Fundamentos de programação

Orientação a Objeto
Classes, atributos e métodos

Edson Moreno

edson.moreno@pucrs.br
http://www.inf.pucrs.br/~emoreno

Contents

- Object-Oriented Programming
- Implementing a Simple Class
- Specifying the Public Interface of a Class
- Designing the Data Representation
- Implementing Instance Methods
- Constructors
- Testing a Class
- Problem Solving:
  - Tracing Objects, Patterns for Object Data
- Object References
- Static Variables and Methods

8.1 Object-Oriented Programming

- You have learned structured programming
  - Breaking tasks into subtasks
  - Writing re-usable methods to handle tasks
- We will now study Objects and Classes
  - To build larger and more complex programs
  - To model objects we use in the world

A class describes objects with the same behavior. For example, a Car class describes all passenger vehicles that have a certain capacity and shape.

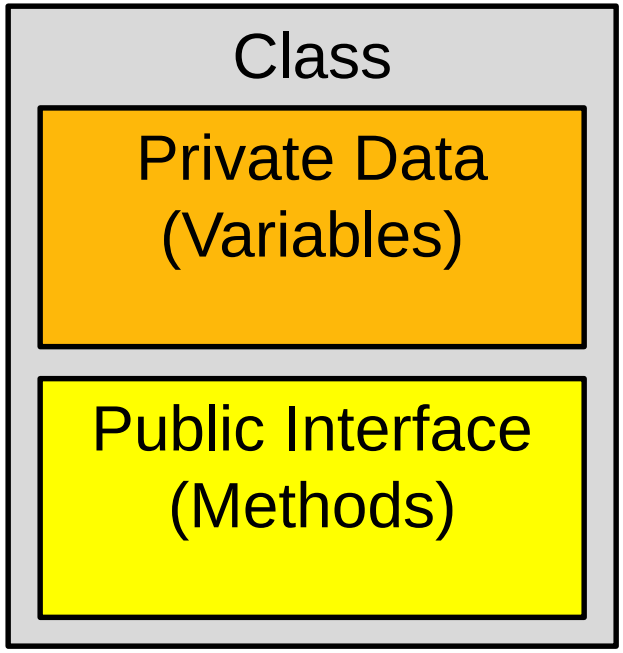
Objects and Programs

- Java programs are made of objects that interact with each other
  - Each object is based on a class
  - A class describes a set of objects with the same behavior
- Each class defines a specific set of methods to use with its objects
  - For example, the `String` class provides methods:
    - Examples: `length()` and `charAt()` methods

```
String greeting = "Hello World";
int len = greeting.length();
char c1 = greeting.charAt(0);
```

Diagram of a Class

- Private Data
  - Each object has its own private data that other objects cannot directly access
  - Methods of the public interface provide access to private data, while hiding implementation details:
  - This is called Encapsulation
- Public Interface
  - Each object has a set of methods available for other objects to use

