Classes: information hiding & encapsulation

JC08-3.1

(A. Nguyen)

Class

- A class represents a category of objects that share common **characteristics** and **behaviors**; it is a blue print for all objects in that class
- For example,
  - a class many be `BankAccount`
  - an object of that class may be `momsAccount`
- A class has 3 main parts, as documented in the Java API:
  - Instance variables aka fields, to represent the **characteristics**
  - Constructors, to initialize the fields
  - Methods, to represent the **behaviors**

Information hiding – private fields

- A field keeps one piece of information about an object
- A field is declared with a (reserved) word: `public` or `private`:

```
private double balance; // request RAM space & initialize to 0
```

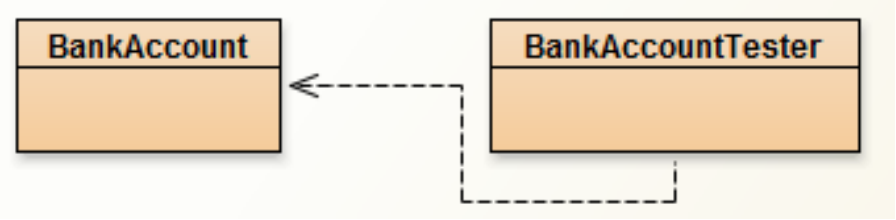
- A field is **often** declared as `private` to "hide" from the client of the class – information hiding – which makes the modification of the information safe

Encapsulation – methods

- A method is a set of instructions to carry out a task and/or return a result
- To be able to use a class, a programmer/you only need to know the method header or method signature (and not how things are done in the body); this is called encapsulation; i.e., a method is a blackbox

Client of a class

- A client of a class A is a class B that uses class A
- BlueJ automatically draws an arrow from the client of a class to that class; e.g.: `BankAccountTester` is a client of class `BankAccount`



Public Interface & Implementing a class

JC08-3.2

(A. Nguyen)

Parts of a class

- To implement a class/method is to write/code it
- A class has 3 main parts, usually presented in this order:
  - Fields
  - Constructors
  - Methods

Parts of a class

```
public class BankAccount
{
  // Fields
  private double balance;
  // Constructors
  public BankAccount()
  {
    // body— not yet coded
  }
  public BankAccount(double initialBalance)
  {
    // body-- not yet coded
  }
  // Methods
  public void deposit(double amount)
  {
    // body-- not yet coded
  }
  public void withdraw(double amount)
  {
    // body-- not yet coded
  }
  public double getBalance()
  {
    // body-- not yet coded
  }
}
```

Naming convention

- A name in Java must not contain blanks; thus, all words are next to each other, and each word starts with an upper-case letter (or the words are "separated" with an underscore); e.g., `unitPrice`
- Names starting with an upper-case letter are class name (and, thus, constructor name); e.g., `Rectangle`
- Names starting with an lower-case letter are variable names and method names; e.g., `setHeight`
- Reserved words are in all lower-case letters
- Names in all upper-case letters are constants (i.e., unmodifiable); e.g., `Color.RED`

The fields

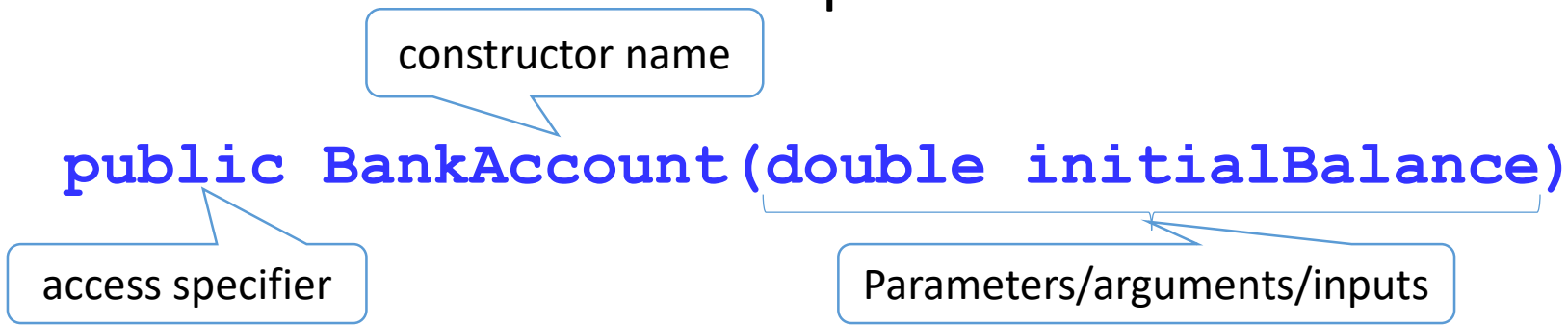
- It is advisable to declare a field `private`, so that it cannot be modify by other classes; e.g.:

```
private double balance; // request RAM space & initialize to 0
```

