Chapter 8 - Objects and Classes
Chapter 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
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

```java
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.
Answer: No—the object "Hello, World" belongs to the String class, and the String class has no println method.
Self Check 8.2

When using a `String` object, you do not know how it stores its characters. How can you access them?

**Answer:** Through the `substring` and `charAt` methods.
Self Check 8.3

Describe a way in which a String object might store its characters.

**Answer:** As an ArrayList<Character>. As a char array.
Suppose the providers of your Java compiler decide to change the way that a `String` object stores its characters, and they update the `String` method implementations accordingly. Which parts of your code do you need to change when you get the new compiler?

**Answer:** None. The methods will have the same effect, and your code could not have manipulated `String` objects in any other way.
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:

```java
public class Counter {
    private int value;
    // ... Type of the variable
}
```

- 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:
  ```java
  Scanner in = new Scanner(System.in);
  ```

- Creating two instances of `Counter` objects:
  ```java
  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()`

```java
public class Counter {
    private int value;

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

    public int getValue() {
        return value;
    }
}
```
Self Check 8.5

Supply the body of a method `public void reset()` that resets the counter back to zero.

**Answer:**

```java
public void reset()
{
    value = 0;
}
```
Consider a change to the implementation of the counter. Instead of using an integer counter, we use a string of \(|\) characters to keep track of the clicks, just like a human might do.

```java
public class Counter {
    private String strokes = "";
    public void count() {
        strokes = strokes + "|";
    }
    . . .
}
```

How do you implement the `getValue` method with this data representation?

**Answer:**

```java
public int getValue() {
    return strokes.length();
}
```
Self Check 8.7

Suppose another programmer has used the original Counter class. What changes does that programmer have to make in order to use the modified class?

**Answer:** None—the public interface has not changed.
Suppose you use a class `Clock` with private instance variables `hours` and `minutes`. How can you access these variables in your program?

**Answer:** You cannot access the instance variables directly. You must use the methods provided by the `Clock` class.
### 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?

<table>
<thead>
<tr>
<th>Task</th>
<th>Method</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add the price of an item</td>
<td>addItem(double)</td>
<td>void</td>
</tr>
<tr>
<td>Get the total amount owed</td>
<td>getTotal()</td>
<td>double</td>
</tr>
<tr>
<td>Get the count of items purchased</td>
<td>getCount()</td>
<td>int</td>
</tr>
<tr>
<td>Clear the cash register for a new sale</td>
<td>clear()</td>
<td>void</td>
</tr>
</tbody>
</table>
Writing the Public Interface

```java
/**
   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:
  ```java
  public static void addItem(double val)
  ```

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

```
register1 = CashRegister
```

- Then invoke methods of the object

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

```java
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
      ```java
      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
      ```java
      public void addItem(double price) { }
      public void clear() { }
      ```
Self Check 8.9

What does the following code segment print?

```java
CashRegister reg = new CashRegister();
reg.clear();
reg.addItem(0.95);
reg.addItem(0.95);
reg.addItem(0.95);
System.out.println(reg.getCount() + " " + reg.getTotal());
```

**Answer:** 2 1.90
What is wrong with the following code segment?

```java
CashRegister reg = new CashRegister();
reg.clear();
reg.addItem(0.95);
System.out.println(reg.getAmountDue());
```

**Answer:** There is no method named `getAmountDue`. 
Self Check 8.11

Declare a method `getDollars` of the `CashRegister` class that yields the amount of the total sale as a dollar value without the cents.

**Answer:** `public int getDollars();`
Self Check 8.12

Name two accessor methods of the String class.

Answer: length, substring. In fact, all methods of the String class are accessors.
Is the `nextInt` method of the `Scanner` class an accessor or a mutator?

**Answer:** A mutator. Getting the next number removes it from the input, thereby modifying it. Not convinced? Consider what happens if you call the `nextInt` method twice. You will usually get two different numbers. But if you call an accessor twice on an object (without a mutation between the two calls), you are sure to get the same result.
Self Check 8.14

Provide documentation comments for the `Counter` class of Section 8.2.

