User-Defined Classes

Reading: Appendix A

- Writing Methods
- Defining a Class.
- Defining Instance Variables.
- The Object Reference this.
- The toString and equals Methods.
- static Members of a Class.
- Documentation Using Javadoc.

Why Methods?
- Suppose we want the sum of the integers from 1 to 10, from 20 to 30, and from 42 to 73:

```java
public class Sums1
{
    public static void main( String[] args )
    {
        int i, sum;
        sum = 0;
        for ( i = 1; i <= 10; i++ ) {
            sum += i;
        }
        System.out.println("Sum from 1 to 10 is " + sum);
        sum = 0;
        for ( i = 20; i <= 30; i++ ) {
            sum += i;
        }
        System.out.println("Sum from 20 to 30 is " + sum);
        sum = 0;
        for ( i = 42; i <= 73; i += 1 ) {
            sum += i;
        }
        System.out.println("Sum from 42 to 73 is " + sum);
    } // end of method main
} // end of class Sums1
```

Similar, but not identical, loops.
Writing Methods:

- A method is a collection of statements grouped together to perform an operation.
- Can group the for statement and the declarations of i and sum together to create a method:

```java
public static int findSum( int start, int end)
{
    int i;
    int sum;
    sum = 0;
    for ( i = start; i <= end; i++ ) {
        sum += i;
    }
    return sum;
} // end of method findSum
```

- Can then invoke this method each time we want to compute a new sum.

```
public class Sums2
{
    public static int findSum( int start, int end)
    {
        int i;
        int sum;
        sum = 0;
        for ( i = start; i <= end; i++ ) {
            sum += i;
        }
        return sum;
    } // end of method findSum

    public static void main( String[] args )
    {
        System.out.println("Sum from 1 to 10 is " + findSum(1, 10) );
        System.out.println("Sum from 20 to 30 is " + findSum(20, 30) );
        System.out.println("Sum from 42 to 73 is " + findSum(42, 73) );
    } // end of method main
}
```

Writing Methods (continued):

- Can then invoke findSum from multiple places
Writing Methods (continued):

- **Syntax:**

```
public static int findSum(int start, int end) {
    int i;
    int sum;
    sum = 0;
    for (i = start; i <= end; i++) {
        sum += i;
    }
    return sum;
}
```

- **Method header** contains the **modifier(s)**, **return value type**, **method name**, and **parameters**.

- **Method body** contains the code to implement the function. Can use any the Java statements.

- **Return value type** is what the method produces. Can be any **type** or **class** we have covered (**int, long, String, double**, etc.).
  - Can also be **void** if the method does not return anything.

For now, the methods you write in lab and for Program #8 will all start with the same two modifiers: **public static**
Invoking a Method:

- If the method returns a value, a call to the method is usually treated as a value:
  ```java
  int answer;
  answer = findSum(5, 15);
  answer = 7 * findSum(7, 72);
  answer = findSum(16, 38) + 8 * findSum(37, 42);
  ```
- The method can be invoked anywhere that an identifier of the same type as the return value type can appear:
  ```java
  System.out.println("The result is " + findSum(3, 13));
  if ( findSum(8, 12) <= xray )
  ...
  ```
- The arguments can be identifiers:
  ```java
  int begin, finish;
  // Ask the user for the starting and ending points of the sum
  begin = inputScan.nextInt();
  finish = inputScan.nextInt();
  // Invoke findSum
  answer = findSum( begin, finish);
  ```
- Can use any type that is compatible with the formal parameters:
  ```java
  public static int findSum(int start, int end)
  {
      int i;
      int sum;
      sum = 0;
      for ( i = start; i <= end; i++ )
      {
          sum += i;
      }
      return sum;
  } // end of method findSum
  ```

Invoking a Method (continued):

- The values of the arguments are copied into the formal parameters.
  - The value of `xray` is copied to `start`. The value of `yoke` is copied to `end`.
- The method is then executed.
- The return value is given back to `main` and `main` continues executing.
Invoking a Method (continued):

- The only thing that comes back from a method is the return value.
- xray and yoke are not changed by the call to findSum. This is the case even when findSum changes the values of start and end.

```java
public class Sums3 {
    public static void main( String[] args )
    {
        int answer;
        int xray = 93;
        int yoke = 104;
        System.out.print("main: xray = " + xray);
        System.out.println("   yoke = " + yoke);
        answer = 17 * findSum( xray, yoke);
        System.out.print("main: xray = " + xray);
        System.out.println("   yoke = " + yoke);
        System.out.println("main: answer is " + answer);
    } // end of method main
}
```

```java
public class Sums3 {
    public static void main( String[] args )
    {
        int answer;
        int xray = 93;
        int yoke = 104;
        System.out.print("main: xray = " + xray);
        System.out.println("   yoke = " + yoke);
        answer = 17 * findSum( xray, yoke);
        System.out.print("main: xray = " + xray);
        System.out.println("   yoke = " + yoke);
        System.out.println("main: answer is " + answer);
    } // end of method main
}
```

