
```

Constructors (4)

- One constructor can invoke another constructor from the same class by using the `this()`. This is called **explicit constructor invocation**.
- If the constructor has `N` parameters, these should be passed to the explicit invocation as `this(param1,...,paramN)`.
- The argument list determines which version of the constructor is invoked.
- If provided, the explicit invocation must be the first statement in the constructor.
- Any expressions that are used as arguments for the explicit constructor invocation must not refer to any fields or methods of the current object.

Constructors (5)

```
public class Account {
        /* Fields */
        private static long nbNextAccount = 0;
        protected long nbAccount; // account number
        protected String owner;   // account owner
        protected float balance;  // actual balance
        /* Constructors */
        Account(String s) {
                this(s,100); //explicit method invocation
        }
        Account(String s, float q) {
                nbAccount = nbNextAccount++;
                owner = s;
                balance = q;
        }
        /* Methods */
}
```

Initialization of non-static fields (1)

- A newly created object is given an initial state:
  - Initialization by default.
  - Initialization when they are declared.
  - When more than a simple initialization is required:
    - **Constructors**: used to initialize an object before the reference to the object is returned by `new`.
    - **Initialization blocks**: executed as if they were placed at the beginning of every constructor in the class.
      - It provides guarantee of correction with blank final fields.

Initialization of non-static fields (2)

- The constructor is invoked after:
  - Initialization, by default, of the non-static fields.
  - Initialization of the non-static fields where they are declared.

Initialization of non-static fields (3)

```
public class Account {
        /* Fields */
        private static long nbNextAccount = 0;
        protected long nbAccount; // account number
        protected String owner;   // account owner
        protected float balance;  // actual balance
        /* Constructors */
        Account(String s) {
                this(s,100); //explicit method invocation
        }
        Account(String s, float q) {
                nbAccount = nbNextAccount++;
                owner = s;
                balance = q;
        }
        /* Methods */
}
```

Initialization of non-static fields (4)

```
public class Account {
        /* Fields */
        private static long nbNextAccount = 0;
        protected long nbAccount; // account number
        /* Initialization block */
        {
                nbAccount = nbNextAccount++;
        }
        protected String owner;   // account owner
        protected float balance;  // actual balance
        /* Constructors */
        Account(String s) {
                this(s,100); //explicit method invocation
        }
        Account(String s, float q) {
                owner = s;
                balance = q;
        }
        /* Methods */
}
```

Initialization of static fields

- The static fields can be initialized:
  - when they are declared.
  - in **static initialization blocks.**
    - Declared as **static**.
    - It can only refer to static members od the class.
- The static initializers are executed after the class is loaded to the JVM, but before it is actually used.