LECTURE 6

OBJECTS AND CLASSES

Ian Stavness, CMPT 270, 2017-18 Term 1

Lecture Goals

- To understand the concepts of classes, objects and encapsulation
- To implement instance variables, methods and constructors
- To be able to design, implement, and test your own classes
- To understand the behavior of object references, static variables and static methods

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.

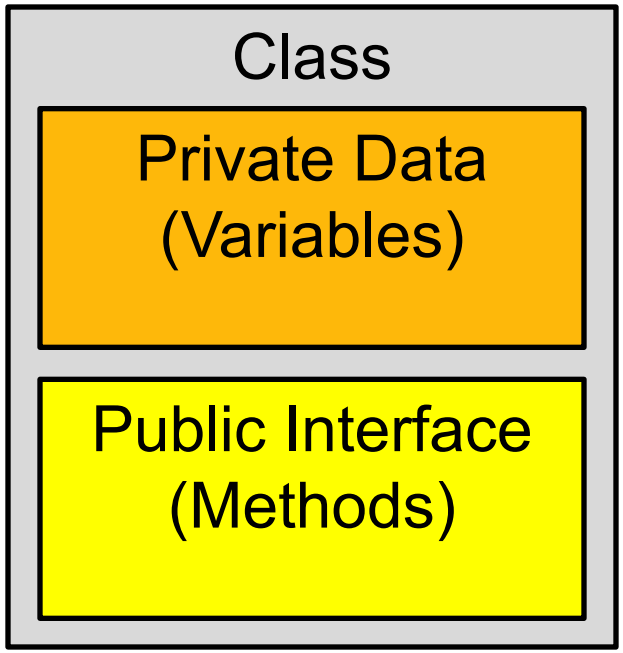
Objects and Programs

- 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

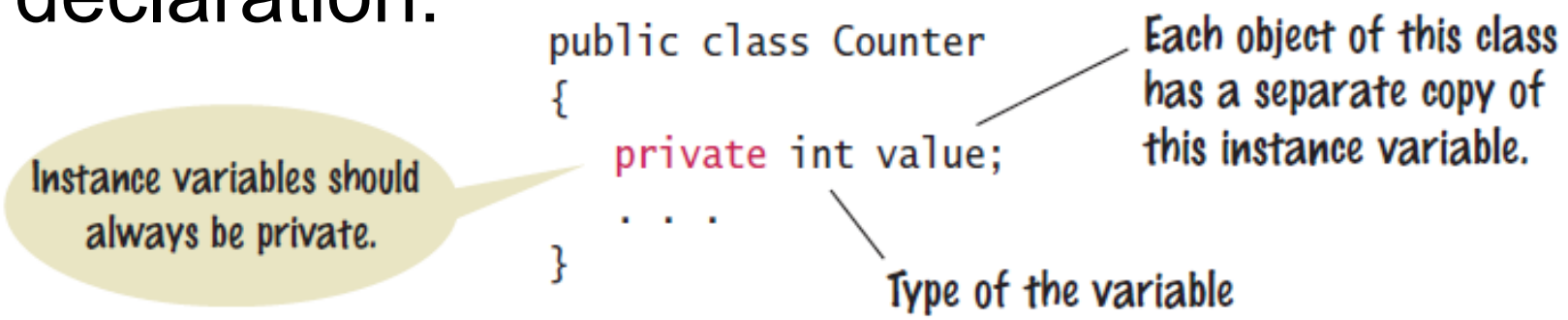


8.2 Implementing a Simple Class

- 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

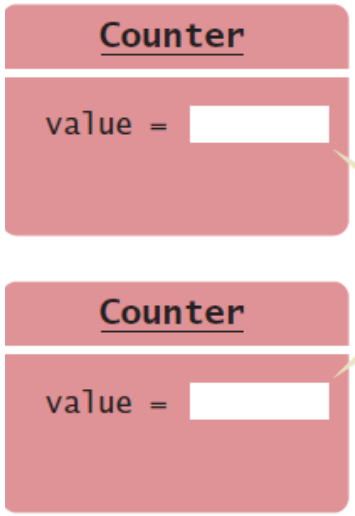
- Objects are created based on classes
  - Use the `new` operator to construct objects
  - Give each object a unique name (like variables)
- You have used the `new` operator before:

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

- Creating two instances of Counter objects:

Class name Object name Class name

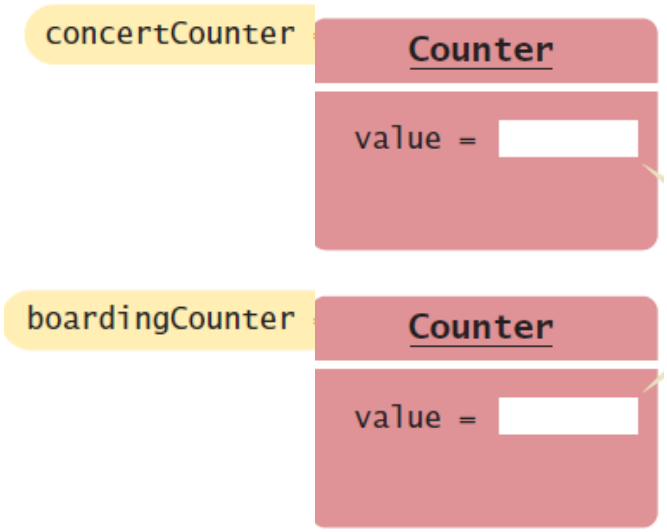
```
Counter concertCounter  = new Counter();
Counter boardingCounter = new Counter();
```