Method Example:

```java
import java.util.Scanner;
public class Iterations {
    public static void main(String[] args)    {
        Scanner inputScan = new Scanner( System.in );
        int i, number;
        String word;
        do {
            System.out.print("Enter number of iterations (0 to stop): ");
            number = inputScan.nextInt();
            // Get iterations from user. Do not want a negative answer
            do {
                System.out.print("Enter number of iterations (0 to stop): ");
                number = inputScan.nextInt();
            } while ( number < 0 );
        } while ( number < 0 );
        // Print word until user gets tired of it :-(
        while ( number != 0 ) {
            System.out.print("Word to print: ");
            word = inputScan.next();
            System.out.println( word );
            System.out.println();
            // Get iterations from user. Do not want a negative answer
            do {
                System.out.print("Enter number of iterations (0 to stop): ");
                number = inputScan.nextInt();
            } while ( number < 0 );
        }
    }
}
```

```java
import java.util.Scanner;
public class Iterations {
    public static void main(String[] args)    {
        Scanner inputScan = new Scanner( System.in );
        int i, number;
        String word;
        do {
            System.out.print("Enter number of iterations (0 to stop): ");
            number = inputScan.nextInt();
            // Get iterations from user. Do not want a negative answer
            do {
                System.out.print("Enter number of iterations (0 to stop): ");
                number = inputScan.nextInt();
            } while ( number < 0 );
        } while ( number < 0 );
        // Print word until user gets tired of it :-(
        while ( number != 0 ) {
            System.out.print("Word to print: ");
            word = inputScan.next();
            System.out.println( word );
            System.out.println();
            // Get iterations from user. Do not want a negative answer
            do {
                System.out.print("Enter number of iterations (0 to stop): ");
                number = inputScan.nextInt();
            } while ( number < 0 );
        }
    }
}
```
Method Example (continued):

```java
import java.util.Scanner;
public class IterationsAgain {
    public static int getCount( Scanner inputScan ) {
        int number;
        do {
            System.out.print("Enter number of iterations (0 to stop): ");
            number = inputScan.nextInt();
        } while ( number < 0 );
        return number;
    } // end of method getCount

    public static void main(String[] args) {
        Scanner inputScan = new Scanner(System.in);
        int i, number;
        String word;
        number = getCount(inputScan);
        // Print word until user gets tired of it :-)
        while ( number != 0 ) {
            System.out.print("Word to print: ");
            word = inputScan.next();
            for (i = 0; i < number; i++)
                System.out.println(word);
            System.out.println();
            number = getCount(inputScan);
        }
    } // end of class IterationsAgain
}
```

Method Example:

```java
public class ReturnGrade {
    public static void main(String[] args) {
        System.out.println("The grade is " + getGrade(78.5));
        System.out.println("The grade is " + getGrade(93.75));
    } // end of method main

    public static char getGrade(double currentGrade) {
        char result;
        if (currentGrade >= 92.0) result = 'A';
        else if (currentGrade >= 80.0) result = 'B';
        else if (currentGrade >= 70.0) result = 'C';
        else if (currentGrade >= 60.0) result = 'D';
        else result = 'E';
        return result;
    } // end of method getGrade
} // end of class ReturnGrade
```

A method that
• has one double argument, and
• returns a char.

Are these else's necessary?
Method Example (continued):

```java
public class ReturnGradeAgain {
    public static void main(String[] args) {
        System.out.println("The grade is " + getGrade(78.5));
        System.out.println("The grade is " + getGrade(93.75));
    } // end of method main

    public static char getGrade(double currentGrade) {
        if (currentGrade >= 92.0) return 'A';
        else if (currentGrade >= 80.0) return 'B';
        else if (currentGrade >= 70.0) return 'C';
        else if (currentGrade >= 60.0) return 'D';
        else return 'E';
    } // end of method getGrade

} // end of class ReturnGradeAgain
```

The `return` statement can appear multiple places in the method.
- Here, it is in each branch of the `if` statement.
- When a `return` is executed, the method is finished.

At this point, Methods:
- How to call a method.
- How to pass arguments to a method.
- How to get a value back from a method.
- Using `static` methods by calling them from `main`.

What is next, creating a class:
- Now:
  - Instance variables.
  - Methods that access and manipulate instance variables.
- Still to come:
  - Constructors to create instances of the class
Why User-Defined Classes?

• Primitive data types (**int**, **long**, **double**, etc.) provide only a limited ability to represent data from complex objects:

  • Books:
    • ISBN.
    • Title.
    • Author(s).
    • Publisher.
    • Year published.
    • Pages.
    • ...

  • Baseball batting statistics for one player:
    • Hits.
    • Walks.
    • Batting average.
    • Strike outs.
    • ...

• Trains:
  • Locomotives: number, type, where they are located in the train.
  • Cars: type of each car, cargo in each car.
  • Length of the train.
  • Origin.
  • Destination.
  • ...

• Student record:
  • ssn, or student id -- int 2,000,000,000
  • name
  • GPA -- float or double
    • credits
    • birthdate
    • emergency contacts
    • major(s)/minor(s)

• Music:
  • composer(s)
  • artist(s) -- String (or two, or three)
  • genre
  • album
  • length -- minutes/secs, int, int
  • name
  • label -- String
  • year -- int
  • audio file itself (mp4)
  • tempo
  • last time played -- long
  • how many times played
  • sheet music
Why User-Defined Classes? (continued)

• Object-oriented programming lets us manipulate real-world objects:
  • Trains, baseball players, student information, medical records, science data, ...

• Or, hypothetical objects:
  • Vampires, Loch Ness monsters, Dungeons/Dragons, …

• Or, future objects:
  • Human colonies on Mars.

User-Defined Classes:

• Combine data and the methods that operate on the data.
  • Advantages:
    • Class is responsible for the validity of the data.
    • Implementation details can be hidden.
    • Class can be reused in many other programs.

