Chapter 3: Using Classes and Objects
CS 121

Department of Computer Science
College of Engineering
Boise State University

April 2, 2015
Chapter 3 Topics

- Part 0: Intro to Object-Oriented Programming
  Go to part 0
- Part 1: Creating Java objects
  Go to part 1
- Part 2: The String class
  Go to part 2
- Part 3: The Random class
  Go to part 3
- Part 4: The Math class
  Go to part 4
- Part 5: Formatting output
  Go to part 5
- Part 6: Intro to enum
  Go to part 6
- Part 7: Wrapper classes and autoboxing
  Go to part 7
Brief Intro to Object-Oriented Programming

- Java is an **object-oriented** programming language.
- **Objects** are used to represent real-world things.
- **Objects** have **state** and **behaviors**.

  - **Dog**
    - state: name, breed, color, age, hungry, etc.
    - behavior: walk, run, bark, lick, fetch
  
  - **String** "Hello World!"
    - state: length, characters
    - behavior: get length, equals, sub-string, compare to, to upper case, etc.
Objects are defined by **classes**.

- Multiple objects can be created from the same class.
- Define **variables** to represent **state**.
- Define **methods** to define **behaviors**.
We can think of a class as the blueprint of an object.

One blueprint to create several similar, but different, houses.
An object is an *instance* of a class.

Objects are *encapsulated* to protect the data it manages.

Classes can be created based on another class using *inheritance* (You will see more of inheritance in CS 221).
Creating Objects

- **Recall**: A variable holds either a primitive type or a reference to an object.
- A variable referring to an object is known as a **reference variable**.
- The class name of the object is used as the type in the declaration statement.
  ```java
  String title;
  ```
- This declaration *does not create* an instance of the object.
- It is just a **reference** to an actual object stored in memory.
- The object must be *explicitly* created.
We use the `new` operator to create a new object.

\[
\text{String } \text{courseName} = \text{new String("CS 121");}
\]

This calls the String constructor – a special method that sets up the object.

The new object is an instance of the class.
A quick note.... We don’t have to use the `new` operator to create a String.

We can use string literals.

```java
String courseName = “CS 121”;
```

This is only supported for `String` objects (because they are so frequently used). The Java compiler creates the object for us as a convenience.
After we instantiate an object, we can use the dot operator to invoke its methods.

```java
String courseName = new String("CS 121");
int length = courseName.length();
```

Methods may return values that can be used in an assignment or expression.

Invoking an object’s method can be thought of as asking the object to do something.
Object References

- Primitive variables and object variables store different information.
- Primitive variables (e.g. `int`, `char`, `boolean`) contain the value itself.
- Object variables (e.g. `String`) contain the address of the object it references.
```java
int age = 52;
String name = new String("Bruce Banner");
String alterEgo = "The Hulk'';
double health = 100.0;
int hits = 0;
```
**Recall:** The act of assignment takes a copy of a value (the RHS) and stores it in the target variable (the LHS).

For primitive types, the *value* of the variable is copied.

\[
\begin{array}{c|c|c}
\text{Before:} & \text{hits} & \text{newHits} \\
& 0 & 5 \\
\hline
\text{Assignment:} & \text{hits = newHits;} \\
\hline
\text{After:} & \text{hits} & \text{newHits} \\
& 5 & 5
\end{array}
\]
Assignment Revisited

- For objects, the *address* of the object is copied.

**Before:**

- **name**
  - Address: 0x01
  - Value: "Bruce Banner"

- **alterEgo**
  - Address: 0x02
  - Value: "The Hulk"

**Assignment:**

```java
name = alterEgo;
```

**After:**

- **name**
  - Address: 0x01
  - Value: "Bruce Banner"

- **alterEgo**
  - Address: 0x02
  - Value: "The Hulk"
Two or more references that refer to the same object are aliases of each other.

A single object can be accessed using multiple references.

This is useful, but can cause issues if not managed properly.

Changing an object through one reference changes it for all of its aliases, because there is really only one object stored in memory.
Example: Swapping Two Variables

- Suppose we have two `int` variables that we want to swap. We need a temporary variable to complete the swap:

```java
int n1 = 100, n2 = 200;
int tmp = n1; n1 = n2; n2 = temp;
```

- Suppose we have two `String` variables that we want to swap. We need a temporary `String` variable to complete the swap. This is a good example of using aliases:

```java
String s1 = "hello";
String s2 = "goodbye";
String tmp = s1; s1 = s2; s2 = tmp;
```
Garbage Collection

- If there are no variables that refer to an object, the object is inaccessible and referred to as garbage.
- Java performs automatic garbage collection in the background, reclaiming the memory used by garbage objects for future use.
- In some languages, the programmer is responsible for freeing the memory used by garbage objects.
The Java API

