## Why User-Defined Classes?

- Primitive data types (**`int`**, **`long`**, **`double`**, etc.) provide only a limited ability to represent data from complex objects such as:
  - Books:
    - ISBN.
    - Title.
    - Author(s).
    - Publisher.
    - Year published.
    - Pages.
    - ...
  - Baseball batting statistics for one player:
    - Hits.
    - Walks.
    - Batting average.
    - Strike outs.
    - ...

## A class/object:

- Combines data and the methods that operate on the data.
  - Advantages:
    - Class is responsible for the validity of the data.
    - Implementation details can be hidden.
    - It is a way to organize and think about a program/solution.
    - Classes are easily reused in many programs.
- A class is a template (think "cookie cutter")
- An object is an instance of a class (think cookie)
  - That said, there are data/methods tied to the class instead of an object
- Terminology: a c*lient* of a class is a program that uses that 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.

## Not a new idea

- You've seen and used classes/objects before in Python
- The basic idea is the same in Java, but the syntax is different.
- Some differences are:
    - In Python a class has one constructor - in Java it may have many
    - In Java class/object variables must be declared
    - In Java everything has to have a type, including the return value of methods

## Syntax of a Class:

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

```
public class ClassName {
   // declare Fields 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.


}
```

- Conventions:
  - Use a noun for the class name.
  - Begin the class name with a Capital letter.
  - Capitalize interior words.
  - Examples:

```
public class Train
public class MedicalRecord
public class PlayingCards
```

**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.

- 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.

| **Access Modifier** | **Class or member can be referenced by…** |
|---|---|
| **`public`** | methods of the same class<br>methods of other classes |
| **`private`** | methods of the same class only |
| **`protected`** | methods of the same class,<br>methods of subclasses,<br>methods of classes in the same package |
| No access modifier<br>(known as *package access*) | methods in the same package only |

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

## Rules of Thumb

- Instance variables are usually declared to be **`private`**.
- 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`**.

- You may only have one class declared as **`public`** per file, and the file name must match the class name (with the extension **`.java`**)

**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`**.

**Defining Instance Variables** (continued):

- Example:
  - A class to hold statistics regarding hitting for a baseball player.

```
import java.text.*;

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

   private DecimalFormat averageFormat;
```

- The **`DecimalFormat`** class provides a way to control the appearance of numbers that are converted to String's.
- You create an instance of **`DecimalFormat`** for each format you want to use.
- Note: Have to import the **`DecimalFormat`** class from the *java.text* package:

```
   import java.text.DecimalFormat;
```

## **Private vs Public**:

- What is the difference between **public** and **private** fields?
  - public fields can be accessed by any code
  - For example, suppose we had declared the singles field to be public in the previously defined **`BaseballStats`** class?

```
public int singles;
```

  - Then any code that creates a **`BaseballStats`** object can use the field directly:

```
BaseballStats player;
// code here to instantiate player
System.out.println("Singles = " + player.singles);
player.singles = 3;
```

## **Private vs Public**:

- Declaring a field to be public violates the notion of *encapsulation* or *information hiding*.
    - Encapsulation is the idea that clients of the class do not know how the class is implemented. They just know how to work with it through public methods.
    - Why would you want this?
        - It allows you to protect the integrity of the objects data.
        - It allows you to change how the class is implemented without causing code that used the object to break.

**Private vs Public**:

- If we have the fields declared to be **private**

```
private int singles;
```

  - Then the following code causes a compiler error:

    ```
    BaseballStats player;
    // code here to instantiate player
    System.out.println("Singles = " + player.singles);
    player.singles = 3;
    ```

- So how do we access private fields?

**Accessor Methods**:

- If 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:

```
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.

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

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

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

```
  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:

```
    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.");
```

## **Mutator Methods**:

- Instance variables are usually **`private`** — not accessible from outside the class.
- 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:

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

- Mutator methods do not return anything.