• Client of a class:
  • A program that instantiates objects and calls methods of the class.
    • For example, you have been writing programs that create String objects and call String methods.
    • Thus, you have been writing clients of the String class.
Syntax of a Class:

- Combine data and the methods that operate on the data.

```java
public class ClassName
{
    // declare instance variables here...
    // declare one or more Constructor methods here...
    // declare methods that the programmer calls here...
    // these methods will manipulate the instance variables,
    // compute values, format results, etc.
}
```

- Use a noun for the class name.
- Begin the class name with a **Capital** letter.
- Capitalize interior words.
- Examples:
  ```java
  public class Train
  public class MedicalRecord
  public class PlayingCards
  ```

Syntax of a Class (continued):

**Terminology:**

- **Fields:** identifiers declared inside the class, but outside any method.
  - **Instance variables:** the data for each object created.
    - Contain the data specific to each instance of the object.
  - **Class variables:** static data that all objects of the class share.
  - Both types are available to all the methods.
  - For example, we might have an identifier of type `int` to hold the number of hits.
    - Each instance of the object (each player) will have a different number of hits.
- **Method:**
  - Each method in the class contains code that can manipulate one (or more) of the fields.
- **Members:**
  - Refers to both fields and methods.
**Syntax of a Class** (continued):

**Terminology** (continued):

- **Access Modifier**:
  - Determines the access rights for the class and its members (fields and methods).
  - Defines where the class and its members can be used.

<table>
<thead>
<tr>
<th>Access Modifier</th>
<th>Class or member can be referenced by…</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>methods of the same class&lt;br&gt;methods of other classes</td>
</tr>
<tr>
<td>private</td>
<td>methods of the same class only</td>
</tr>
<tr>
<td>protected</td>
<td>methods of the same class,&lt;br&gt;methods of subclasses,&lt;br&gt;methods of classes in the same package</td>
</tr>
<tr>
<td>No access modifier (known as package access)</td>
<td>methods in the same package only</td>
</tr>
</tbody>
</table>

We will primarily use either `public` or `private`.

---

**Access Modifiers** (continued):

- **Classes** are usually declared to be `public`.
  - In assignments and labs for 227, assume they are `public` unless the assignment states differently.

- **Instance variables** are usually declared to be `private` (and this will be true in 227).

- **Methods** that will be called by clients of the class are usually declared `public`.
  - **Methods** that are **only** called by other methods of the **same** class are declared `private`. 
Syntax of a Class (continued):

Defining Instance Variables:

- Declared usually at the beginning of the class.
- Will normally all be `private`.
- Can be any identifier and any of the types that we have covered.
  - `char`, `byte`, `short`, `int`, `long`, `float`, `double`
  - Can be instances of other classes; i.e., `String`, `Random`, `Scanner`.

```
import java.text.DecimalFormat;

public class BaseballStats {
    private String playerName;
    private int singles, doubles, triples, homeRuns;
    private int walks, outs, rbis;
    private DecimalFormat averageFormat;
}
```

- The class, `BaseballStats`, is `public` because we want clients (other Java classes) to be able to use it.
- The instance variables are all `private`; their values can only be accessed or changed by methods declared inside `BaseballStats` class.
- The instance variables are not declared inside any method.
Syntax of a Class (continued):

Defining Instance Variables (continued):
• Example:
  • A class to hold statistics regarding hitting for a baseball player.
    ```java
    import java.text.DecimalFormat;
    public class BaseballStats {
      private String playerName;
      private int singles, doubles, triples, homeRuns;
      private int walks, outs, rbis;
      private DecimalFormat averageFormat;
    }
    ```
  • The instance variables are all `private`; their values can only be accessed or changed by methods declared inside the `BaseballStats` class.
  • The class, `BaseballStats`, is `public` because we want clients (other Java classes) to be able to use it.

• Define instance variables for the data that all objects will have in common.
• Every instance of `BaseballStats` will need:
  • a `playerName` field.
  • a `walks` field.
  • an `outs` field.
  • etc.
• Instance variables are identifiers and their names should follow the convention we have been using:
  • Start with a lower-case letter.
  • Capitalize interior words.
Syntax of a Class (continued):

Writing Methods:

• Syntax:
  ```java
  accessModifier returnType methodName( parameter list )
  {
    // method body goes here
  }
  ```

• where:
  - `accessModifier` is one of `public`, `private`, `protected`, or absent (which means package access).
  - in 227, all methods will be either `public` or `private` (unless explicitly stated otherwise).
  - `returnType` is what the method produces. Can be any type or class we have covered (int, long, String, double, etc.). Can also be `void` if the method does not return anything.

• The `parameter list` is zero or more arguments. These are values that are sent into the method by the caller of the method.

• Inside the method, they are known as `parameters`.

• When the client calls the method these are `arguments`.

• The method name uses the same conventions as an identifier.

• The method name should start with a verb (or describe an action).

Examples:

```java
import java.text.DecimalFormat;
public class BaseballStats {
  private String playerName;
  private int singles, doubles, triples, homeRuns;
  private int walks, outs, rbis;
  private DecimalFormat averageFormat;
  public void setPlayerName( String newName )
  {
    // method body goes here
  }
  public String getPlayerName()
  {
    // method body goes here
  }
  // end of class BaseballStats
```

• The `{ }`’s are always required for methods. Does not matter how long or short the method is.

• These methods are **NOT** static methods.