Answer:
```java
/**
   * This class models a tally counter.
   */
public class Counter {
    private int value;
    /**
      * Gets the current value of this counter.
      * @return the current value
      */
    public int getValue() {
        return value;
    }
    /**
      * Advances the value of this counter by 1.
      */
    public void count() {
        value = value + 1;
    }
}
```
Special Topic: 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
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?

<table>
<thead>
<tr>
<th>Task</th>
<th>Method</th>
<th>Data Needed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add the price of an item</td>
<td>addItem()</td>
<td>total, count</td>
</tr>
<tr>
<td>Get the total amount owed</td>
<td>getTotal()</td>
<td>total</td>
</tr>
<tr>
<td>Get the count of items purchased</td>
<td>getCount()</td>
<td>count</td>
</tr>
<tr>
<td>Clear the cash register for a new sale</td>
<td>clear()</td>
<td>total, count</td>
</tr>
</tbody>
</table>

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

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

- Use accessor methods of the class instead!

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

Encapsulation provides a public interface and hides the implementation details.

The compiler will not allow this violation of privacy.
What is wrong with this code segment?

```java
CashRegister register2 = new CashRegister();
register2.clear();
register2.addItem(0.95);
System.out.println(register2.totalPrice);
```

**Answer:** The code tries to access a private instance variable.
Consider a class `Time` that represents a point in time, such as 9 a.m. or 3:30 p.m. Give two sets of instance variables that can be used for implementing the `Time` class. (*Hint for the second set:* Military time.)

**Answer:**

(1) `int hours; // Between 1 and 12`  
`int minutes; // Between 0 and 59`  
`boolean pm; // True for p.m., false for a.m.`  

(2) `int hours; // Military time, between 0 and 23`  
`int minutes; // Between 0 and 59`  

(3) `int totalMinutes // Between 0 and 60 * 24 - 1`
Suppose the implementor of the `Time` class changes from one implementation strategy to another, keeping the public interface unchanged. What do the programmers who use the `Time` class need to do?

**Answer:** They need not change their programs at all because the public interface has not changed. They need to recompile with the new version of the `Time` class.
Consider a class `Grade` that represents a letter grade, such as A+ or B. Give two different sets of instance variables that can be used for implementing the `Grade` class.

**Answer:**
(1) `String letterGrade; // "A+", "B"
(2) `double numberGrade; // 4.3, 3.0`
Implementing Instance Methods

- Implement instance methods that will use the private instance variables

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

<table>
<thead>
<tr>
<th>Task</th>
<th>Method</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add the price of an item</td>
<td>addItem(double)</td>
<td>void</td>
</tr>
<tr>
<td>Get the total amount owed</td>
<td>getTotal()</td>
<td>double</td>
</tr>
<tr>
<td>Get the count of items purchased</td>
<td>getCount()</td>
<td>int</td>
</tr>
<tr>
<td>Clear the cash register for a new sale</td>
<td>clear()</td>
<td>void</td>
</tr>
</tbody>
</table>
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

```java
public class CashRegister {
    public void addItem(double price) {
        itemCount++;  // Instance variables of the implicit parameter
        totalPrice += price;  // Explicit parameter
    }
    ...  
}
```
Implicit and Explicit Parameters

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

1. Before the method call.

   ```
   register1 = 
   ```

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

2. After the method call `register1.addItem(1.95)`.

   ```
   register1 = 
   ```

   - The implicit parameter references this object.
   - The explicit parameter is set to this argument.
What are the values of \texttt{register1.itemCount}, \texttt{register1.totalPrice}, \texttt{register2.itemCount}, and \texttt{register2.totalPrice} after these statements?

```java
CashRegister register1 = new CashRegister();
register1.addItem(0.90);
register1.addItem(0.95);
CashRegister register2 = new CashRegister();
register2.addItem(1.90);
```

\textbf{Answer:} 2 1.85 1 1.90
Self Check 8.20

Implement a method `getDollars` of the `CashRegister` class that yields the amount of the total sale as a dollar value without the cents.