- The Java API is the standard class library that provides a large collection of pre-built classes that we can use in our programs.
- API = Application Programming Interface
- Before writing our own classes, we will practice using several classes that are part of the Java API.
- The classes of the Java API are organized into packages. Java comes with hundreds of packages and tens of thousands more can be obtained from third-party vendors.
- Java API docs:
  http://docs.oracle.com/javase/8/docs/api/
## Selected Java Packages

<table>
<thead>
<tr>
<th>Package</th>
<th>Provides</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.lang</td>
<td>Fundamental classes</td>
</tr>
<tr>
<td>java.util</td>
<td>Various useful utility classes</td>
</tr>
<tr>
<td>java.io</td>
<td>Classes for variety of input/output functions</td>
</tr>
<tr>
<td>java.awt</td>
<td>Classes for creating graphical user interfaces and graphics</td>
</tr>
<tr>
<td>java.swing</td>
<td>Lightweight user interfaces that extend AWT capabilities</td>
</tr>
<tr>
<td>java.net</td>
<td>Networking operations</td>
</tr>
<tr>
<td>java.security</td>
<td>Encryption and decryption</td>
</tr>
</tbody>
</table>
Import Declarations

▶ When you want to use a class from a Java API package, you need to `import` the package.

   ```java
   import java.awt.Graphics;
   ```

▶ To import all classes in a package, you can use the wild card character (`*`).

   ```java
   import java.awt.*;
   ```

▶ All classes in the `java.lang` package are automatically imported into all programs.

   ▶ This includes `String` and `System` (among others)
The **Graphics** class from the `java.awt` package is a useful class for drawing shapes on a canvas.

- See the Intro to Graphics notes for details on how to use the **Graphics** class.
- [http://cs.boisestate.edu/~cs121/notes/graphics-handout.pdf](http://cs.boisestate.edu/~cs121/notes/graphics-handout.pdf)
The **String** Class

- In Java, strings are **immutable**: Once we create a `String` object, we cannot change its value or length.
- The **String** class provides several useful methods for manipulating `String` objects. Many of these return a new `String` object since strings are immutable. For example:
  ```java
  String babyWord = "googoo";
  String str = babyWord.toUpperCase();
  ```
- See javadocs for **String** for list of available methods:
  [http://docs.oracle.com/javase/8/docs/api/java/lang/String.html](http://docs.oracle.com/javase/8/docs/api/java/lang/String.html)
### Selected Methods in String class

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int length()</code></td>
<td>Returns the length of the string.</td>
</tr>
<tr>
<td><code>char charAt(int index)</code></td>
<td>Returns the character at the specified index.</td>
</tr>
<tr>
<td><code>String toLowerCase()</code></td>
<td>Converts the string to lowercase.</td>
</tr>
<tr>
<td><code>String toUpperCase()</code></td>
<td>Converts the string to uppercase.</td>
</tr>
<tr>
<td><code>String trim()</code></td>
<td>Removes leading and trailing whitespace from the string.</td>
</tr>
<tr>
<td><code>boolean equals(String str)</code></td>
<td>Returns true if the string is equal to another string, false otherwise.</td>
</tr>
<tr>
<td><code>boolean equalsIgnoreCase(String str)</code></td>
<td>Returns true if the string is equal to another string ignoring case.</td>
</tr>
<tr>
<td><code>int compareTo(String str)</code></td>
<td>Compares two strings and returns an integer.</td>
</tr>
<tr>
<td><code>String concat(String str)</code></td>
<td>Concatenates two strings.</td>
</tr>
<tr>
<td><code>String replace(char oldChar, char newChar)</code></td>
<td>Replaces all occurrences of <code>oldChar</code> with <code>newChar</code> in the string.</td>
</tr>
<tr>
<td><code>String substring(int offset, int endIndex)</code></td>
<td>Returns a string that equals the substring from index <code>offset</code> to <code>endIndex</code> - 1</td>
</tr>
<tr>
<td><code>int indexOf(char ch)</code></td>
<td>Returns the index of the first occurrence of <code>ch</code> in the string.</td>
</tr>
<tr>
<td><code>int indexOf(String str)</code></td>
<td>Returns the index of the first occurrence of <code>str</code> in the string.</td>
</tr>
</tbody>
</table>

This table lists some of the most commonly used methods in the `String` class. Each method is briefly described, and some examples are provided to illustrate their usage.
The **String** class represents a string internally as a series of characters. These characters have an **index** that we can use to refer to a specific character.

We can use the `charAt(int index)` method to get the character at the index position.

```java
char ch = babyWord.charAt(0);
char ch = babyWord.charAt(4);
```
String Examples

- Example: StringPlay.java.