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

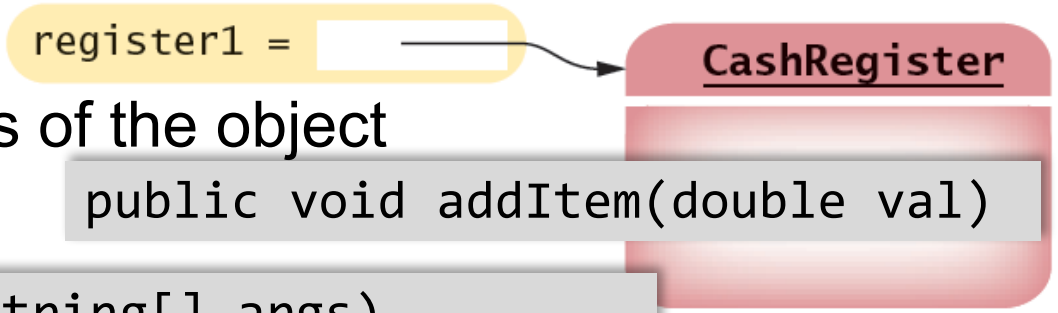
The data and method bodies make up the *private implementation* of the class

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() {  }
```

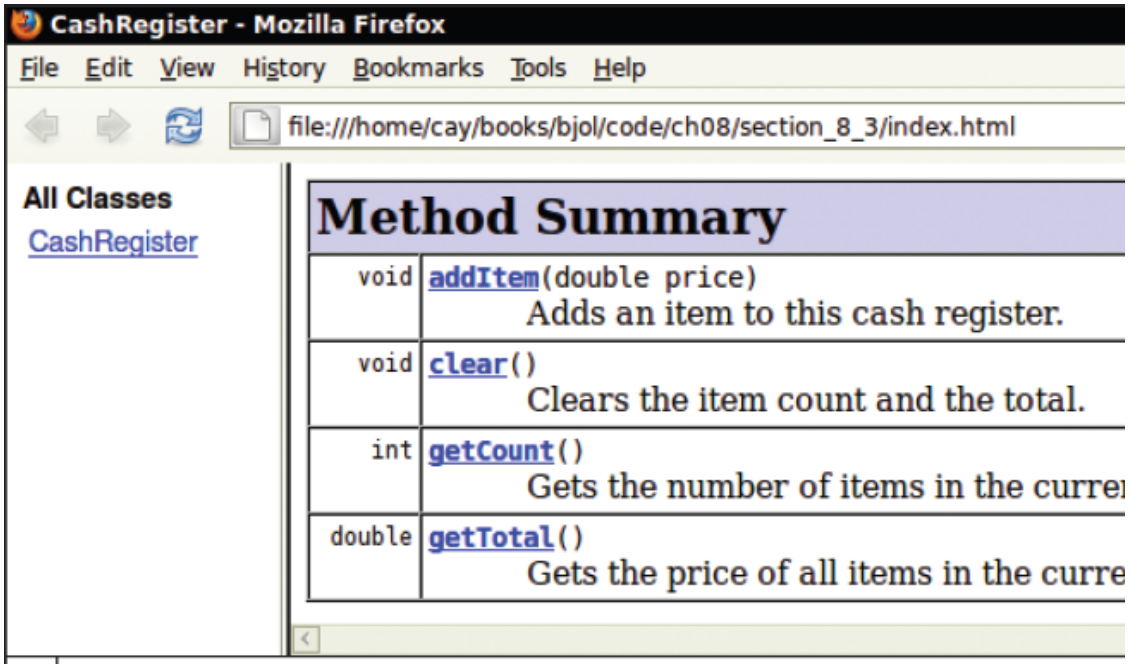
2) Mutator Methods: **'set'** methods

- 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() {  }
```

Special Topic 8.1: Javadoc

- The Javadoc utility generates a set of HTML files from the Javadoc style comments in your source code
  - Methods document parameters and returns:
    - `@param`
    - `@return`



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?