- By convention, the name of the mutator method will be:
  - The word **`set`** followed by:
  - The name of the instance variable with the first letter capitalized.

- Baseball example: How to modify the number of triples a player has hit:

```
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:

```
veteran.setTriples( veteran.getTriples() + 1 );
```

- Write the validation code for the instance variable in the mutator method:

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

}
```

Validation: Cannot have a negative number of triples.

- 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:

```
favoritePlayer.setTriples( favoritePlayer.getTriples() - 1);
```

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

```
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).

- Note that we say in the style guide that all methods should start with comments saying what their parameters are and what they do. Accessor and mutator methods are and exception to this rule.

## **Constructors**:

- Special-purpose methods that are called only when an object is ***instantiated*** using the **`new`** keyword.
- A class can have several constructors.
    - We’ve seen two constructors for the **`Scanner`** class:

```
Scanner in;
 File inFile = new File(“myFile”);
 in = new Scanner(System.in);  //one form of constructor
 in = new Scanner(inFile);     //second form of constructor
```

- These calls might look the same, but the use different types for the parameter.
- **`System.in`** is an **`InputStream`** object
- **`inFile`** is a **`File`** object
- There are actually 10 different constructors for **`Scanner`** in Java 8

**Constructors, Python vs Java**:

- A constructor in Python looks like this:

```
class Word:
    def __init__(self, word):
        self.word = word.lower()
        self.count = 0
```

- The name of the constructor is `__init__`. In Java constructors have the same name as the class.
- In Python the object itself is a parameter (self). In Java the object is not passed as an explicit parameter.
- The constructor above in Python would look something like:

```
public Word(String newWord) {
  word = newWord.toLowerCase();
  count = 0;
}
```

- A class can have several constructors.
- A constructor initializes (some or all) of the instance variables for the new object.
- Syntax:

```
public ClassName( parameter list ) {
   // constructor body
}
```

- Notes:
  - There is *no* return type, not even **`void`**.
  - Constructors do *not* ever specify a return type.

- **`Rectangle`** example:

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

```
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;
  }
```

Use the methods for setting the **`width`** and **`length`**.

- A client of **`Rectangle`** can create instance(s) of **`Rectangle`**:

```
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();

    ...
```

- A class needs one constructor, but can have more than one!
  - Java allows this provided the <u>types</u> and/or <u>number</u> of the arguments is different for each.

```
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 <u>two</u> **`int`** parameters of the first constructor.

- Example:

```
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");
  }
...
```

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

```
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:

```
BaseballStats rookie;
rookie = new BaseballStats("The New Guy", 0, 0, 0, 0, 0, 0, 0);
```

- 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.

| **Data Type** | **Default Value** |
|---|---|
| `byte`, `short`, `int`, `long` | `0` |
| `float`, `double` | `0` |
| `char` | `nul character` |
| `boolean` | `false` |
| `String` (and any other object reference) | `null` |

- 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:

```
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);
    ...
```

## The Object Reference *this*:

- `this` is an object reference to the object for which the method was called. It is analogous to ***`self`*** in Python.
  - `this` is an *implicit parameter* automatically sent to methods.

```
 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:

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

 }
```

  - Either form is fine.

- Using **`this`**, we could write:

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

 }
```

  - The code above will work just fine.
  - It is <u>**not**</u> 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 Methods**:

- Syntax:

```
accessModifier returnType methodName( parameter list )
{
   // method body goes here

}
```

  - where:
    - *accessModifier* is one of **`public`**, **`private`**, **`protected`**, or absent (which means *package access*).
    - *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:

```
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;
   }
   . . .
```

- **`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 **`return`** statement in Java returns the value of the indicated expression.
- Here, the expression is the contents of the variable **`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);
 }
```

- 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.

```
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);
 }
```

- 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.