• The method body can:
  - Declare variables.
  - Call other methods.
  - Use any of the program structures we have covered:
    - `if/else` statements, `switch` statements.
    - `while` loops, `for` loops, `do/while` loops.
Syntax of a Class (continued):

Writing Methods (continued):

- **Examples:**
  ```java
  public class BaseballStats
  {
  private String playerName;
  private int singles, doubles, triples, homeRuns;
  private int walks, outs, rbis;
  private DecimalFormat averageFormat;

  public void setPlayerName( String newName )
  {
  playerName = newName;
  }

  public String getPlayerName()
  {
  return playerName;
  }
  ...

  public int getHits()
  {
  return singles + doubles + triples + homeRuns;
  }

  public int getAtBats()
  {
  return getHits() + walks + outs;
  }

  public String getBattingAverage()
  {
  double average;
  int hits, atBats;

  hits = getHits();
  atBats = getAtBats();
  if ( atBats != 0 )
  average = (double) hits / atBats;
  else
  average = 0.0;

  return averageFormat.format(average);
  }
  ```

- **setPlayerName** assigns a value to the instance variable `playerName`.
- **The method does not return anything; thus, the return type is `void`**.
- **getPlayerName** returns the current contents of the instance variable `playerName`.
- **The method’s return type is `String`, since that is the type of `playerName`**.

- The number of hits is not an instance variable; it is a value computed from other variables.
- We “hide” the detail of whether the number of hits is stored as an instance variable or is computed.
- We make **public** the ability to retrieve the number of hits.

- The batting average is computed, rather than being an instance variable.
- We make **public** the ability to retrieve the batting average.

- A **String** is returned instead of a **double**.
- Baseball averages are always printed with exactly 3 decimal places.
Syntax of a Class (continued):

Writing Methods (continued):

• How to use a user-defined class.

```java
public class BaseballClient {
    public static void main(String []args) {
        BaseballStats myPlayer, nextPlayer;
        // create a player for myPlayer and for nextPlayer
        // need to call a constructor method for each...
        myPlayer.setPlayerName("Patrick");
        nextPlayer.setPlayerName("Bob");
        System.out.print("The player's are ");
        System.out.print(myPlayer.getPlayerName() + " and ");
        System.out.println(nextPlayer.getPlayerName());
        nextPlayer.setSingles(3);
        int someSingles;
        someSingles = nextPlayer.getSingles() + 2;
        nextPlayer.setSingles(someSingles);
        nextPlayer.setSingles(nextPlayer.getSingles() + 2);
        myPlayer.setSingles(myPlayer.getSingles() + 1);
    }
}
```

• Note: this example is not complete, since we have not covered constructor's.

Syntax of a Class (continued):

Writing Methods (continued):

• main is a method:

```java
    public static void main( String[] args )
    {
        // body of main
    }
```

• main is public so it can be called from outside the class.

• The Java Virtual Machine (JVM) calls main.

• static means main can be called by the JVM without instantiating an object.

• That is, the JVM does not need to use new to create an instance of the class that contains main.

• The Math class has all static methods, so we can call Math.pow() without first creating an instance of the Math class.

• void means main does not return a value.

• main is passed an array of strings. (We will cover arrays starting with the next chapter).
Syntax of a Class (continued):

Writing Methods (continued):

• Another class example. This time, a class that draws text rectangles:

    public class Rectangle
    {
        private int width;
        private int length;

        // Constructor method(s) would go here...

        public void setWidth( int wide )
        {
            // We want rectangles that have
            // a positive width
            if ( wide > 0 )
                width = wide;
            else
                width = 1;
        }

        public int getWidth()
        {
            return width;
        }

        public void setLength( int len )
        {
            // We want rectangles that have
            // a positive length
            if ( len > 0 )
                length = len;
            else
                length = 1;
        }

        public int getLength()
        {
            return length;
        }

        public void printRectangle()
        {
            // print the rectangle.
            // Use the * character.
            // The length determines the horizontal number of *'s printed.
            // The width determines the number of rows printed.

            int i, j;
            for ( i = 0; i < width; i++ )
            { 
                for ( j = 0; j < length; j++ )
                    System.out.print("*");
                System.out.println();
            }
        }
    }
Writing Methods (continued):

    public void printRotatedRectangle()
    {
        // print the rectangle.
        // Use the * character.
        // Print the rectangle rotated 90-degrees from the way that
        // printRectangle does it.
        int i, j;
        for (i = 0; i < length; i++) {
            for (j = 0; j < width; j++)
                System.out.print("*");
                System.out.println();
        }
    }

} // end of class Rectangle

Syntax of a Class (continued):

Writing Methods (continued):

- A client class that uses the Rectangle class:

    public class RectangleClient
    {
        public static void main(String[] args)
        {
            Rectangle one, two;
            one = new Rectangle( 4, 12);
            two = new Rectangle( 9, 5);
            System.out.println("One printing a rectangle:");
            one.printRectangle();
            System.out.println();
            System.out.println("Two printing a rectangle:");
            two.printRectangle();
            System.out.println();
            System.out.println("Two printing a rotated rectangle:");
            two.printRotatedRectangle();
            System.out.println();
        }
    }

These are calls to the constructor to create two instances of the Rectangle class.
Syntax of a Class (continued):

Writing Methods (continued):

```java
System.out.println("One printing a rectangle with a larger width:");
one.setWidth( 6 );
one.printRectangle();
System.out.println();
System.out.println("One printing a rotated rectangle:");
one.printRotatedRectangle();
} // end of method main
} // end of class RectangleClient
```

Syntax of a Class (continued):