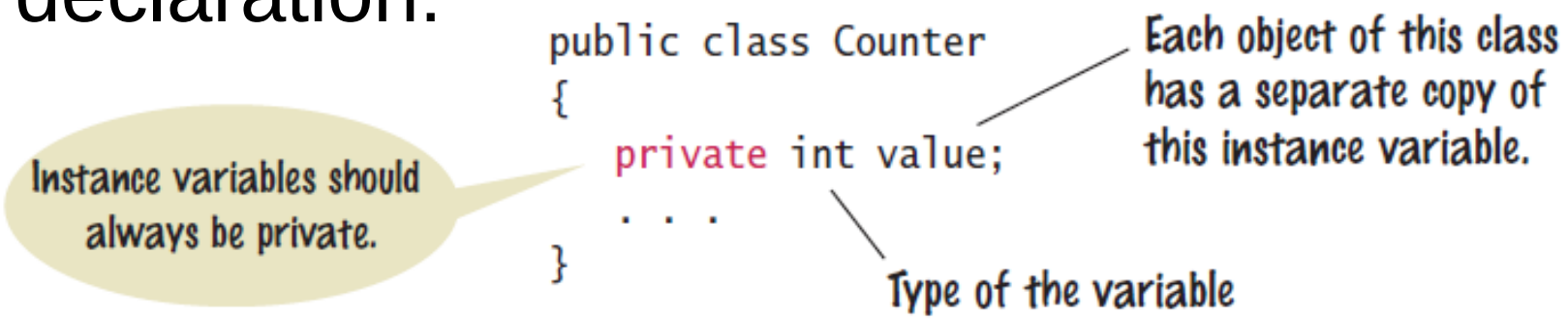


8.2 Implementing a Simple Class

- Example: Tally Counter: A class that models a mechanical device that is used to count people
  - For example, to find out how many people attend a concert or board a bus
- What should it do?
  - Increment the tally
  - Get the current total

Tally Counter Class

- Specify instance variables in the class declaration:



- Each object instantiated from the class has its own set of instance variables
  - Each tally counter has its own current count
- Access Specifiers:
  - Classes (and interface methods) are `public`
  - Instance variables are always `private`

Instantiating Objects

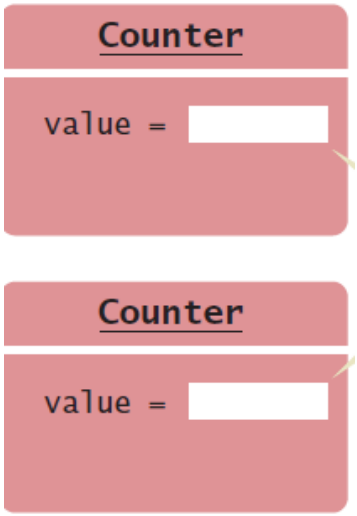
- Objects are created based on classes
  - Use the `new` operator to construct objects
  - Give each object a unique name (like variables)
- You have used the `new` operator before:

```
Scanner in = new Scanner(System.in);
```

- Creating two instances of Counter objects:

Class name Object name Class name

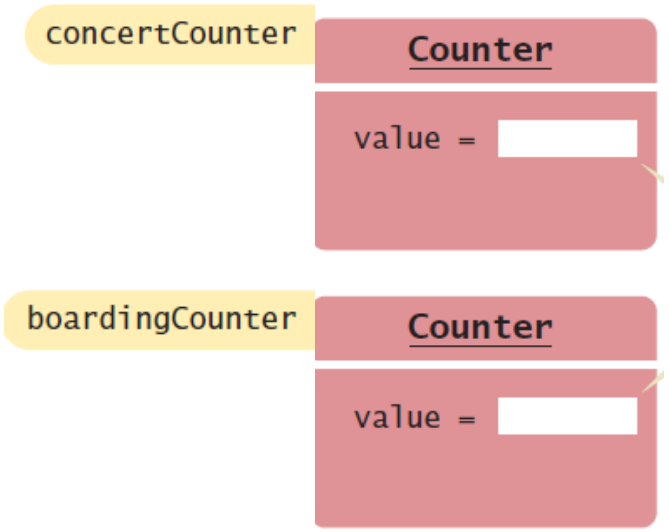
```
Counter concertCounter  = new Counter();
Counter boardingCounter = new Counter();
```



Use the `new` operator to construct objects of a class.

Tally Counter Methods

- Design a method named `count` that adds 1 to the instance variable
- Which instance variable?
  - Use the name of the object
    - `concertCounter.count()`
    - `boardingCounter.count()`




```
public class Counter
{
   private int value;

   public void count()
   {
      value = value + 1;
   }

   public int getValue()
   {
      return value;
   }
}
```

8.3 Public Interface of a Class

- When you design a class, start by specifying the public interface of the new class
  - Example: A Cash Register Class
    - What tasks will this class perform?
    - What methods will you need?
    - What parameters will the methods need to receive?
    - What will the methods return?