- What output is produced by the following code?

```java
String babyWords = "googoo gaagaa";
System.out.println(babyWords.length());
System.out.println(babyWords.toUpperCase());
System.out.println(babyWords.substring(7, 10));
System.out.println(babyWords.replace('g', 'm'));
System.out.println(babyWords.length());
```
The Random Class

- The `Random` class provides methods that generate pseudorandom numbers. The class is part of the `java.util` package.
- **True random numbers** are usually generated from nature or physical processes.
- Give some examples of physical processes that generate random numbers:
  - Flipping a coin
  - Rolling dice
  - Shuffling playing cards
  - Brownian motion of molecules in a liquid

- Pseudorandom numbers are generated using algorithms that start with a **seed** value. The values generated pass statistical tests. There are two main advantages of pseudorandom numbers:
  - Unlimited supply
  - Reproducibility

- Random numbers are used in simulations, security, testing software, design, games and many other areas.
Selected Methods in the `Random` class

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Random Random()</code></td>
<td>Constructor: creates a new pseudorandom generator.</td>
</tr>
<tr>
<td><code>Random Random(long seed)</code></td>
<td>Constructor: with a seed value to be able to reproduce random sequence.</td>
</tr>
<tr>
<td><code>int nextInt(int bound)</code></td>
<td>returns a random number over the range 0 to <code>bound-1</code>.</td>
</tr>
<tr>
<td><code>int nextInt()</code></td>
<td>returns a random number over all possible values of <code>int</code>.</td>
</tr>
<tr>
<td><code>double nextDouble()</code></td>
<td>returns a <code>double</code> random number between 0.0 (inclusive) and 1.0 (exclusive).</td>
</tr>
</tbody>
</table>
Using the **Random** Class

- Import the class, construct an instance and then use the appropriate methods.

  ```java
  import java.util.Random;
  Random generator = new Random();
  System.out.println(generator.nextInt(10));
  System.out.println(generator.nextInt(10));
  ```

- Use the constructor with a seed argument to create a pseudorandom number sequence that is the same each time:

  ```java
  import java.util.Random;
  long seed = 12345; //arbitrary number!
  Random generator = new Random(seed);
  System.out.println(generator.nextInt(10));
  System.out.println(generator.nextInt(10));
  ```

- Example: **RandomNumbers.java**
In-class Exercises

- Given a `Random` object named `generator`, what range of values are produced by the following expressions?

  ```java
  generator.nextInt(25)
  generator.nextInt(10) + 1
  generator.nextInt(50) + 100
  generator.nextInt(10) - 5
  generator.nextInt(21) - 10
  ```

- Write an expression using `generator` that produces the following range of random values:

  - 0 to 12
  - 1 to 100
  - 15 to 20
  - -10 to 0

- Create a random color using the `Color` class and the `Random` class.
The **Math** class contains methods for basic mathematical operations like exponentiation, square root, logarithm and trigonometric functions.

- Part of the `java.lang` package so no need to import.
- The methods in the **Math** class are static methods (also known as class methods).
- Static methods can be invoked using the class name — no **Math** object needs to be instantiated. For example:
  ```java
double value = Math.sin(Math.PI) + Math.cos(Math.PI);
```
- Example: Quadratic.java, TrigDemo.java.
Selected Methods in the \textbf{Math} class

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>static int abs(int num)</td>
<td>Calculates the absolute value of an integer</td>
</tr>
<tr>
<td>static double sqrt(double num)</td>
<td>Calculates the square root of a number</td>
</tr>
<tr>
<td>static double ceil(double num)</td>
<td>Rounds a number up to the nearest integer</td>
</tr>
<tr>
<td>static double floor(double num)</td>
<td>Rounds a number down to the nearest integer</td>
</tr>
<tr>
<td>static double log(double num)</td>
<td>Calculates the natural logarithm of a number</td>
</tr>
<tr>
<td>static double log10(double num)</td>
<td>Calculates the base-10 logarithm of a number</td>
</tr>
<tr>
<td>static double pow(double num, double power)</td>
<td>Raises a number to a power</td>
</tr>
<tr>
<td>static double min(double num1, double num2)</td>
<td>Returns the minimum of two numbers</td>
</tr>
<tr>
<td>static double max(double num1, double num2)</td>
<td>Returns the maximum of two numbers</td>
</tr>
<tr>
<td>static int min(int num1, int num2)</td>
<td>Returns the minimum of two integers</td>
</tr>
<tr>
<td>static int max(int num1, int num2)</td>
<td>Returns the maximum of two integers</td>
</tr>
<tr>
<td>static double sin(double angle)</td>
<td>Calculates the sine of an angle</td>
</tr>
<tr>
<td>static double cos(double angle)</td>
<td>Calculates the cosine of an angle</td>
</tr>
<tr>
<td>static double tan(double angle)</td>
<td>Calculates the tangent of an angle</td>
</tr>
<tr>
<td>static double toRadians(double angleInDegrees)</td>
<td>Converts degrees to radians</td>
</tr>
<tr>
<td>static double toDegrees(double angleInRadians)</td>
<td>Converts radians to degrees</td>
</tr>
</tbody>
</table>
The `java.text` package provides classes to format values for output.