```
public class BaseballClient {
   public static void main(String []args)
   {
      BaseballStats myPlayer, nextPlayer;
      myPlayer = new BaseballStats("");
      nextPlayer = new BaseballStats("");

      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 );

      System.out.println("hits: " + nextPlayer.getSingles() );
   }
} // end of class BaseballClient
```

## <u>public vs private methods</u>

- Just like with field access, **public** methods can be called by anyone and **private** methods can only be called by the class itself.
- Why would we ever want to use **private** methods?
  - Clearly any method which performs an action an object in the class is expected to do should be made public.
    - For example our BaseballStats class should have public methods for getting or updating stats and our Rectangle class might have a method for printing the rectangle.
  - Sometimes you write "helper" methods to make your implementation of the main class methods neater, better organized, and easier to maintain.
    - What not make those methods public? What's the harm?
      - It violates the "spirit" of what the object is.
      - You now need to support the helper method to maintain compatibility. In other words, if you would like to change what it does or what parameters it takes or even get rid of it, you can't without breaking all the clients that call it.

## Overloading

- Remember a class can have more than one constructor.
  - It can also have more than one method with the same name.
  - It is the concept of *overloading* that allows this.
- Java identifies a method not just by it’s class and name, but also by the types of its parameters.
- For example, suppose you wanted to have a method to square a number. You might write

```
public static integer squareMe(int x) {
  return x * x;
}
```

- This would work to square an integer, but not a double. ☹
- How would I square a double?

## Overloading

- How would I square a double?
- I could write

```
public static double squareMe(double x) {
  return x * x;
}
```

- Now I have two methods named **squareMe**, but this is OK because the parameters are of different types. If I write a client that does:

```
int x = 47;
int sx = MyClass.squareMe(x);
```

- The compiler knows to use the first version because the parameter is an **`int`**.

## Overloading

- Java does NOT look at the return type when determining which method to use. In other words, you are NOT allowed to have two methods in a class with the same name and the same parameter types even if they have a different return type.
- Suppose I wanted to write a method to halve an integer, truncating the remainder.

```
public static int halveMe(int x) {
  return x / 2;
}
```

- Now suppose I also want a version that returns a double to capture the .5 for odds.
- If I added the method:

```
public static double halveMe(int x) {
  return x / 2.0;
}
```

- The compiler would give me an error because I have two methods named "**halveMe**" that take a single `int` argument. The fact that the have different return values doesn't matter.

## static fields

- If a field is declared with the **`static`** modifier, it is a *static* or *class* variable instead of an *instance* variable.
- A class variable is tied to a class, not an object. Only one copy of that variable exists and all objects in the class have access to it. For example

```
public class Nums {
  private int num1;
  private static int num2 = 1;

  public Nums() {
    num1 = 1;
  }

  public void incPrintNums() {
    System.out.println(++num1);   //increment and print
    System.out.println(++num2);
  }
```

## static fields

- Given the code on the previous slide, what will the following print?.

```
Nums n1 = new Nums();
Nums n2 = new Nums();
n1.incPrintNums();
n2.incPrintNums();
```

- It will print:

```
  2
  2
  2
  3
```

## static fields

- Why would we want static fields?
- A good example is an id number.
- Suppose you had a class students. You might want to have a unique identifier for each student.
- Using the student's name won't work because some people have the same name.
- A common solution is to have a large integer that gets incremented each time you create a new student.
- That integer would be stored as static variable (code on next slide)

## static fields

```
public class Students {
  private String name;
  private long studentID;
  private static long masterID = 1;

  public Student (String newName) {
    name = newName;
    studentID = masterID++;  //assigns student id and inc master
  }
. . .
```

## final fields

- A field can be declared final.
  - A final instance field must be initialized when the object is constructed and afterwards can never be modified.
  - An example might be the student id from the previous slide. A student id would presumable never change once set, so we could have written:

```
public class Students {
  private String name;
  private final long studentID;
  private static long masterID = 1;

  public Student (String newName) {
    name = newName;
    studentID = masterID++;
  }
```

## final fields

- Note that a final field can have an accessor method, but not a mutator method.
- Note that we didn't use all caps for final instance fields in the last example since fields tied to different objects will have different values.
- It can be very confusing to have a reference variable to a mutable class.
- A final static field is called a *static constant*.
  - This will have a constant value for the life of the program.
  - An example is the Math class which has a static final of PI