**Answer:**

```java
public int getDollars()
{
    int dollars = (int) totalPrice;  // Truncates cents
    return dollars;
}
```
Consider the substring method of the String class that is described in Section 2.5.6. How many parameters does it have, and what are their types?

**Answer:** Three parameters: two explicit parameters of type `int`, and one implicit parameter of type `String`. 
Self Check 8.22

Consider the length method of the String class. How many parameters does it have, and what are their types?

**Answer:** One parameter: the implicit parameter of type String. The method has no explicit parameters.
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

```java
public class CashRegister {
  // A constructor
  public CashRegister() {
    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

```java
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) { ... }
}

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

The compiler picks the constructor that matches the construction parameters.
**Syntax 8.2 Constructors**

- One constructor is invoked when the object is created with the `new` keyword based on arguments you supply.

```java
public class BankAccount {
    private double balance;

    public BankAccount() {
        balance = 0;
    }

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

    ...  // Additional methods
}
```

A constructor has no return type, not even void. These constructors initialize the balance instance variable. This constructor is picked for the expression `new BankAccount(499.95)`. A constructor has the same name as the class.
Initializing Instance Variables

- A constructor creates an object by initializing all instance variables defined in the class.
- If you don’t initialize an instance variable explicitly, Java does it for you by default:
  - Numbers are set to zero
  - Boolean variables are initialized as false
  - Object and array references are set to the special value null that indicates no object is associated with the variable.
- It is a good programming practice to initialize all the instance variables in a class.
Null Object References

- Uninitialized object references in a constructor are set to by `null` default.
- Calling a method on a null reference results in a runtime error: `NullPointerException`.

```java
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.
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

```java
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`. 
```java
/**
 * A simulated cash register that tracks the total amount due.
 */
public class CashRegister {
    private int itemCount;
    private double totalPrice;

    /**
     * Constructs a cash register with clear counts.
     */
    public CashRegister() {
        itemCount = 0;
        totalPrice = 0;
    }

    /**
     * Adds an item to this cash register. @param price the price of this item
     */
    public void addItem(double price) {
        itemCount++;
        totalPrice = totalPrice + price;
    }

    /**
     * Gets the price of all items in the current sale. @return the total amount
     */
    public double getTotal() {
        return totalPrice;
    }

    /**
     * Gets the number of items in the current sale. @return the item count
     */
    public int getCount() {
        return itemCount;
    }

    /**
     * Clears the item count and the total.
     */
    public void clear() {
        itemCount = 0;
        totalPrice = 0;
    }
}
```
Consider this class:

```java
public class Person
{
    private String name;

    public Person(String firstName, String lastName)
    {
        name = lastName + "", " + firstName;
    }
    ... 
}
```

If an object is constructed as

```java
Person harry = new Person("Harry", "Morgan");
```

what is its name instance variable?

**Answer:** "Morgan, Harry"
Self Check 8.24

Provide an implementation for a `Person` constructor so that after the call

```java
Person p = new Person();
```

the name instance variable of `p` is "unknown".

**Answer:**

```java
public Person() { name = "unknown"; }
```
What happens if you supply no constructor for the `CashRegister` class?

**Answer:** A constructor is generated that has the same effect as the constructor provided in this section. It sets both instance variables to zero.
Consider the following class:

```java
public class Item {
    private String description;
    private double price;
    public Item() { . . . }
    // Additional methods omitted
}
```

Provide an implementation for the constructor.