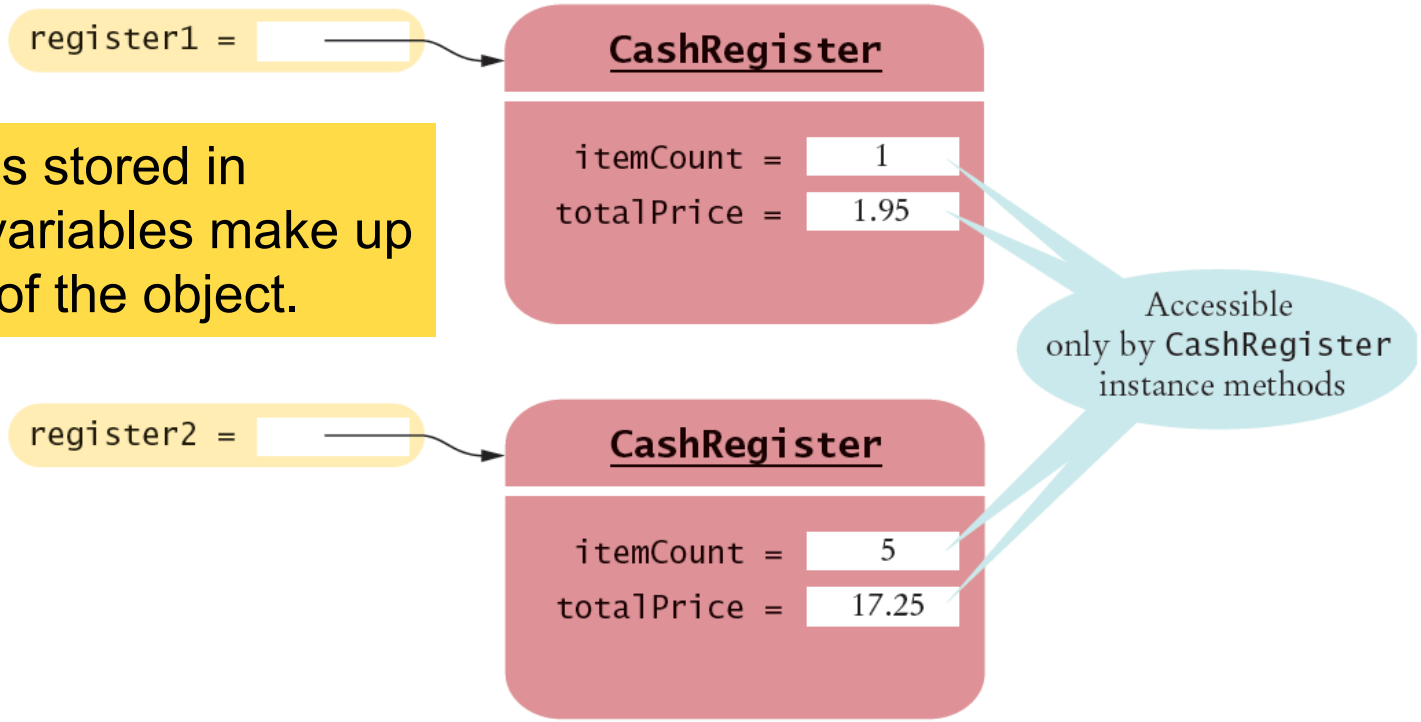
Task	Method	Data Needed
Add the price of an item	`addItem()`	`total, count`
Get the total amount owed	`getTotal()`	`total`
Get the count of items purchased	`getCount()`	`count`
Clear the cash register for a new sale	`clear()`	`total, count`