Task	Method	Returns
Add the price of an item	`addItem(double)`	void
Get the total amount owed	`getTotal()`	double
Get the count of items purchased	`getCount()`	int
Clear the cash register for a new sale	`clear()`	void

Writing the Public Interface

```
/**
   A simulated cash register that tracks the item count
   and the total amount due.
*/
public class CashRegister
{
   /**
      Adds an item to this cash register.
      @param price: the price of this item
   */
   public void addItem(double price)
   {
      // Method body
   }
   /**
      Gets the price of all items in the current sale.
      @return the total price
   */
   public double getTotal()  ...
```

Javadoc style comments document the class and the behavior of each method

The method declarations make up the *public interface* of the class

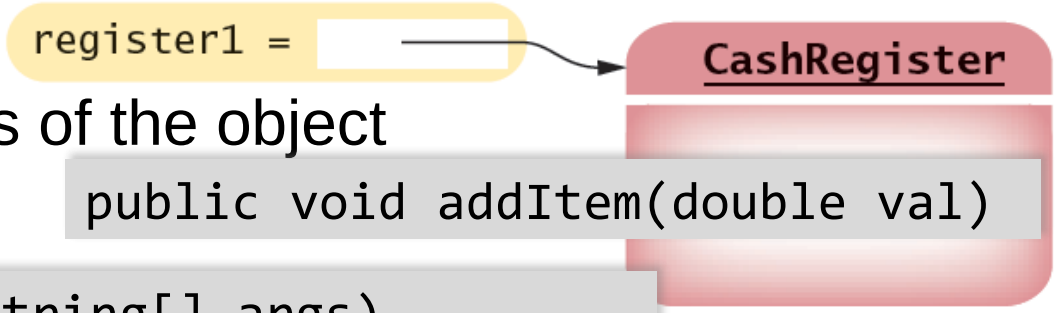
The data and method bodies make up the *private implementation* of the class

Non-static Methods Means…

- We have been writing *class* methods using the `static` modifier:

```
public static void addItem(double val)
```

- For non-static (*instance*) methods, you must instantiate an object of the class before you can invoke methods
  - Then invoke methods of the object




```
public static void main(String[] args)
{
   // Construct a CashRegister object
   CashRegister register1 = new CashRegister();
   // Invoke a non-static method of the object
   register1.addItem(1.95);
}
```

Accessor and Mutator Methods

- Many methods fall into two categories:

1) Accessor Methods: '**get**' methods

- Asks the object for information without changing it
- Normally return a value of some type

```
public double getTotal() {  }
public int getCount() {  }
```

2) Mutator Methods: '**set**' methods

- Changes values in the object
- Usually take a parameter that will change an instance variable
- Normally return void

```
public void addItem(double price) {  }
public void clear() {  }
```

8.4 Designing the Data Representation

- An object stores data in instance variables
  - Variables declared inside the class
  - All methods inside the class have access to them
    - Can change or access them
  - What data will our `CashRegister` methods need?