**Accessor Methods:**

- Instance variables are **private** — not accessible from outside the class.
- Provide an **Accessor Method** for each instance variable that will be needed by outside users (clients) of the class.
- General form of accessor methods:

  ```java
  public returnType getInstanceVariable( )
  {
      return instanceVariable;
  }
  ```

  - By convention, the name of the accessor method will be:
    - The word **get**
    - The name of the instance variable with the first letter capitalized.
Syntax of a Class (continued):

Accessor Methods (continued):

- For the baseball example, we have the following private variables that need accessor methods:

  ```java
  private String playerName;
  private int singles, doubles, triples, homeRuns;
  private int walks, outs, rbis;
  ```

- Following the naming convention, the accessor methods will be named:

  ```java
  public String getPlayerName()
  public int getSingles()
  public int getDoubles()
  public int getTriples()
  ```

  Etc.

- Example of how this would be used to print the number of triples hit by a player:

  ```java
  BaseballStats veteran;
  veteran = new BaseballStats("Chipper Jones", 86, 23, 2, 18, 101, 359, 71);
  System.out.print(veteran.getPlayerName() + " hit ");
  System.out.println(veteran.getTriples() + " triples.");
  ```

Syntax of a Class (continued):

Mutator Methods:

- Instance variables are `private` — not accessible from outside the method.
- How do new values get assigned to these hidden variables?
- Provide an `Mutator Method` for each instance variable that will be needed by outside users of the class.
- General form of mutator methods:

  ```java
  public void setInstanceVariable( dataType newValue )
  {
    // validate newValue
    // then assign newValue to the instance variable
  }
  ```

- By convention, the name of the mutator method will be:

  - The word `set`
  - The name of the instance variable with the first letter capitalized.

  Mutator methods do not return anything.
Syntax of a Class (continued):

Mutator Methods (continued):

- Baseball example: How to modify the number of triples a player has hit:
  
  ```java
  public void setTriples( int newTriples )
  {
      if ( newTriples >= 0 )
          triples = newTriples;
      else
          triples = 0;
  }
  ```

- How do we add one to the number of triples for a player?
  - First, we need to get the current number of triples.
    - Use the `getTriples()` method to do this: `veteran.getTriples()`
  - Second, need to add 1 to the number of triples: `veteran.getTriples() + 1`
  - Third, use this as the new value for triples:
    
    ```java
    veteran.setTriples( veteran.getTriples() + 1 );
    ```

Syntax of a Class (continued):

Mutator Methods (continued):

- Write the validation code for the instance variable in the mutator method:
  
  ```java
  public void setTriples( int newTriples )
  {
      if ( newTriples >= 0 )
          triples = newTriples;
      else
          triples = 0;
  }
  ```

- Make the mutator method take care of validating the value of the instance variable.
  - Makes other code simpler, since only the mutator worries about the correct values. I.e., we do not have to worry about making the value of triples negative:
    
    ```java
    favoritePlayer.setTriples( favoritePlayer.getTriples() - 1 );
    ```
Syntax of a Class (continued):

Methods Hints (continued):

• Do not give the parameter the same name as the instance variable:

```java
public void setTriples( int newTriples )
{
    if ( newTriples >= 0 )
        triples = newTriples;
    else
        triples = 0;
}
```

The parameter is `newTriples`. The instance variable is `triples`.

• In general, do not use the name of an instance variable as the name of a local variable (one that is declared inside the method).

Syntax of a Class (continued):

The `toString` Method:

• Returns a `String` representing the data of an object.

• Example:

```java
BaseballStats rookie, veteran;

veteran = new BaseballStats("Chipper Jones", 86, 23, 2, 18, 101, 359, 71);

System.out.println(veteran.toString());
System.out.println(veteran);
```

Both print’s produce the same output.

Clients can call `toString` explicitly by coding the method call. Clients can call `toString` implicitly by using an object reference where a `String` is expected.

• The `toString` method is a great way to check the correctness of a program!

• So, what does the method look like?
Syntax of a Class (continued):

The `toString` Method (continued):

- The `toString` method for any class always begins with:
  ```java
  public String toString()
  {
    The method is public so it can be called from outside the class.
    The method returns a String.
    The method has no parameter list.
  }
  ```

- As to what should appear in the String that `toString` produces:
  - That depends(!)
  - What are the instance values stored in the class?
  - Of these, which ones would be useful to a programmer using the class?
  - For `BaseballStats`, we would want the player’s name. What else?
    - singles? doubles? averageFormat?

  Creating the String inside `toString`:
  - Take the different items and concatenate them together.
  - If there are only a few items, you can create the String in the return statement:
    ```java
    public String toString()
    {
      int playerHits = singles + doubles + triples + homeRuns;
      return playerName + " has " + playerHits + " hits.";
    }
    ```

    ```java
    public String toString()
    {
      int playerHits = getHits();
      return playerName + " has " + playerHits + " hits.";
    }
    ```

  But, for `BaseballStats`, it will likely be more useful to put more of the instance variables into the String.
Syntax of a Class (continued):

The `toString` Method (continued):

- Here’s what I have in `toString` for `BaseballStats`:
  ```java
  public String toString()
  {
      // Want to print:
      // playerName:
      //   singles: num
      //   doubles: num
      // etc.
      String stats;
      stats = playerName + ":\n";
      stats = stats + "   singles:  " + singles + "\n";
      stats = stats + "   doubles:  " + doubles + "\n";
      stats = stats + "   triples:  " + triples + "\n";
      stats = stats + "   homeruns: " + homeRuns + "\n";
      stats = stats + "   walks:    " + walks + "\n";
      stats = stats + "   outs:     " + outs + "\n";
      stats = stats + "   rbis:     " + rbis + "\n";
      return stats;
  }
  ```