- A `private` field is accessible by an accessor (whose name usually starts with `get`), and possibly modifiable by a mutator (whose name usually starts with `set`)
- In the absence of the word `private`, the field is `public`
- A field is a kind of variables (instance variable), so it follows the naming rules of variables
- A meaningful field name is a noun

Constructors

- A constructor is a set of statements whose purpose is to initialize the data about an object
- A constructor name is ***always*** the same as its class name (and is capitalized), which is also the same as the name of the source file (i.e., with extension .java)
- A constructor header has 3 parts:



Constructors (cont.)

- When a class has no constructor, Java provides a no-args constructor and set the fields to default values: zero for a numeric value, `null` for an object (i.e., non-existing object), `false` for `boolean`.
- Tip: consult the field names (in the "field" section) to make sure that all fields are initialized in the constructor
- The constructor may be overloaded: same constructor name, with different data types for parameters
- Each parameter is expressed with a data type ***and*** a name; and the parameters are separated by commas (inside the parentheses) – as shown in the API

Constructors: errors

- A constructor does not have a return value or `void`, so do not include it in the signature because doing that would make it a method (i.e., no longer a constructor to be called at `new`):

```
public ~~double~~ BankAccount(double initialBalance)
public ~~void~~ BankAccount(double initialBalance)
```

- Do NOT re-declare a field name in the constructor because re-declaring asks for (different) RAM space for a local variable with the same name as the field:

```
~~double~~ balance = initialBalance;
```

- For a field that is an object, if you forget to create a new object, the default initialized value remains `null` (i.e., non-existing object), which will cause a run-time error when asked to do something

Methods

- A method is a set of Java statements that does a task and/or returns a value
- The programmer who creates a method chooses the method's name (which is usually a verb), starting with a lower-case letter and an upper-case letter at the beginning of each word; e.g., method `getHeight` of class `Rectangle`

Methods (cont.)

- A method is often declared `public`, so that other classes can call it; `public` methods appear in the API
- A method can also be declared `private`, so that only the methods of its own class can call – `private` methods do not appear in the API
- A method signature has 4 main pieces of information:
  - Access specifier: `public` or `private`
  - Return value or `void`
  - Method name
  - Parameters inside the parentheses
- Note: a data type must precede ***each*** parameter name

Methods (cont.)

- A method may have inputs (called parameters) and/or output (called return value)
- The parameters of a method follow the same rules & convention as those of a constructor

Methods (cont.)

- A method with a return value **must** have a `return` statement:

```
public double getBalance()
{
  return balance;
}
```

- A `return` statement, in this case, does two things: (1) quit the method, (2) pass the **one** result (of the same **return type** declared in the method signature) to the calling method
- **Error**: a `return` statement is NOT equivalent to `System.out.println/print` because a print statement simply displays text in the console window & still stays in the current method

Methods (cont.)

- Like a constructor, a method may be overloaded, to do the same task but with different given info: same method name, with different data types for parameters
- Example: overloaded method from the `Rectangle` class:

```
void setBounds(Rectangle r)
void setBounds(int x, int y, int width, int height)
```

Methods (cont.)