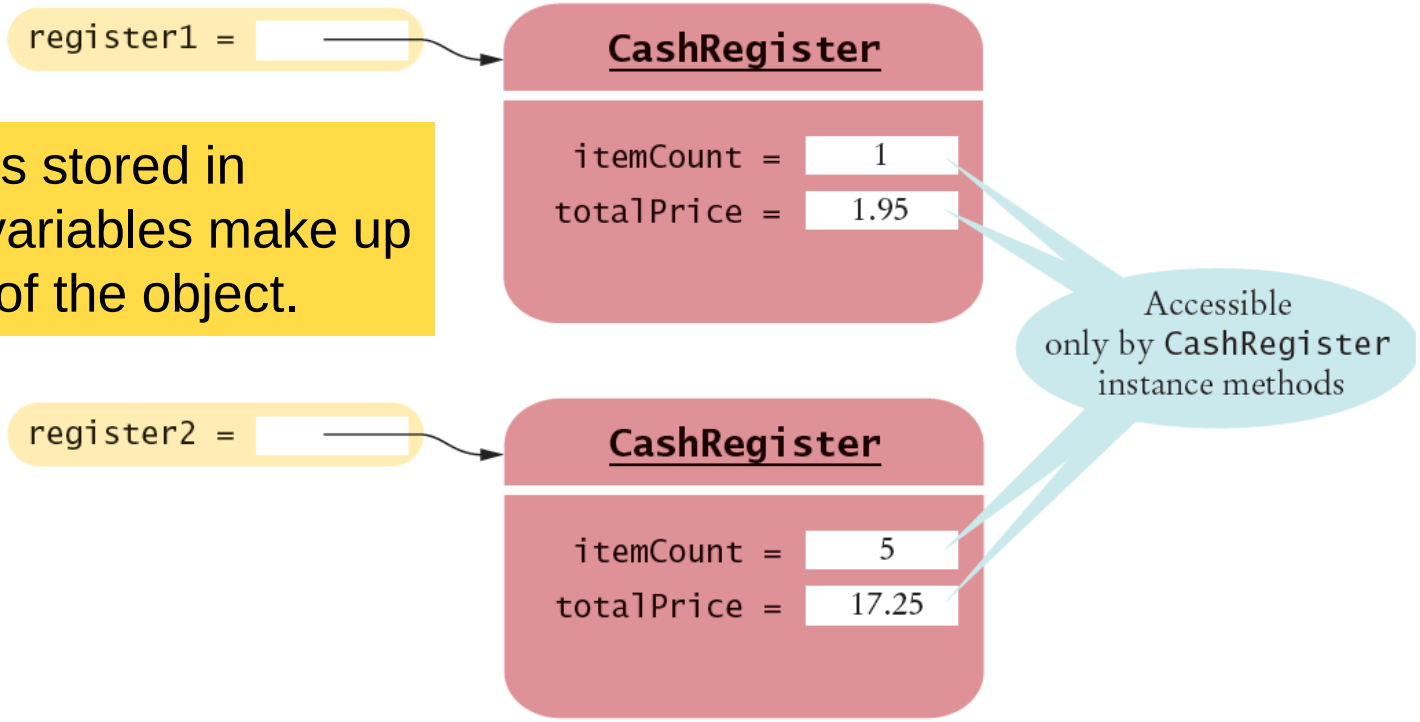
Task	Method	Data Needed
Add the price of an item	`addItem()`	`total, count`
Get the total amount owed	`getTotal()`	`total`
Get the count of items purchased	`getCount()`	`count`
Clear the cash register for a new sale	`clear()`	`total, count`

An object holds instance variables that are accessed by methods

Instance Variables of Objects

- Each object of a class has a separate set of instance variables.

The values stored in instance variables make up the **state** of the object.



Accessing Instance Variables

- `private` instance variables cannot be accessed from methods outside of the class

```
public static void main(String[] args)
{
   . . .
   System.out.println(register1.itemCount); // Error
   . . .
}
```

The compiler will not allow this violation of privacy

- Use accessor methods of the class instead!

```
public static void main(String[] args)
{
   . . .
   System.out.println( register1.getCount() ); // OK
   . . .
}
```

Encapsulation provides a public interface and hides the implementation details.

8.5 Implementing Instance Methods

- Implement instance methods that will use the private instance variables

```
public void addItem(double price)
{
   itemCount++;
   totalPrice = totalPrice + price;
}
```

Task	Method	Returns
Add the price of an item	`addItem(double)`	void
Get the total amount owed	`getTotal()`	double
Get the count of items purchased	`getCount()`	int
Clear the cash register for a new sale	`clear()`	void

Syntax 8.2: Instance Methods

- Use instance variables inside methods of the class
  - There is no need to specify the implicit parameter (name of the object) when using instance variables inside the class
  - Explicit parameters must be listed in the method declaration