Syntax of a Class (continued):

The `equals` Method:

- If we have two `BaseballStats` objects, how can we determine if they contain the same player?
  - Classes have an `equals()` method for this purpose.
    - We have seen this before when comparing two `String`'s. I.e.,
      ```java
      String zebra, xray;
      zebra = new String("Hello");
      xray = new String("Hello");
      if ( zebra.equals(xray) )
          System.out.println("the contents of zebra and xray are the same");
      else
          System.out.println("the contents of zebra and xray are NOT the same");
      // This will also work:
      if ( xray.equals(zebra) )
          System.out.println("the contents of zebra and xray are the same");
      else
          System.out.println("the contents of zebra and xray are NOT the same");
      ```
Syntax of a Class (continued):

The `equals` Method (continued):

- For a user-defined class, the author of the class has to write the `equals()` method.
- The `equals()` method is:
  - `public`.
  - Returns a `boolean`.
  - Has one parameter, which will be of the same type as the class.
- In general, the method should compare the contents of all the instance variables.
  - Return `true` if they all match; else return `false`.

```java
public boolean equals(BaseballStats otherPlayer)
{
    if ( playerName.equals( otherPlayer.getPlayerName() ) &&
         singles == otherPlayer.getSingles() &&
         doubles == otherPlayer.getDoubles() &&
         triples == otherPlayer.getTriples() &&
         homeRuns == otherPlayer.getHomeRuns() &&
         walks == otherPlayer.getWalks() &&
         outs == otherPlayer.getOuts() &&
         rbis == otherPlayer.getRbis() )
        return true;
    else
        return false;
}
```

Why does the `equals()` method not check the `averageFormat` instance variable?

Note that `playerName` is a `String`; thus, we use the `equals()` method.

Use the various get methods to check values inside `otherPlayer`. Why does the `equals()` method not check the `averageFormat` instance variable?
Syntax of a Class (continued):

The equals Method (continued):

- An alternative way to write the equals method for BaseballStats:
  
  ```java
  public boolean equals(BaseballStats obj)
  {
    return ( playerName.equals( otherPlayer.getPlayerName() ) &&
            singles == otherPlayer.getSingles() &&
            doubles == otherPlayer.getDoubles() &&
            triples == otherPlayer.getTriples() &&
            homeRuns == otherPlayer.getHomeRuns() &&
            walks == otherPlayer.getWalks() &&
            outs == otherPlayer.getOuts() &&
            rbis == otherPlayer.getRbis() );
  }
  ```

Syntax of a Class (continued):

Constructors:

- Special-purpose methods that are called only when an object is instantiated using the `new` keyword.
- A class can have several constructors.
  - The Random class has two constructors:
    ```java
    Random one;
    one = new Random( );
    ```
    - A constructor that takes no arguments is called a default constructor.
  - The second form for Random:
    ```java
    Random two;
    two = new Random( 42 );
    ```
    - This constructor takes one argument, a long.
    - Is the seed (or starting) value for the mathematical formula that Random uses to generate random values.
Syntax of a Class (continued):

Constructors (continued):

• A class can have several constructors.
  • It must have at least one!
• A constructor initializes (some or all) of the instance variables for the new object.
• Syntax:
  
  ```java
  public ClassName( parameter list )
  {
    // constructor body
  }
  ```

• Notes:
  • There is **no** return type, not even `void`.
  • Constructors do **not** return anything (ever).

**Rectangle example (from Slide #32):**

```java
public class Rectangle
{
  private int width;
  private int length;

  public Rectangle( int newWidth, int newLength )
  {
    setWidth( newWidth );
    setLength( newLength );
  } // end of constructor for a rectangle

  public void setWidth( int wide )
  {
    // We want rectangles that have
    // a positive width
    if ( wide > 0 )
      width = wide;
    else
      width = 1;
  }

  public int getWidth()
  {
    return width;
  }

  public void setLength( int len )
  {
    // We want rectangles that have
    // a positive length
    if ( len > 0 )
      length = len;
    else
      length = 1;
  }

  public int getLength()
  {
    return length;
  }
}
```

Basic idea: Pass in an initial value for each instance variable.

Use the methods for setting the width and length.
Syntax of a Class (continued):

Constructors (continued):

- Rectangle example (continued):
  - A client of Rectangle can create instance(s) of Rectangle:
    ```java
    public class RectangleClient
    {
        public static void main(String[] args)
        {
            Rectangle one, two, square;
            one = new Rectangle( 4, 12);
            two = new Rectangle( 9, 5);
            Rectangle aSquare;
            aSquare = new Rectangle( 8, 8);
            System.out.println("One printing a rectangle:");
            one.printRectangle();
            System.out.println();
            System.out.println("Two printing a rectangle:");
            two.printRectangle();
            System.out.println();
            ...
    }
    ```

Syntax of a Class (continued):

Constructors (continued):

- Rectangle example (continued):
  - A class needs one constructor, but can have more than one!
    - Java allows this provided the types and/or number of the arguments is different for each.

```java
public class Rectangle
{
    private int width;
    private int length;

    public Rectangle( int newWidth, int newLength )
    {
       .setWidth( newWidth );
        setLength( newLength );
    } // end of constructor for a rectangle

    public Rectangle( int side )
    {
        setWidth( side );
        setLength( side );
    } // end of constructor for a square
```