**Answer:**

```java
public Item() {
    price = 0;
    description = "";
}
```

The `price` instance variable need not be initialized because it is set to zero by default, but it is clearer to initialize it explicitly.
Self Check 8.27

Which constructors should be supplied in the Item class so that each of the following declarations compiles?

a. Item item2 = new Item("Corn flakes");
b. Item item3 = new Item(3.95);
c. Item item4 = new Item("Corn flakes", 3.95);
d. Item item1 = new Item();
e. Item item5;

**Answer:** (a) Item(String) (b) Item(double) (c) Item(String, double) (d) Item() (e) No constructor has been called.
Common Error

- 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

- Declaring a constructor as `void`
  - A constructor is not a method and doesn’t return a value
  - Constructors are never given return types

```
public void class BankAccount  //Syntax error ... don’t use void!
```
Special Topic

- 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

```java
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
We wrote a CashRegister class but...
  - You cannot execute the class – it has no `main` method
It can become part of a larger program
  - Test it first though with **unit testing**
To test a new class, you can use:
  - Programming tools that interactively create objects:
    - DrJava: [www.drjava.org](http://www.drjava.org)
    - BlueJ: [www.bluej.org](http://www.bluej.org)
  - Or write a tester class:
    - With a `main`

```java
public class CashRegisterTester {
    public static void main(String[] args) {
        CashRegister c1 = new CashRegister();
        ...
    }
}
```
BlueJ: An IDE for Testing

- BlueJ can interactively instantiate objects of a class, and allows you to invoke their methods
  - Great for testing!
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.

```java
/**
   * This program tests the CashRegister class.
   */

public class CashRegisterTester
{
  public static void main(String[] args)
  {
    CashRegister register1 = new CashRegister();
    register1.addItem(1.95);
    register1.addItem(0.95);
    register1.addItem(2.50);
    System.out.println(register1.getCount());
    System.out.println("Expected: 3");
    System.out.printf("%.2f\n", register1.getTotal());
    System.out.println("Expected: 5.40");
  }
}
```

**Program Run**

3
Expected: 3
5.40
Expected: 5.40
Self Check 8.28

How would you enhance the tester class to test the `clear` method?

**Answer:** Add these lines:

```java
register1.clear();
System.out.println(register1.getCount());
System.out.println("Expected: 0");
System.out.printf("%.2f\n", register1.getTotal());
System.out.println("Expected: 0.00");
```
Self Check 8.29

When you run the CashRegisterTester program, how many objects of class CashRegister are constructed? How many objects of type CashRegisterTester?

Answer: 1, 0
Why is the `CashRegisterTester` class unnecessary in development environments that allow interactive testing, such as BlueJ?

**Answer:** These environments allow you to call methods on an object without creating a `main` method.
Steps to Implementing a Class

1) Get an informal list of responsibilities for your objects

   Display the menu.
   Get user input.

2) Specify the public interface

   public Menu();
   public void addOption(String option);
   public int getInput();

3) Document the public interface

   - Javadoc comments

     /**
      * Adds an option to the end of this menu.
      * @param option the option to add
      */

4) Determine the instance variables

   private ArrayList<String> options;

5) Implement constructors and methods

   public void addOption(String option)
   {
     options.add(option);
   }

6) Test your class
Problem Solving: Tracing Objects

- Use an Index card for each object

- An object is manipulated through the public interface (front of the card)
- The encapsulated data is on the back of the card
Mutator Methods and Cards

- As mutator methods are called, keep track of the value of instance variables

```java
CashRegister reg2(7.5); // 7.5 percent sales tax
reg2.addItem(3.95, false); // Not taxable
reg2.addItem(19.95, true); // Taxable
```

<table>
<thead>
<tr>
<th>itemCount</th>
<th>totalPrice</th>
<th>taxableTotal</th>
<th>taxRate</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>3.95</td>
<td>0</td>
<td>7.5</td>
</tr>
<tr>
<td>2</td>
<td>23.90</td>
<td>19.95</td>
<td></td>
</tr>
</tbody>
</table>
Consider a `Car` class that simulates fuel consumption in a car. We will assume a fixed efficiency (in miles per gallon) that is supplied in the constructor. There are methods for adding gas, driving a given distance, and checking the amount of gas left in the tank. Make a card for a `Car` object, choosing suitable instance variables and showing their values after the object was constructed.

**Answer:**