```
System.out.println(Math.PI); // prints 3.14159265358979323846
```

- Note we do use the all caps notation for static final fields.

## static methods

- A static method is one that is tied to a class instead of an object.
  - This means the class does not have to be instantiated for the method to be called.
  - Recall an instance method is called like:

    ```
    object.method(arguments);
    ```

  - for example:

```
String str = “The Beatles”;
System.out.println(str.toLower());
```

  - A static method is called like:

    ```
    class.method(arguments);
    ```

  - for example:

    ```
    double x = Math.sqrt(y); // Math is a class
    ```

- **`the main` method**:
  ```
  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.
    - Since the code in **`main`** is the first code executed, this is important. (What code would have created the object containing **`main`**?)
  - **`void`** means **`main`** does not return a value.
  - **`main`** is passed an array of strings.
  - Every class can have a **main** method, but it will only be called if the JVM is invoked with that class.

## The *toString* Method:

- Returns a **`String`** representing the data of an object.
- Example:

```
BaseballStats rookie, veteran;

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

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

Clients can call **`toString`** explicitly by coding the method call.

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

Clients can call **`toString`** implicitly by using an object reference where a **`String`** is expected.

  - Both print's produce the <u>same</u> output.

- The **`toString`** method is a great way to check the correctness of a program!
- So, what does the method look like?

- The **`toString`** method for any class always begins with:
  **`public String toString() {`**
  - The method is **`public`** so it can be called from outside the class.
  - The method returns a **`String`**.
  - The method has <u>no</u> 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:

```
public String toString() {
    int playerHits = singles + doubles + triples + homeRuns;
    return playerName + " has " + playerHits + " hits.";

 }
```

**Or**

```
 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`**.

- Here's an example for **`toString`** for **`BaseballStats`**:

```
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;
 }
```

## 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.,

```
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");
```

- 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`**.

- For the **BasebalIStats** class:

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

```
public boolean equals(BaseballStats otherPlayer) {
  if ( playerName.equals( otherPlayer.playerName ) &&
       singles == otherPlayer.singles &&
       doubles == otherPlayer.doubles &&
       triples == otherPlayer.triples &&
       homeRuns == otherPlayer.homeRuns &&
       walks == otherPlayer.walks &&
       outs == otherPlayer.outs &&
       rbis == otherPlayer.rbis )

     return true;

  else
     return false;
}
```

Notice that even though the variables are **private**, values inside **otherPlayer** can still be accessed. This is because equals is in the same class as **otherPlayer**.

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

- An alternative way to write the equals method for **`BaseballStats`**:

```
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() );
}
```

## Using of Multiple Source Files

- You can define more than one class in a single file, though only one can be public.
  - These classes can use each other.
- You can also write the classes in different files.
  - When compiling a program that uses another class, the compiler will look for the source code for that class (in the current directory by default)
  - For example, your code in section had a source file for **`Driver`**, which used **`MyLinkedList`**, which in turn might have used another class.
    - In this case, when you use the command **`javac Driver.java`**
      1. The compiler will see the **`MyLinkedList`** class is being used.
      2. It will look for files called **`MyLinkedList.class`** and **`MyLinkedList.java`**.
      3. If it finds **`MyLinkedList.java`** and **`MyLinkedList.class`** does not exist or is older than the **`MyLinkedList.java`** file then it will compile **`MyLinkedList.java`**

## Packages

- Java allows you to group classes into a collection called a *package*.
  - You have already used packages in the standard Java library: **`java.util`**, **`java.io`**
- One of the main reason for having packages is to guarantee uniqueness of class names.
  - For example, suppose someone else wrote a class called **`Node`**. How would you specify which one you wanted to use?
  - You could indicate which one by including the package it is in.