- A second constructor, used for creating a Rectangle that is a square.
- This method has one int parameter, compared with the two int parameters of the first constructor.
Syntax of a Class (continued):

Constructors (continued):

• Example:

```java
public class BaseballStats {
    private String playerName;
    private int singles, doubles, triples, homeRuns;
    private int walks, outs, rbis;
    private DecimalFormat averageFormat;

    // Constructor for new player who has a batting record
    public BaseballStats(String newName, int newSingles, int newDoubles,
                          int newTriples, int newHomeRuns, int newWalks,
                          int newOuts, int newRbis) {
        setPlayerName(newName);
        setSingles(newSingles);
        setDoubles(newDoubles);
        setTriples(newTriples);
        setHomeRuns(newHomeRuns);
        setWalks(newWalks);
        setOuts(newOuts);
        setRbis(newRbis);
        averageFormat = new DecimalFormat("#.000");
    }
}
```

Pass in to the constructor values for a player who already has a batting record.

Calls the set method for each of the instance variables, using the parameters passed into the constructor.

Creates averageFormat by calling the DecimalFormat constructor.

• When we want to create a player who has been playing:

```java
public class BaseballClient {
    public static void main(String[] args) {
        BaseballStats veteran;
        veteran = new BaseballStats("Chipper Jones", 86, 23, 2, 18, 101, 359, 71);
    }
}
```

• What about a new player, with no batting record?
• We could do:

```java
BaseballStats rookie;
rookie = new BaseballStats("The New Guy", 0, 0, 0, 0, 0, 0, 0);
```
Syntax of a Class (continued):

Constructors (continued):

• Java will auto-assign default values depending on the instance variable data type.
• And, Java allows multiple constructors within the same class.

    // Constructor for new player with no statistics
    public BaseballStats(String newName)
    {
        setPlayerName(newName);
        averageFormat = new DecimalFormat("#.000");
        // rest of instance variables default to zero
    }

    The instance variables not specifically set within a constructor will be set to auto-assigned default values,
    depending on the type.

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte, short, int, long</td>
<td>0</td>
</tr>
<tr>
<td>float, double</td>
<td>0</td>
</tr>
<tr>
<td>char</td>
<td>null character</td>
</tr>
<tr>
<td>boolean</td>
<td>false</td>
</tr>
<tr>
<td>String (and any other object reference)</td>
<td>null</td>
</tr>
</tbody>
</table>

Syntax of a Class (continued):

Constructors (continued):

• For our baseball example, there is a minimum realistic constructor:
  • We would have a name for each player, even one with no hitting record.
  • We always want the `averageFormat` to be the same for all players, even those whose average is 0.
• Our client can now create both new players and players who already have a hitting record:

```java
public class BaseballClient
{
    public static void main(String[] args)
    {
        BaseballStats rookie, veteran;

        rookie = new BaseballStats("The New Guy");
        veteran = new BaseballStats("Chipper Jones", 86, 23, 2, 18, 101, 359, 71);
        BaseballStats shortstop;
        shortstop = new BaseballStats("Cal Ripken, Jr.", 123, 19, 1, 17, 53, 402, 82);
        ...
    }
}
```
BaseballStats Source Code:

```java
import java.text.DecimalFormat;

public class BaseballStats {
    private String playerName;
    private int singles, doubles, triples, homeRuns;
    private int walks, outs, rbis;
    private DecimalFormat averageFormat;

    // Constructor for new player who has a batting record
    public BaseballStats( String newName, int newSingles, int newDoubles,
                          int newTriples, int newHomeRuns, int newWalks,
                          int newOuts, int newRbis)
    {
        setPlayerName( newName );
        setSingles( newSingles );
        setDoubles( newDoubles );
        setTriples( newTriples );
        setHomeRuns( newHomeRuns );
        setWalks( newWalks );
        setOuts( newOuts );
        setRbis( newRbis );
        averageFormat = new DecimalFormat("#.000");
    }

    // Constructor for new player with no batting record
    public BaseballStats( String newName )
    {
        setPlayerName( newName );
        averageFormat = new DecimalFormat("#.000");
        // rest of instance variables default to zero
    }

    public void setPlayerName( String newName )
    {
        playerName = newName;
    }

    public String getPlayerName()
    {
        return playerName;
    }

    public void setSingles( int newSingles )
    {
        if ( newSingles >= 0 )
            singles = newSingles;
        else
            singles = 0;
    }

    public int getSingles()
    {
        return singles;
    }
}
```

Perform check for valid input.
public void setDoubles( int newDoubles )
{
    if ( newDoubles >= 0 )
        doubles = newDoubles;
    else
        doubles = 0;
}
public int getDoubles()
{
    return doubles;
}

public void setTriples( int newTriples )
{
    if ( newTriples >= 0 )
        triples = newTriples;
    else
        triples = 0;
}
public int getTriples()
{
    return triples;
}

public void setHomeRuns( int newHomeRuns )
{
    if ( newHomeRuns >= 0 )
        homeRuns = newHomeRuns;
    else
        homeRuns = 0;
}
public int getHomeRuns()
{
    return homeRuns;
}

public void setOuts( int newOuts )
{
    if ( newOuts >= 0 )
        outs = newOuts;
    else
        outs = 0;
}
public int getOuts()
{
    return outs;
}
public void setWalks( int newWalks )
{
    if ( newWalks >= 0 )
        walks = newWalks;
    else
        walks = 0;
}
public int getWalks()
{
    return walks;
}