```
Car myCar

Car(mpg)
addGas(amount)
drive(distance)
getGasLeft
```

```
gasLeft  milesPerGallon
0        25
```
Trace the following method calls:

```java
Car myCar = new Car(25);
myCar.addGas(20);
myCar.drive(100);
myCar.drive(200);
myCar.addGas(5);
```

Answer:

<table>
<thead>
<tr>
<th>gasLeft</th>
<th>milesPerGallon</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>20</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
</tr>
</tbody>
</table>
Self Check 8.33

Suppose you are asked to simulate the odometer of the car, by adding a method `getMilesDriven`. Add an instance variable to the object’s card that is suitable for computing this method.

**Answer:**

<table>
<thead>
<tr>
<th>gasLeft</th>
<th>milesPerGallon</th>
<th>totalMiles</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>25</td>
<td>0</td>
</tr>
</tbody>
</table>
Self Check 8.34

Trace the methods of Self Check 32, updating the instance variable that you added in Self Check 33.

**Answer:**

<table>
<thead>
<tr>
<th>gasLeft</th>
<th>milesPerGallon</th>
<th>totalMiles</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>25</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>25</td>
<td>100</td>
</tr>
<tr>
<td>16</td>
<td>25</td>
<td>300</td>
</tr>
<tr>
<td>8</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>25</td>
<td></td>
</tr>
</tbody>
</table>
Object References

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

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

- Object Reference

```java
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
    
    ```java
    CashRegister reg1 = new CashRegister;
    CashRegister reg2 = reg1;
    ```

- Shared References
  
  The internal values can be changed through either reference
### Primitive versus Reference Copy

- Primitive variables can be copied, but work differently than object references.

<table>
<thead>
<tr>
<th>Primitive Copy</th>
<th>Reference Copy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Two locations</td>
<td>One location for both</td>
</tr>
</tbody>
</table>

**Primitive Copy**
- `int num1 = 0;`
- `int num2 = num1;`
- `num2++;`

**Reference Copy**
- `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

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

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

```java
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:

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

- It can clarify when instance variables are used:

```
reg1 = CashRegister
this = reg1
price = 2.95
```
Constructor **this** Reference

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

```java
public class Student
{
    private int id;
    private String name;
    public Student(int id, String name)
    {
        this.id = id;
        this.name = name;
    }
}
```
Suppose we have a variable

```java
String greeting = "Hello";
```

What is the effect of this statement?

```java
String greeting2 = greeting;
```

**Answer:** Both `greeting` and `greeting2` refer to the same string "Hello".
After calling `String greeting3 = greeting2.toUpperCase()` , what are the contents of `greeting` and `greeting2`?

**Answer:** They both still refer to the string "Hello". The `toUpperCase` method computes the string "HELLO", but it is not a mutator—the original string is unchanged.
What is the value of `s.length()` if `s` is
a. the empty string ""?
b. `null`?

**Answer:**
(a) 0
(b) A null pointer exception is thrown.
What is the type of this in the call `greeting.substring(1, 4)`?

**Answer:** It is a reference of type `String`. 
Supply a method `addItems(int quantity, double price)` in the `CashRegister` class to add multiple instances of the same item. Your implementation should repeatedly call the `addItem` method. Use the `this` reference.

**Answer:**

```java
public void addItems(int quantity, double price)
{
    for (int i = 1; i <= quantity; i++)
    {
        this.addItem(price);
    }
}
```
Common Error

- 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

```java
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
Special Topic

▪ Calling one constructor from another
  ▪ Use this to call another constructor of the same class

```java
public class BankAccount
{
    public BankAccount(double initialBalance)
    {
        balance = initialBalance;
    }
    public BankAccount()
    {
        this(0);
    }
    . . .
}
```
Special Topic

▪ When `this` is followed by parentheses and zero or more parameters, it denotes a constructor call – `this(0)`
▪ We will not use this technique in the book
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

```java
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

```java
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

```java
double tax = Financial.percentOf(taxRate, total);
```
Self Check 8.40

Name two static variables of the System class.