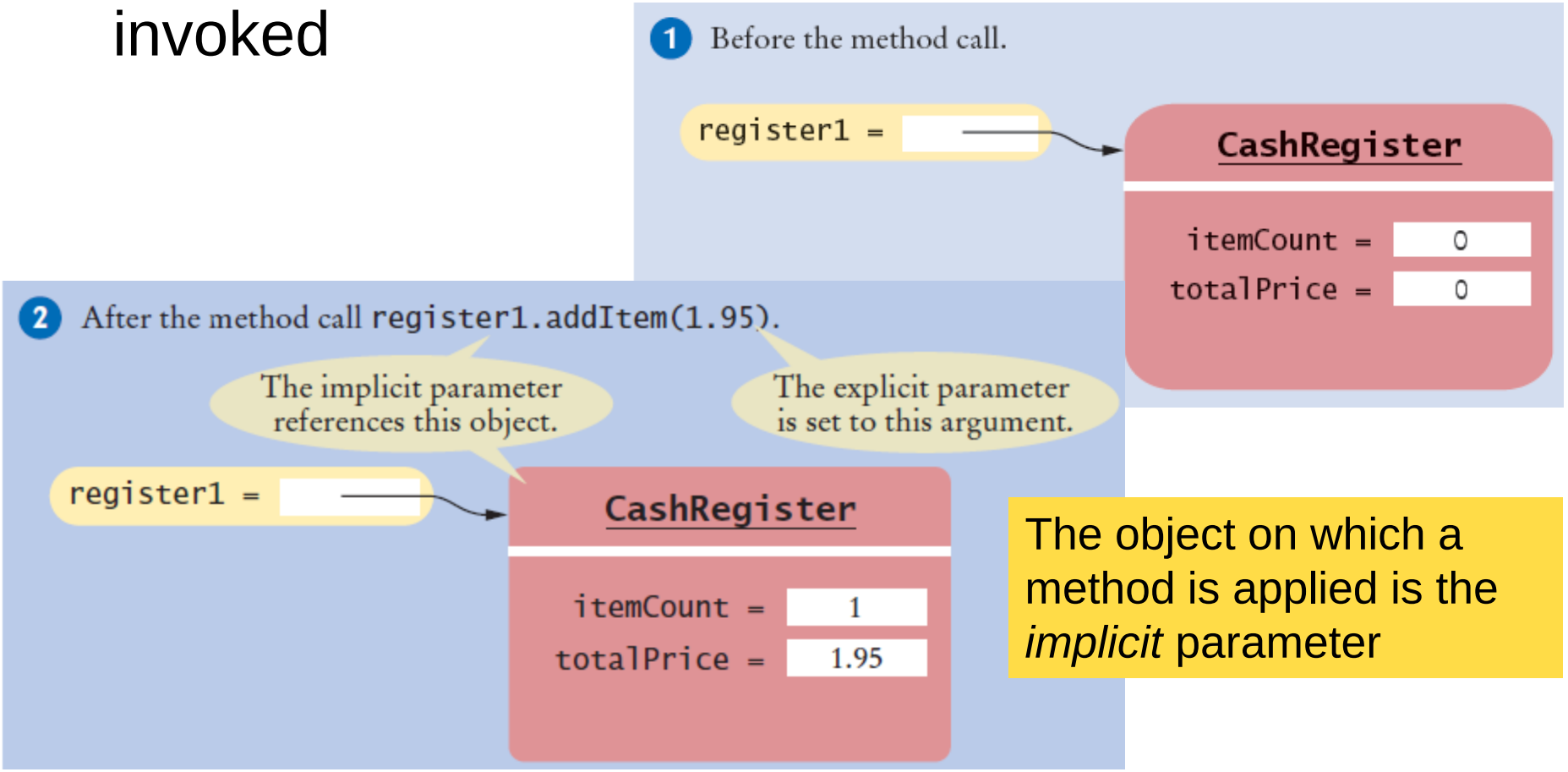
```
public class CashRegister
{
   . . .
   public void addItem(double price)
   {
      itemCount++;
      totalPrice = totalPrice + price;
   }
   . . .
}
```

Explicit parameter

Instance variables of the implicit parameter

Implicit and Explicit Parameters

- When an item is added, it affects the instance variables of the object on which the method is invoked