public void setRbis( int newRbis )
{
    if ( newRbis >= 0 )
        rbis = newRbis;
    else
        rbis = 0;
}
public int getRbis()
{
    return rbis;
}

public int getHits()
{
    return singles + doubles + triples + homeRuns;
}

public int getAtBats()
{
    return getHits() + outs;
}

public String getBattingAverage()
{
    double average;
    int hits, atBats;
    hits = getHits();
    atBats = getAtBats();
    if ( atBats != 0 )
        average = (double) hits / atBats;
    else
        average = 0.0;
    return averageFormat.format(average);
}
public String getOnbasePercentage()
{
    double onBasePercent;
    int hitsAndWalks, atBatsAndWalks;

    hitsAndWalks = getHits() + walks;
    atBatsAndWalks = getAtBats() + walks;
    if ( atBatsAndWalks != 0 )
        onBasePercent = (double) hitsAndWalks / atBatsAndWalks;
    else
        onBasePercent = 0.0;

    return averageFormat.format(onBasePercent);
}

public String getSluggingPercentage()
{
    double sluggingPercent;
    int totalBases, atBats;

    totalBases = singles + doubles * 2 + triples * 3 + homeRuns * 4;
    atBats = getAtBats();
    if ( atBats != 0 )
        sluggingPercent = (double) totalBases / atBats;
    else
        sluggingPercent = 0.0;

    return averageFormat.format(sluggingPercent);
} // end of class BaseballStats

Rectangle Source Code:
public class Rectangle
{
    private int width;
    private int length;

    public Rectangle( int newWidth, int newLength )
    {
        this.setWidth( newWidth );
        this.setLength( newLength );
    } // end of constructor for a rectangle

    public Rectangle( int side )
    {
        setWidth( side );
        setLength( side );
    } // end of constructor for a square

    public void setWidth( int wide )
    {
        // We want rectangles that have a positive width
        if ( wide > 0 )
            width = wide;
        else
            width = 1;
    }
public int getWidth()
{
    return width;
}

public void setLength( int len )
{
    // We want rectangles that have a positive length
    if ( len > 0 )
        length = len;
    else
        length = 1;
}

public int getLength()
{
    return length;
}

public void printRectangle()
{
    // print the rectangle.
    // Use the * character.
    // The length determines the horizontal number of *'s printed.
    // The width determines the number of rows printed.

    int i, j;
    for (i = 0; i < width; i++) {
        for (j = 0; j < length; j++)
            System.out.print("*");
        System.out.println();
    }
}

public void printRotatedRectangle()
{
    // print the rectangle.
    // Use the * character.
    // Print the rectangle rotated 90-degrees from the way that
    // printRectangle does it.

    int i, j;
    for (i = 0; i < length; i++) {
        for (j = 0; j < width; j++)
            System.out.print("*");
        System.out.println();
    }
}
public boolean equals( Rectangle anotherRec )
{
    if ( width == anotherRec.width &&
         length == anotherRec.length )
        return true;
    else
        return false;
}

public String toString()
{
    String answer;
    answer = "width = " + width + ";  length = " + length;
    return answer;
}

public String toString()
{
    // Want to print:
    //   playerName:
    //      singles: num
    //      doubles: num
    //      etc.
    String stats;
    stats = playerName + ";\n";
    stats = stats + " singles:  " + singles + "\n";
    stats = stats + " doubles:  " + doubles + "\n";
    stats = stats + " triples:  " + triples + "\n";
    stats = stats + " homeruns: " + homeRuns + "\n";
    stats = stats + " walks:    " + walks + "\n";
    stats = stats + " outs:     " + outs + "\n";
    stats = stats + " rbis:     " + rbis + "\n";
    return stats;
}
} // end of class Rectangle
Syntax of a Class (continued):

The Object Reference this:

• **this** is an object reference to the object for which the method was called.

• **this** is an *implicit parameter* automatically sent to methods.

```java
public void setTriples( int newTriples )
{
    if ( newTriples >= 0 )
        this.triples = newTriples;
    else
        this.triples = 0;
}
```

• **this.triples** is the instance variable.

• The code above is equivalent to:

```java
public void setTriples( int newTriples )
{
    if ( newTriples >= 0 )
        triples = newTriples;
    else
        triples = 0;
}
```

• Either form is fine.

Syntax of a Class (continued):

The Object Reference this (continued):

• Using **this**, we could write:

```java
public void setTriples( int triples )
{
    if ( triples >= 0 )
        this.triples = triples;
    else
        this.triples = 0;
}
```

• The code above will work just fine.

• It is **not** a good way to write clear, understandable code.

• It works much better to use a different name for the parameter

  • I.e., **newTriples** from the previous slide.

• Writing code like the above in this course will cost you points!
**Class Summary:**

- Name the class: Capitalize the first letter, and internal words.
  - Declare the class to be `public`.
- What instance variables will the class need?
  - Declare the instance variables to be `private`.
- Constructor:
  - Need at least one.
  - Declare each to be `public`.
  - Pass as parameters those values to be used as the starting values for a particular object.
- Accessor and Mutator methods (`get` and `set`, respectively).
  - Each instance variable gets a `get` and a `set` method.
- Write `toString()` method to print the contents of the instance variables.
  - Very useful in tracking down errors!
- Write `equals()` method to compare two objects of the class.