**Answer:** System.in and System.out
Self Check 8.41

Name a static constant of the Math class.

**Answer:** Math.PI
The following method computes the average of an array of numbers:

```java
public static double average(double[] values)
```

Why should it not be defined as an instance method?

**Answer:** The method needs no data of any object. The only required input is the `values` argument.
Self Check 8.43

Harry tells you that he has found a great way to avoid those pesky objects: Put all code into a single class and declare all methods and variables \texttt{static}. Then \texttt{main} can call the other static methods, and all of them can access the static variables. Will Harry’s plan work? Is it a good idea?

\textbf{Answer:} Yes, it works. Static methods can call each other and access static variables—any method can. But it is a terrible idea. A program that consists of a single class with many methods is hard to understand.
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`

```java
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

```java
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

```java
public class Question {
    private ArrayList<String> choices;
    public Question() {
        choices = new ArrayList<String>();
    }
    public void add(String choice) {
        choices.add(choice);
    }
}
```
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

```java
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 String 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

```java
public class Fish {
    private int hungry;
    public static final int NOT_HUNGRY = 0;
    public static final int SOMEWHAH_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

```java
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;
            ...}
    }
}
```
Suppose we want to count the number of transactions in a bank account in a statement period, and we add a counter to the `BankAccount` class:

```java
public class BankAccount {
    private int transactionCount;
    ...
}
```

In which methods does this counter need to be updated?

**Answer:** It needs to be incremented in the `deposit` and `withdraw` methods. There also needs to be some method to reset it after the end of a statement period.
In the example in Section 8.11.3, why is the add method required? That is, why can’t the user of a `Question` object just call the add method of the `ArrayList<String>` class?

**Answer:** The `ArrayList<String>` instance variable is private, and the class users cannot access it.
Self Check 8.46

Suppose we want to enhance the `CashRegister` class in Section 8.6 to track the prices of all purchased items for printing a receipt. Which instance variable should you provide? Which methods should you modify?

**Answer:** Add an `ArrayList<Double>` `prices`. In the `addItem` method, add the current price. In the `reset` method, replace the array list with an empty one. Also supply a method `printReceipt` that prints the prices.
Consider an Employee class with properties for tax ID number and salary. Which of these properties should have only a getter method, and which should have getter and setter methods?

**Answer:** The tax ID of an employee does not change, and no setter method should be supplied. The salary of an employee can change, and both getter and setter methods should be supplied.
Look at the `direction` instance variable in the bug example in Section 8.11.6. This is an example of which pattern?

**Answer:** It is an example of the “state pattern” described in Section 8.11.5. The direction is a state that changes when the bug turns, and it affects how the bug moves.
# Packages

- Related classes are organized into Java packages

## Table 1

<table>
<thead>
<tr>
<th>Package</th>
<th>Purpose</th>
<th>Sample Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.lang</td>
<td>Language support</td>
<td>Math</td>
</tr>
<tr>
<td>java.util</td>
<td>Utilities</td>
<td>Random</td>
</tr>
<tr>
<td>java.io</td>
<td>Input and output</td>
<td>PrintStream</td>
</tr>
<tr>
<td>java.awt</td>
<td>Abstract Windowing Toolkit</td>
<td>Color</td>
</tr>
<tr>
<td>java.applet</td>
<td>Applets</td>
<td>Applet</td>
</tr>
<tr>
<td>java.net</td>
<td>Networking</td>
<td>Socket</td>
</tr>
<tr>
<td>java.sql</td>
<td>Database access through Structured Query Language</td>
<td>ResultSet</td>
</tr>
<tr>
<td>javax.swing</td>
<td>Swing user interface</td>
<td>JButton</td>
</tr>
<tr>
<td>org.w3c.dom</td>
<td>Document Object Model for XML documents</td>
<td>Document</td>
</tr>
</tbody>
</table>
Organizing Your Classes