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

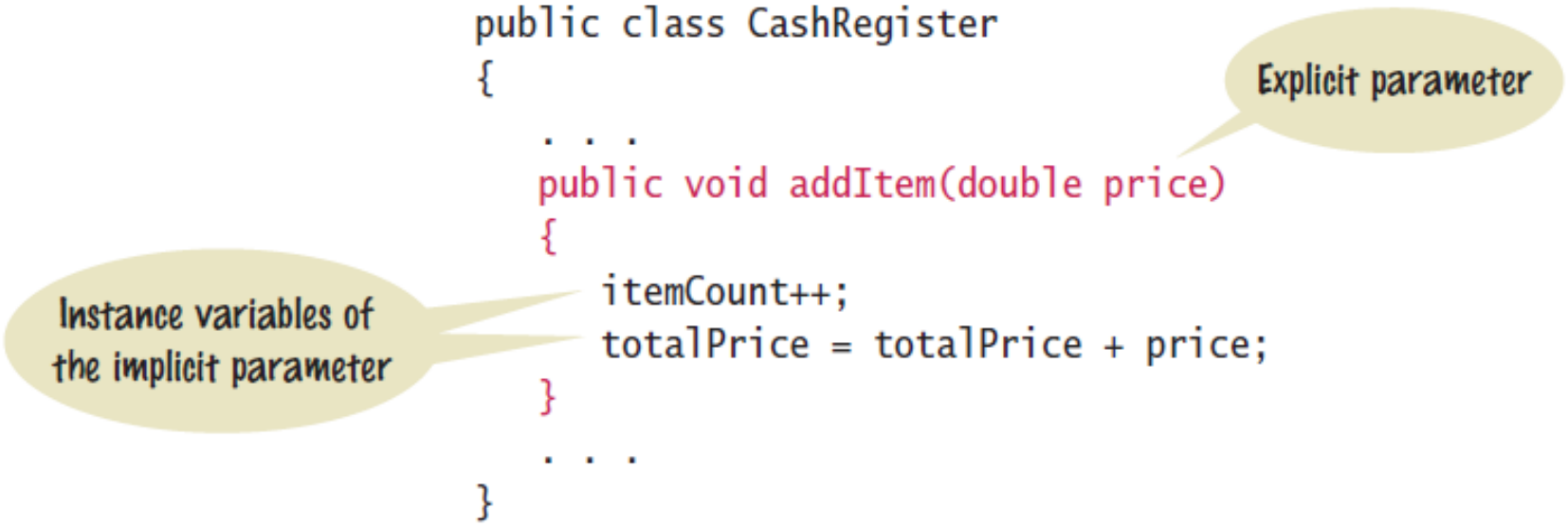
- Implement instance methods that will use the private instance variables

```
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

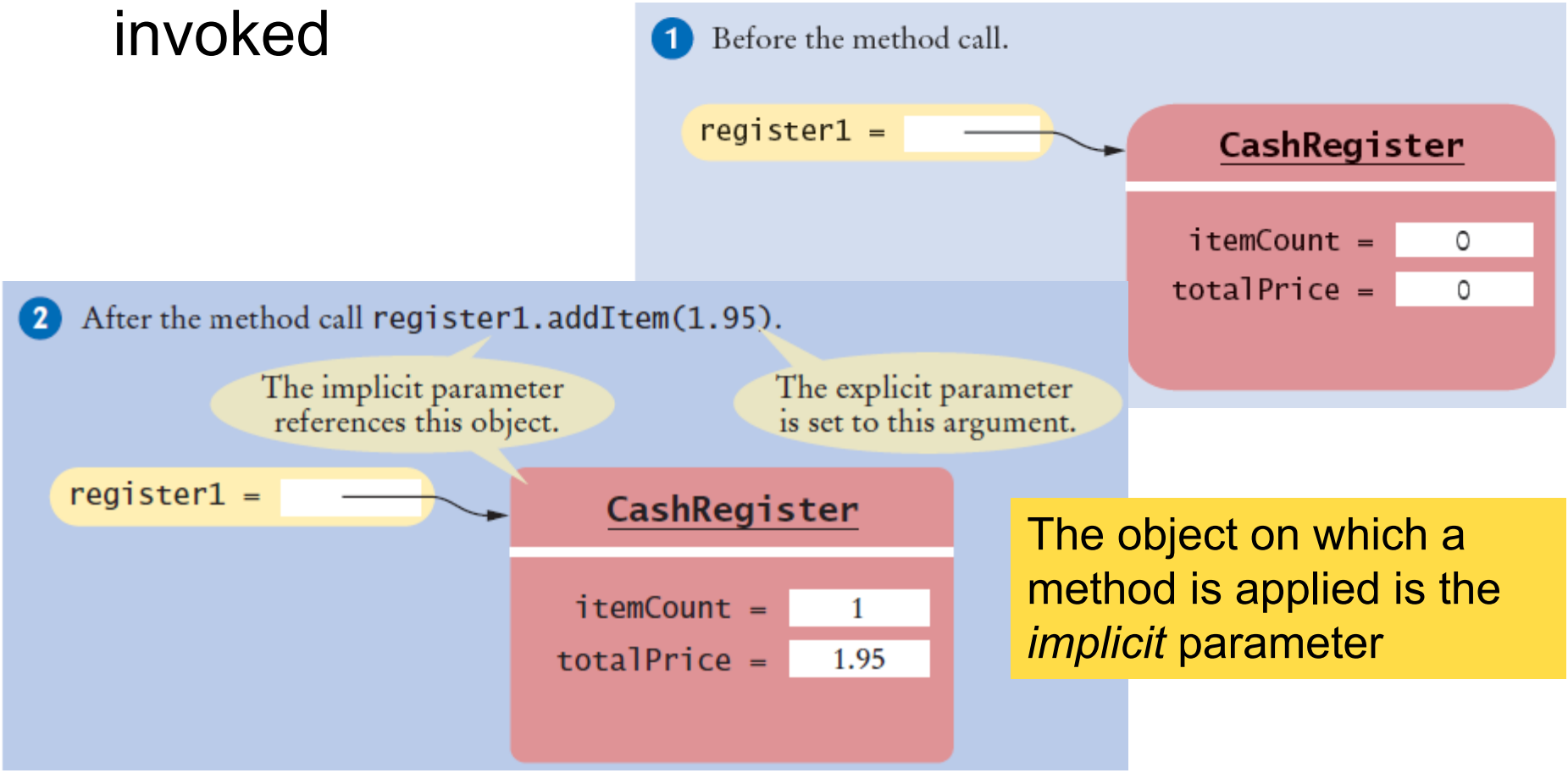
Syntax 8.2: Instance Methods

- 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



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

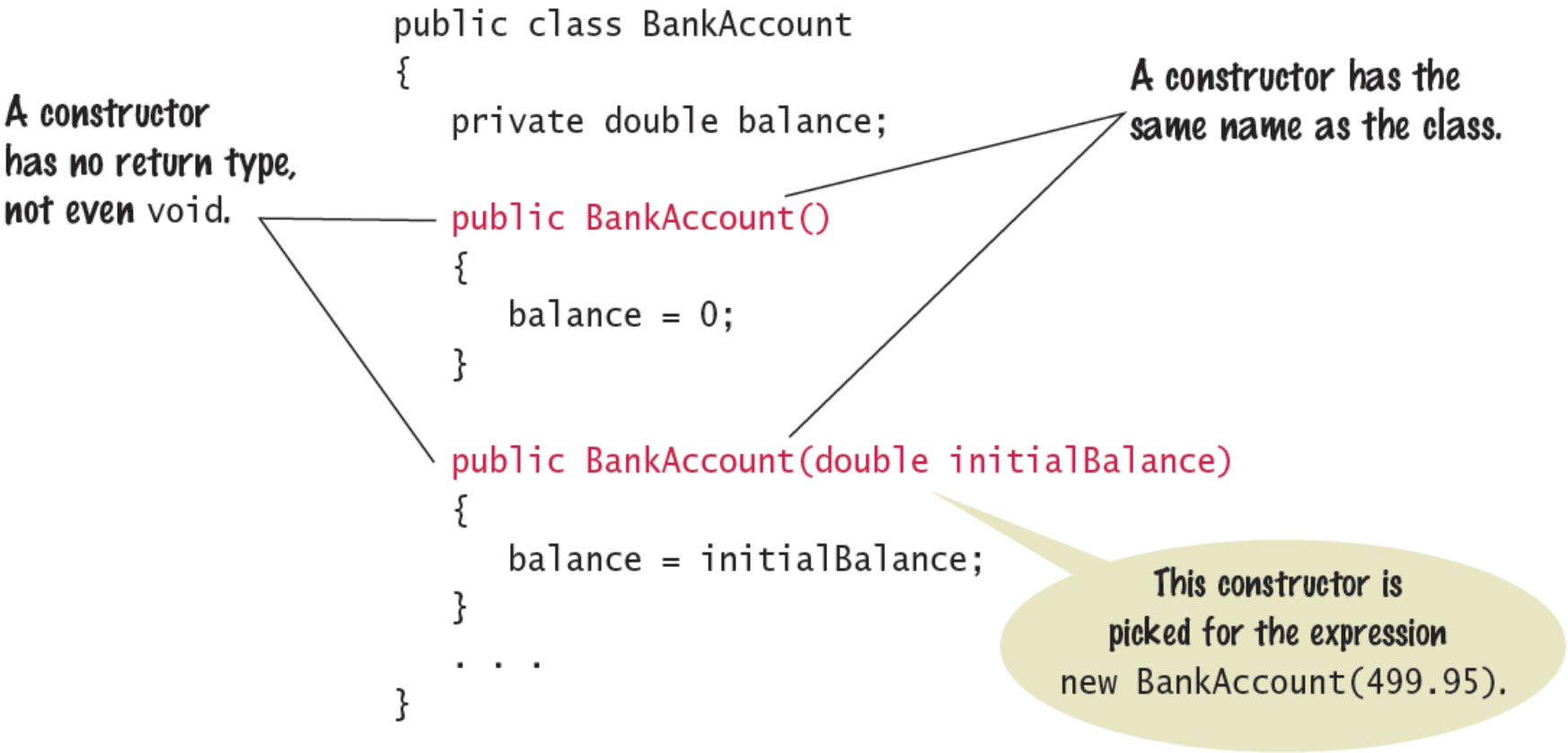
```
public class BankAccount
{
   . . .
   /**
      Constructs a bank account with a zero balance.
   */
   public BankAccount( ) { . . . }
   /**
      Constructs a bank account with a given balance.
      @param initialBalance the initial balance
   */
   public BankAccount(double initialBalance)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