The object on which a method is applied is the *implicit* parameter

8.6 Constructors

- A *constructor* is a method that initializes instance variables of an object
  - It is automatically called when an object is created
  - It has exactly the same name as the class

```
public class CashRegister
{
   . . .
   /**
      Constructs a cash register with cleared item count and total.
   */
   public CashRegister() // A constructor
   {
      itemCount = 0;
      totalPrice = 0;
   }
}
```

Constructors never return values, but do not use void in their declaration

Multiple Constructors

- A class can have more than one constructor
  - Each must have a unique set of parameters

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

The compiler picks the constructor that matches the construction parameters.

```
BankAccount joesAccount = new BankAccount();
BankAccount lisasAccount = new BankAccount(499.95);
```

Syntax 8.3: Constructors

- One constructors is invoked when the object is created with the new keyword

```
public class BankAccount
{
   private double balance;

   public BankAccount()
   {
      balance = 0;
   }

   public BankAccount(double initialBalance)
   {
      balance = initialBalance;
   }
   . . .
}
```

A constructor has no return type, not even void.

A constructor has the same name as the class.

This constructor is picked for the expression new BankAccount(499.95).

The Default Constructor

- If you do not supply any constructors, the compiler will make a default constructor automatically
  - It takes no parameters
  - It initializes all instance variables

```
public class CashRegister
{
   . . .
   /**
      Does exactly what a compiler generated constructor would do.
   */
   public CashRegister()
   {
      itemCount = 0;
      totalPrice = 0;
   }
}
```

By default, numbers are initialized to `0`, `booleans` to `false`, and objects as `null`.

CashRegister.java

```
1  /**
2     A simulated cash register that tracks the ite
3     the total amount due.
4  */
5  public class CashRegister
6  {
7     private int itemCount;
8     private double totalPrice;
9
10    /**
11       Constructs a cash register with cleared it
12    */
13    public CashRegister()
14    {
15       itemCount = 0;
16       totalPrice = 0;
17    }
18
19    /**
20       Adds an item to this cash register.
21       @param price the price of this item
22    */
23    public void addItem(double price)
24    {
25       itemCount++;
26       totalPrice = totalPrice + price;
27    }
28
29    /**
30       Gets the price of all items in the current sale.
31       @return the total amount
32    */
33    public double getTotal()
34    {
35       return totalPrice;
36    }
37
38    /**
39       Gets the number of items in the current sale.
40       @return the item count
41    */
42    public int getCount()
43    {
44       return itemCount;
45    }
46
47    /**
48       Clears the item count and the total.
49    */
50    public void clear()
51    {
52       itemCount = 0;
53       totalPrice = 0;
54    }
55 }
```

Common Error 8.1

- Not initializing object references in constructor
  - References are by default initialized to `null`
  - Calling a method on a null reference results in a runtime error: **`NullPointerException`**
  - The compiler catches uninitialized local variables for you

```
public class BankAccount
{
  private String name;   // default constructor will set to null

  public void showStrings()
  {
    String localName;
    System.out.println(name.length());
    System.out.println(localName.length());
  }
}
```

Runtime Error:
java.lang.NullPointerException

Compiler Error: variable localName might not have been initialized

Common Error 8.2

- Trying to Call a Constructor
  - You cannot call a constructor like other methods
  - It is 'invoked' for you by the `new` reserved word

```
CashRegister register1 = new CashRegister();
```

  - You cannot invoke the constructor on an existing object:

```
register1.CashRegister(); // Error
```

  - But you can create a new object using your existing reference

```
CashRegister register1 = new CashRegister();
Register1.newItem(1.95);
CashRegister register1 = new CashRegister();
```

Common Error 8.3

- Declaring a Constructor as `void`
  - Constructors have no return type
  - This creates a method with a return type of `void` which is NOT a constructor!
    - The Java compiler does not consider this an error

```
public class BankAccount
{
   /**
      Intended to be a constructor.
   */
   public void BankAccount( )
   {
      . . .
   }
}
```

Not a constructor…. Just another method that returns nothing (void)