- To put one of your classes in a package, add a line as the first instruction of the source containing the class

  ```java
  package packagename;
  ```

- The package name consists of one or more identifiers separated by periods (see 8.12.3)

  ```java
  package com.horstmann.bigjava;
  public class Financial {
      ...
  }
  ```

  The package statement adds the Financial class to the `com.horstmann.bigjava` package.
Importing Classes

- To use a class from a package, you can refer to it by its full name (package name plus class name)

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

- Using package names in this way can be inconvenient, so Java allows classes to be imported

```java
import java.util.Scanner;
```

- Imported classes can be referenced without using the package name prefix
Importing Multiple Classes

- You can import all classes of a package with an import statement that ends in \.*

```java
import java.util.*;
```

This `import` statement allows you to refer to the `Scanner` or `Random` classes without qualifying the names with `java.util`.

- You never need to import the classes in the `java.lang` package explicitly.
- Effectively, an automatic import `java.lang.*;` statement has been placed into every source file.
- You don't need to import other classes in the same package.
Syntax 8.4 Package Specification

- Organize the classes in your source file with a `package` statement

```
Syntax  package packageName;
```

```
package com.horstmann.bigjava;
```

The classes in this file belong to this package.

A good choice for a package name is a domain name in reverse.
Package Names

- When two classes have the same name, **name clashes** are avoided by putting them into different packages
- Two different Timer classes
  - `java.util.Timer`
  - `javax.swing.Timer`
- Unique package names are often constructed by reversing domain names or email addresses
  - `com.horstman`
  - `walters@cs.sjsu.edu`
Package and Source Files

- A source file must be located in a subdirectory that matches the package name.
- Parts of a package name between periods represent successively nested directories.

A class we are writing in the `problem1` directory can import the `Financial.java` class from the `com.horstmann.bigjava` subdirectory.
Self Check 8.49

Which of the following are packages?

a. java
b. java.lang
c. java.util
d. java.lang.Math

Answer: (a) No; (b) Yes; (c) Yes; (d) No
Self Check 8.50

Is a Java program without import statements limited to using the default and java.lang packages?

**Answer:** No—you can use fully qualified names for all other classes, such as java.util.Random and java.awt.Rectangle.
Self Check 8.51

Suppose your homework assignments are located in the directory 
/home/me/cs101 (c:\Users\Me\cs101 on Windows) . Your instructor tells 
you to place your homework into packages. In which directory do you place the class 
hw1.problem1. TicTacToeTester ?

**Answer:** /home/me/cs101/hw1/problem1 or, on Windows, 
c:\Users\Me\cs101\hw1\problem1
Common Error

- Dots ( . ) are used in several situations which can be confusing
  - Between package names (java.util)
  - Between package and class names (homework1.Bank)
  - Between class and inner class names (Ellipse2D.Double)
  - Between class and instance variable names (Math.PI)
  - Between objects and methods (account.getBalance())

- Consider `java.lang.System.out.println(x);`;
  - `out` – an object of type `PrintStream`
  - `System` – without context, might be an object with a public variable `out`, or a class with a static variable

- Start class names with an uppercase letter and variables, methods and packages with lowercase
Special Topic

- Package Access
  - A class, instance variable or method without an access modifier has **package access**
  - Features with package access can be accessed by all classes in the same package, which is usually not desirable.

```java
public class Window extends Container {
    String warningString;
    ...
}
```

The variable `warningString` has package access and can be accessed by any other class in `java.awt` – the package that contains the `Window` class.