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



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)

Special Topic 8.2

- Overloading
  - We have seen that multiple constructors can have exactly the same name
    - They require different lists of parameters
  - Actually any method can be overloaded
    - Same method name with different parameters

```
void print(CashRegister register)   { . . . }
void print(BankAccount account)     { . . . }
void print(int value)               { . . . }
Void print(double value)            { . . . }
```

  - We will not be using overloading in this book
    - Except as required for constructors

CashRegisterTester.java

```
1  /**
2     This program tests the CashRegister clas
3  */
4  public class CashRegisterTester
5  {
6     public static void main(String[] args)
7     {
8        CashRegister register1 = new CashRegister();
9        register1.addItem(1.95);
10       register1.addItem(0.95);
11       register1.addItem(2.50);
12       System.out.println(register1.getCount());
13       System.out.println("Expected: 3");
14       System.out.printf("%.2f\n", register1.getTotal());
15       System.out.println("Expected: 5.40");
16    }
17 }
```

Program Run

```
3
Expected: 3
5.40
Expected: 5.40
```

- Test all methods
  - Print expected results
  - Output actual results
  - Compare results

A unit test verifies that a class works correctly in isolation, outside a complete program.

8.9 Problem Solving

Patterns for Object Data

- Common patterns when designing instance variables
  - Keeping a Total
  - Counting Events
  - Collecting Values
  - Managing Object Properties
  - Modeling Objects with Distinct States
  - Describing the Position of an Object

Patterns: Keeping a Total

- Examples
  - Bank account balance
  - Cash Register total
  - Car gas tank fuel level
- Variables needed
  - Total (`totalPrice`)
- Methods Required
  - Add (`addItem`)
  - Clear
  - getTotal

```
public class CashRegister
{
   private double totalPrice;

   public void addItem(double price)
   {
      totalPrice += price;
   }
   public void clear()
   {
      totalPrice = 0;
   }
   public double getTotal()
   {
      return totalPrice;
   }
}
```

Patterns: Counting Events

- Examples
  - Cash Register items
  - Bank transaction fee
- Variables needed
  - Count
- Methods Required
  - Add
  - Clear
  - Optional: getCount

```
public class CashRegister
{
   private double totalPrice;
   private int itemCount;
   public void addItem(double price)
   {
      totalPrice += price;
      itemCount++;
   }
   public void clear()
   {
      totalPrice = 0;
      itemCount = 0;
   }
   public double getCount()
   {
      return itemCount;
   }
}
```

Patterns: Collecting Values

- Examples
  - Multiple choice question
  - Shopping cart
- Storing values
  - Array or ArrayList
- Constructor
  - Initialize to empty collection
- Methods Required
  - Add

```
public class Cart
{
   private String[] items;
   private int itemCount;
   public Cart()  // Constructor
   {
      items = new String[50];
      itemCount = 0;
   }
   public void addItem(String name)
   {
      if(itemCount < 50)
      {
         items[itemCount] = name;
         itemCount++;
      }
   }
}
```

Patterns: Managing Properties

A property of an object can be set and retrieved

- Examples
  - Student: name, ID
- Constructor
  - Set a unique value
- Methods Required
  - set
  - get

```
public class Student
{
   private String name;
   private int ID;
   public Student(int anID)
   {
      ID = anID;
   }
   public void setName(String newname)
   {
     if (newName.length() > 0)
       name = newName;
   }
   public getName()
   {
      return name;
   }
}
```

Patterns: Modeling Stateful Objects

Some objects can be in one of a set of distinct states.