## Packages

- A class can use all the classes from its own package and all public classes from other packages.
- You can access classes in other packages one of two ways:
  1. You can specify the full package name every time you access the class. e.g.

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

  2. You can use the **`import`** statement to import a class. e.g.

     ```
     import java.util.Scanner;
     . . .   //code here
     Scanner in = new Scanner()
     ```

- As we've seen, you can also import all the classes in a package using *. e.g. **`import java.util.*;`**

## **Packages**

- Packages are organized in hierarchies.
- We've seen **`java.util`** and **`java.io`**
- You could go deeper for example a package may be called **`arizona.cs.anson.lectures`**
- While the hierarchy is useful for organization (and we'll see with your own classes it dictates where they must be stored), but from the point of the view of the compiler it does not specify a relationship.
  - For example, packages **`arizona.cs.anson`** and **`arizona.cs.anson.lectures`** have no relationship to each other.

## Packages

- This means that while you can use:

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

  To import all the classes in the java.util package and

```
   import java.io.*;
```

  - To import all the classes in the java.io package,
- You can NOT use:

```
import java.*;
```

  - To import all the classes in both the java.util and java.io packages.
  - This command would import all the classes in the java package (if there were such a package) which has no relation to java.io or java.util

## Packages

- Packages are used to resolve name conflicts, but what if you include classes with the same name?
- For example, both the packages **`java.util`** and **`java.sql`** contain a class called **`Date`**.
- If your program imports both packages and tries to use Date e.g.

```
import java.util.*;
import java.sql.*;
 . . . //bunch of code
Date today;
```

  - This will generate a compiler error because the compiler doesn't know which date to use.
- So how can we get around this?

## Packages (cont)

- In addition to specifying all the classes in a package, you can specify a particular class. e.g.

```
import java.util.*;
import java.sql.*;
import java.util.Date;
 . . . //bunch of code
Date today;  //compiler knows to use java.util.Date
```

  - This works since now the compiler knows which **`Date`** to use.
- What if you want to use both **`Dates`** in your program?
- You can always write out the full package name every time you use one.
  **`java.sql.Date today;`**

## Packages

- You can put a class inside a package by putting the name of the package at the top of your source file along with the **`package`** statement:

```
package arizona.anson.lectures;

public class MyClass {
   . . .
```

- Classes defined in source files that do not contain a package statement are located in the default package.
- The source files should be placed in a subdirectory structures that matches the package name. Each '.' in the package name represents a subdirectory.
- For the example above, the file should be placed in the directory: **`arizona/anson/lectures`**

## JAR Files

- Class files can be stored in JAR (Java ARchive) files.
- These files contain multiple class files and there directory structure in a compressed format.
- This is the usual form that Java programmers distribute classes to be used by other programmers.
- To use these classes the compiler and the JVM (Java Virtual Machine) must be told to look in these files.
- The *class path* tells these programs where to look for classes.

## The Class Path

- The class path tells Java compiler and the JVM where to look for classes.
- By default the class path is the current directory.
- The class path works a lot like the PATH in bash and it may contain several directories.
- For example the class path might look like:
  /home/usr/classdir:~/myClasses/anson.jar:.
- This contains the paths /home/usr/classdir , ~/myClasses/anson.jar and .
- Don't forget to include '.' the current directory in the path is you want it

## The Class Path

- You may set the class path by using the class path option for javac or java

```
javac -cp /home/usr/classdir:~/myClasses/anson.jar:. MyProg.java

java -cp /home/usr/classdir:~/myClasses/anson.jar:. MyProg
```

  - **`javac`** will actually always look in the current directory but the JVM will NOT if it is not in the class path.

- You may also set the CLASSPATH environment variable, but do it with caution.

```
export CLASSPATH=/home/usr/classdir:~/myClasses/anson.jar:.
```

  - It can cause problems if you forget the .

## BaseballStats Source Code:

```
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;
 }
```

```
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
```