- Overloaded methods must have different **data types for parameters**
- The difference in return data type or `void` alone does NOT meet this requirement – see case (1) below
- The difference in the parameter names alone does NOT meet this requirement – see case (2) below

```
public class shape {
    // ...
    public void move (int x, int y) { ... }
    public Point move (int x, int y) { ... }
    // ...
    public Point expand(int x, int y) { ... }
    public Point expand(int h, int v) { ... }
    // ...
}
```

Case 1

Case 2

Documenting the Public Interface

- Documentation comments are written inside /** … */
- For the entire class, place the comments at the beginning of the source file, and use @author and @version
- Example for class BankAccount:

```
/**
 * A bank account has a balance that can be changed by
 * deposits and withdrawals.
 * @author C. Horstmann, modified by A. Nguyen
 * @version v. 1.0
*/
```

Documenting the Public Interface (cont.)

- For a constructor or method, place the comments before it, and use @param and/or @return, where appropriate:

```
/**
   Deposits money into the bank account.
   @param amount the amount to deposit
*/
public void deposit(double amount) {...}
```

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

Tracing technique

- A programmer uses this technique understand what the code does to the values in the method
- Follow the statements in the method, and write/replace the values of the variables

The “this” Reference

JC08-3._

(A. Nguyen)

this

- The `this` reserved word refers to "self". It is used sometimes to **clarify**, and sometimes as a **requirement**.
- When the name of a parameter/input to a method is the same as the name of a field (b/c you cannot think of a good name for the parameter), if there is no `this`, the variable is the parameter:

```
public class Circle
{
   private double radius;
   public Circle(double radius)
   {
      radius = radius; // both vars are the parameter/input ← DON'T DO THIS
      this.radius = radius; // this is correct ← DO THIS
   }
   ...
}
```

this (cont.)

- When the constructors are overloaded (i.e., same name but different signatures), and you want one constructor to call another:

```
public class Fraction
{
   int num;
   int denom;
   public Fraction(int n, int d)
   {
     num = n;
     denom = d;
   }
   public Fraction()
   {
     this(1, 1); // default both values to 1 ← "this" refers to the other constructor Fraction
   }
   ...
}
```

- The same is done when the methods are overloaded.

Static methods & fields

CALLING METHODS:

- A method with **`static`** qualifier belongs to the class, and, hence, is called with the class name as prefix (i.e., it is not necessary to create an object with `new` first): **`x = Math.random();`**
- A non-static method called by a method of different class must be prefixed by an object already created with `new`: **`snoopy.move();`**
- A non-static method called by a method of the same class does not need any prefix, or with `this` prefix; e.g., the `multiply` method of `Fraction` may call `reduce`: **`reduce();`** or **`this.reduce();`**

ACCESSING FIELDS: Similar rules apply

Organizing classes

Separate files for class & main program

File BankAccount.java:

```
public class BankAccount
{
   private double balance;
   ...
   public BankAccount()
   {
      ...
   }
   public void deposit(double amount)
   {
      ...
   }
   ...
} // end of class BankAccount
```

Class header

Fields

Constructors

Methods

Body of class BankAccount: ***three main parts,*** inside a pair of braces

File BankAccountTester.java:

```
public class BankAccountTester
{
   public static void main(String[] args)
   {
      BankAccount bobsAcct = new BankAccount();
      bobsAcct.deposit(200);
      ...
   } // end of main program
}
```

Class header

Main program

Body of class BankAccountTester: ***only main program***, inside a pair of braces

Same file for class & main program

File BankAccount.java:

```
public class BankAccount
{
   private double balance;
   ...
   public BamkAccount()
   {
      ...
   }
   public void deposit(double amount)
   {
      ...
   }
   ...
   public static void main(String[] args)
   {
      BankAccount bobsAcct = new BankAccount();
      bobsAcct.deposit(200);
      ...
   } // end of main program
} // end of class BankAccount
```

Class header

Fields

Constructors

Methods

Main program

Body of class BankAccount: ***four main parts***, inside a pair of braces

Other examples: MovingDisk, BMI

Main program with static methods

File <ClassName>.java:

```
public class <ClassName>
{
   public static...(...)
   {
      ...
   }
   ...

   public static void main(String[] args)
   {
      ...method calls ...
      ...without creating objects ...
   } // end of main program
}
```

Class header

static methods (optional)

Main program

Body of class: ***two main parts***, inside a pair of braces

Note that there is not a real class here. And there may be a collection of ***static*** methods or not.
An example of the simplest main program is `HelloWorld`.
This is not object-oriented programming.

THE END