- Example: A fish
  - Hunger states:
    - Somewhat Hungry
    - Very Hungry
    - Not Hungry
- Methods will change the state
  - eat
  - move

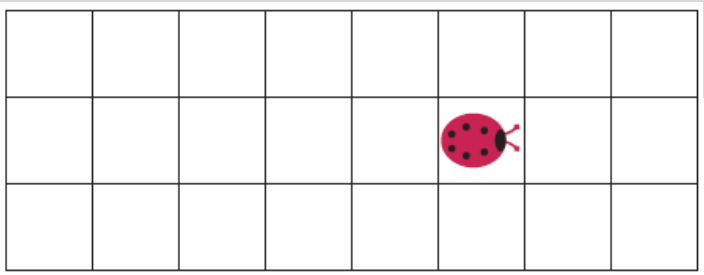
```
public class Fish
{
   private int hungry;
   public static final int
    NOT_HUNGRY = 0;
   public static final int
    SOMEWHAT_HUNGRY = 1;
   public static final int
    VERY_HUNGRY = 2;

   public void eat()
   {
      hungry = NOT_HUNGRY;
   }
   public void move()
   {
   if (hungry < VERY_HUNGRY)
     { hungry++; }
}
```

Patterns: Object Position

- Examples
  - Game object
  - Bug (on a grid)
  - Cannonball
- Storing values
  - Row, column, direction, speed. . .
- Methods Required
  - move
  - turn

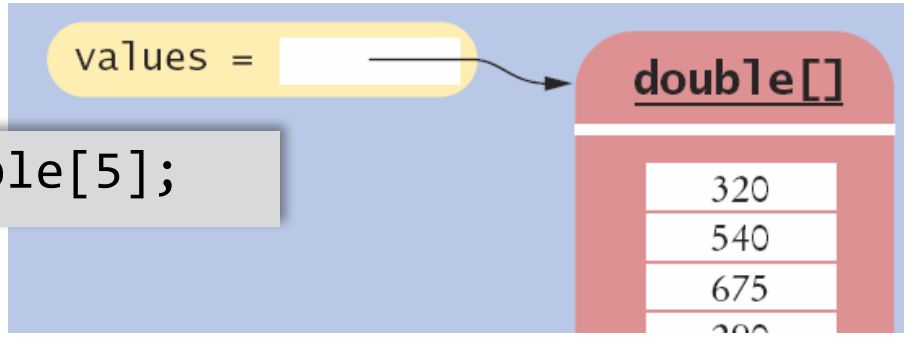
```
public class Bug
{
   private int row;
   private int column;
   private int direction;
// 0 = N, 1 = E, 2 = S, 3 = W
   public void moveOneUnit()
   {
      switch(direction) {
         case 0: row--; break;
         case 1: column++; break;
         . . .
      }
   }
}
```



8.10 Object References

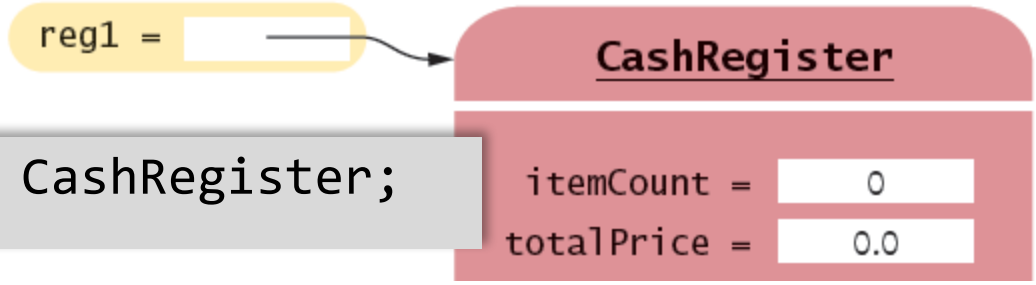
- Objects are similar to arrays because they always have reference variables
  - Array Reference

```
double[] values = new double[5];
```



  - Object Reference

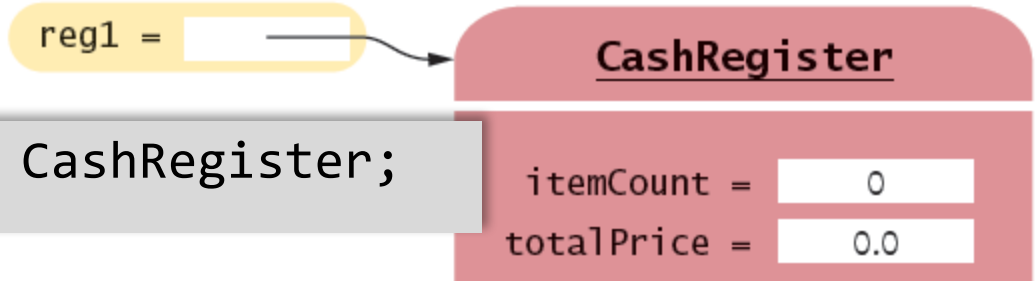
```
CashRegister reg1 = new CashRegister;
```



An object reference specifies the *memory location* of the object

Shared References

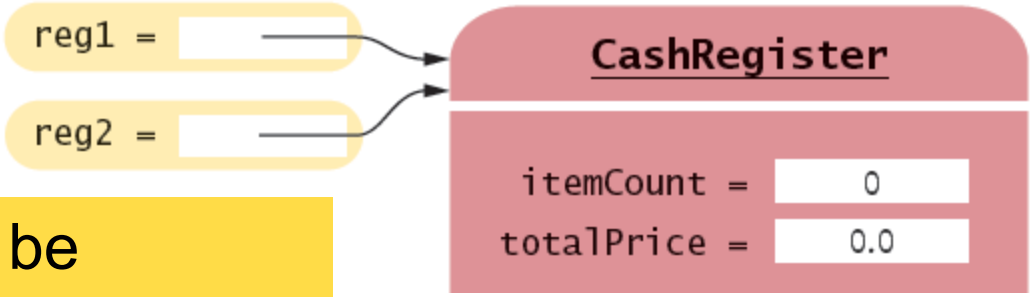
- Multiple object variables may contain references to the same object.
  - Single Reference




```
CashRegister reg1 = new CashRegister;
```

  - Shared References

```
CashRegister reg2 = reg1;
```

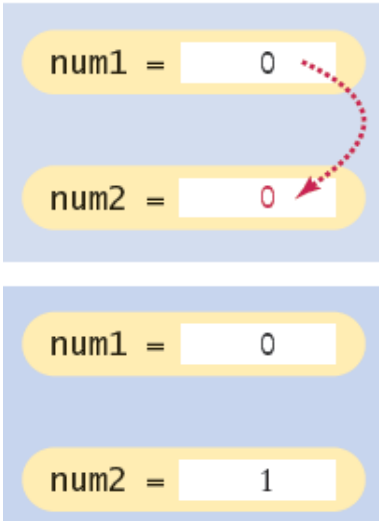


The internal values can be changed through either reference

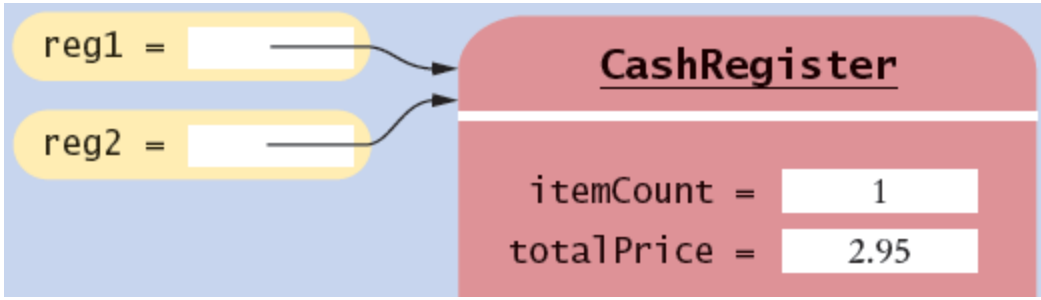
Primitive versus Reference Copy

- Primitive variables can be copied, but work differently than object references
  - Primitive Copy
    - Two locations
  - Reference Copy
    - One location for both

```
int num1 = 0;
int num2 = num1;
num2++;
```




```
CashRegister reg1 = new CashRegister;
CashRegister reg2 = reg1;
reg2.addItem(2.95);
```



Why? Primitives take much less storage space than objects!

The `null` reference

- A reference may point to 'no' object
  - You cannot invoke methods of an object via a `null` reference – causes a run-time error

```
CashRegister reg = null;
System.out.println(reg.getTotal());   // Runtime Error!
```

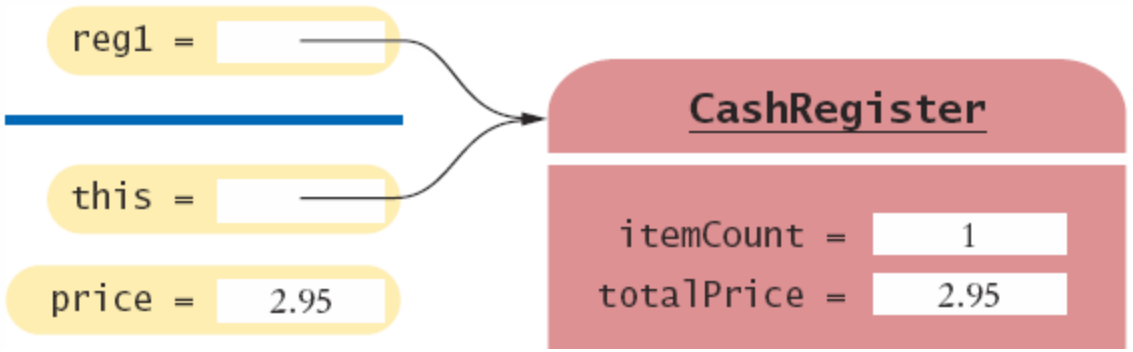
  - To test if a reference is `null` before using it:

```
String middleInitial = null; // No middle initial

if (middleInitial == null)
  System.out.println(firstName + " " + lastName);
else
  System.out.println(firstName + " " + middleInitial + ". "
   + lastName);
```

The `this` reference

- Methods receive the 'implicit parameter' in a reference variable called '`this`'
  - It is a reference to the object the method was invoked on:



  - It can clarify when instance variables are used:

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

Constructor `this` reference

- Sometimes people use the `this` reference in constructors
  - It makes it very clear that you are setting the instance variable:

```
public class Student
{
   private int id;
   private String name;
   public Student(int id, String name)
   {
      this.id = id;
      this.name = name;
   }
}
```

8.11 Static Variables and Methods

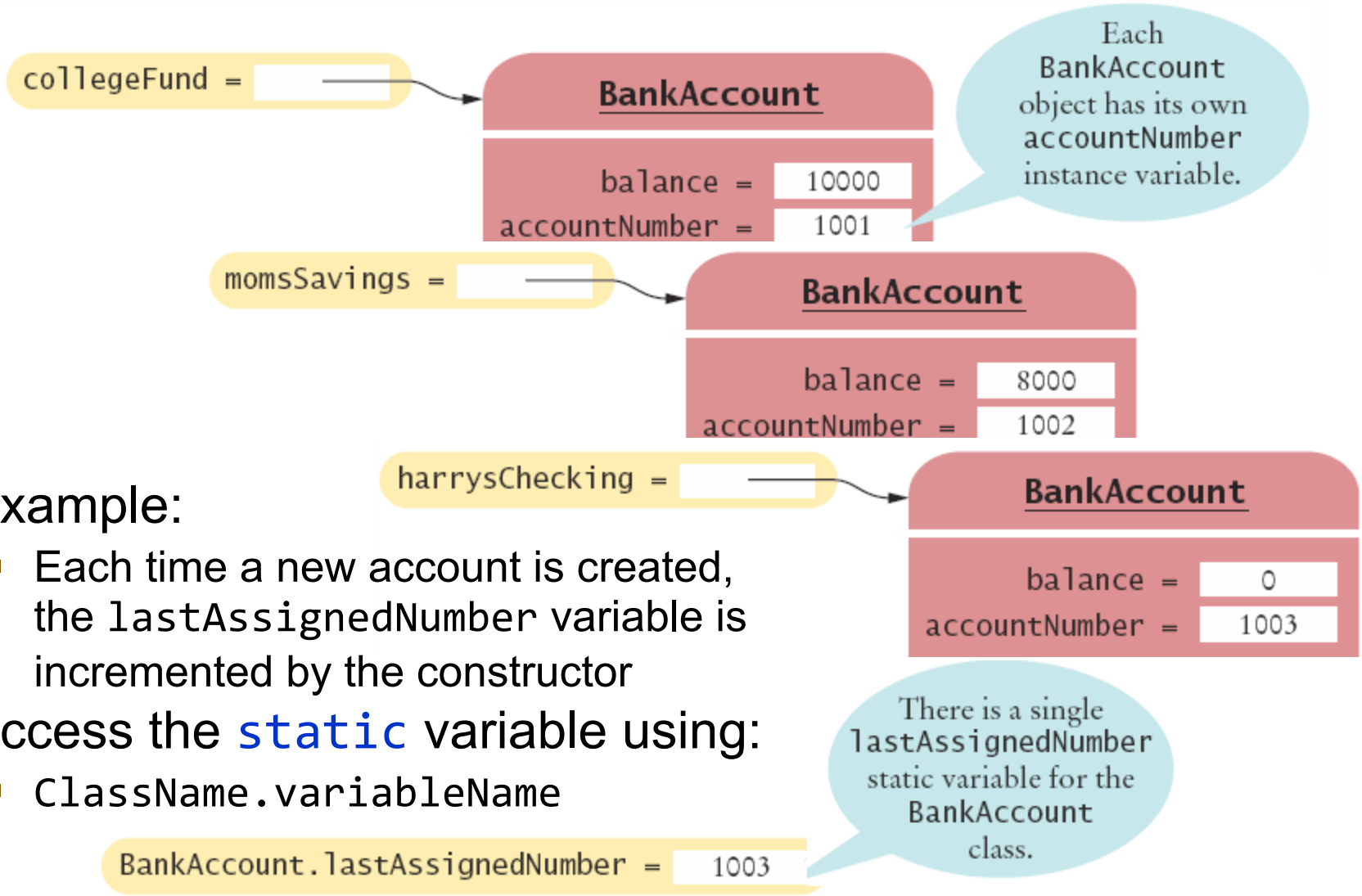
- Variables can be declared as `static` in the Class declaration
  - There is one copy of a `static` variable that is shared among all objects of the Class

```
public class BankAccount
{
  private double balance;
  private int accountNumber;
  private static int lastAssignedNumber = 1000;

  public BankAccount()
  {
    lastAssignedNumber++;
    accountNumber = lastAssignedNumber;
  }
  . . .
}
```

Methods of any object of the class can use or change the value of a static variable

Using Static Variables



- Example:
  - Each time a new account is created, the `lastAssignedNumber` variable is incremented by the constructor
- Access the `static` variable using:
  - `ClassName.variableName`

Using Static Methods

- The Java API has many classes that provide methods you can use without instantiating objects
  - The `Math` class is an example we have used
  - `Math.sqrt(value)` is a `static` method that returns the square root of a value
  - You do not need to instantiate the `Math` class first
- Access `static` methods using:
  - `ClassName.methodName()`

Writing your own Static Methods

- You can define your own `static` methods

```
public class Financial
{
   /**
      Computes a percentage of an amount.
      @param percentage the percentage to apply
      @param amount the amount to which the percentage is applied
      @return the requested percentage of the amount
   */
   public static double percentOf(double percentage, double amount)
   {
      return (percentage / 100) * amount;
   }
}
```

static methods usually return a value. They can only access static variables and methods.

- Invoke the method on the Class, not an object

```
double tax = Financial.percentOf(taxRate, total);
```

Summary: Classes and Objects

- A class describes a set of objects with the same behavior.
  - Every class has a public interface: a collection of methods through which the objects of the class can be manipulated.
  - Encapsulation is the act of providing a public interface and hiding the implementation details.
  - Encapsulation enables changes in the implementation without affecting users of a class

Summary: Variables and Methods

- An object's instance variables store the data required for executing its methods.
- Each object of a class has its own set of instance variables.
- An instance method can access the instance variables of the object on which it acts.
- A private instance variable can only be accessed by the methods of its own class.
- Variables declared as `static` in a class have a single copy of the variable shared among all of the instances of the class.

Summary: Method Headers, Data

- Method Headers
  - You can use method headers and method comments to specify the public interface of a class.
  - A mutator method changes the object on which it operates.
  - An accessor method does not change the object on which it operates.
- Data Declaration
  - For each accessor method, an object must either store or compute the result.
  - Commonly, there is more than one way of representing the data of an object, and you must make a choice.
  - Be sure that your data representation supports method calls in any order.

Summary: Parameters, Constructors

- Methods Parameters
  - The object on which a method is applied is the implicit parameter.
  - Explicit parameters of a method are listed in the method declaration.
- Constructors
  - A constructor initializes the object's instance variables
  - A constructor is invoked when an object is created with the `new` operator.
  - The name of a constructor is the same as the class
  - A class can have multiple constructors.
  - The compiler picks the constructor that matches the construction arguments.