- The `NumberFormat` allows us to format values as currency or percentage.
- The `DecimalFormat` allows us to format values based on a pattern.

Two code code snippets that shows the usage of `NumberFormat` class (Note that the import statement will be at the top of the Java source file):

```java
import java.text.NumberFormat;
NumberFormat fmt1 = NumberFormat.getCurrencyInstance();
double amount = 1150.45;
System.out.println("Amount: " + fmt1.format(amount));

import java.text.NumberFormat;
NumberFormat fmt2 = NumberFormat.getPercentInstance();
double passRate = .8845;
System.out.println("Amount: " + fmt2.format(passRate));
```

Example: `Purchase.java`
The `DecimalFormat` allows us to format values based on a pattern.

- For example, we can specify the number should be rounded to three digits after the decimal point.
- Uses Half Even Rounding to truncate digits: round towards the “nearest whole neighbor” unless both whole neighbors are equidistant, in which case, round towards the even neighbor. See here for details: Half Even Rounding in Java API

A code snippet that shows the usage:

```java
import java.text.DecimalFormat;
DecimalFormat fmt = new DecimalFormat("0.###");
double area = 110.3424;
System.out.println("Amount: " + fmt.format(amount));
//shows 110.342
```

- We can change the rounding mode with the `setRoundingMode` method:

```java
fmt.setRoundingMode(RoundingMode.CEILING);
System.out.println("Amount: " + fmt1.format(amount));
//shows 110.343
```
Commonly used symbols in the pattern:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>digit ((\text{int, short, byte}))</td>
</tr>
<tr>
<td>#</td>
<td>digit, zero shows as absent</td>
</tr>
<tr>
<td>.</td>
<td>decimal separator</td>
</tr>
<tr>
<td>,</td>
<td>grouping separator (for large numbers)</td>
</tr>
<tr>
<td>E</td>
<td>show in scientific notation</td>
</tr>
</tbody>
</table>

Example: \texttt{CircleStatsDecimalFormat.java}

\textbf{In-class exercise} What do the following patterns accomplish?

```
"##.###"
"00.###"
"###,###"
"000,000"
```

We can set minimum and maximum limits on integer and fractional digits. For more information, see the javadocs for the \texttt{DecimalFormat} class.
The class **Formatter** from the **java.util** package provides an alternative way of formatting output that is inspired by the **printf** method in C language.

```java
import java.util.Formatter;
Formatter fmt = new Formatter(System.out);
double area = 1150.45;
fmt.format("The area is %f\n",area);
```

Here the %f is a **conversion** template that says to format the variable area as a floating point number and insert in the output. Various conversions are available for printing a wide variety of types.

Convenience methods exist in the **System.out** object to use **Formatter** class methods.

```java
System.out.printf("The area is %f\n",area);
```

We can also format a **String** object, which often comes in handy.

```java
String output = String.format("The area is %f\",area);
```

In each case, the underlying method used is the same.
Selected printf Style Formatting Conversions

- Commonly used conversions:

<table>
<thead>
<tr>
<th>Conversion</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%d</td>
<td>decimal (int, short, byte)</td>
</tr>
<tr>
<td>%ld</td>
<td>long</td>
</tr>
<tr>
<td>%f</td>
<td>floating point (float, double)</td>
</tr>
<tr>
<td>%e</td>
<td>floating point in scientific notation</td>
</tr>
<tr>
<td>%s</td>
<td>String</td>
</tr>
<tr>
<td>%b</td>
<td>boolean</td>
</tr>
</tbody>
</table>

- Some examples of variations on the default formatting:

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%10d</td>
<td>use a field 10 wide (right-aligned for numeric types)</td>
</tr>
<tr>
<td>%8.2f</td>
<td>use a field 8 wide, with two digits after the decimal point</td>
</tr>
<tr>
<td>%-10s</td>
<td>left justified string in 10 spaces (default is right justified)</td>
</tr>
</tbody>
</table>

- Note that if the output doesn’t fit in the number of spaces specified, the space will expand to fit the output.

- Examples: *CircleStatsFormatter.java*, *CircleStatsPrintfTable.java*, *PrintfExample.java*
An Enumerated type is a set of values or elements that behave as constants. For example:

```java
public enum Season {WINTER, SPRING, SUMMER, FALL};
public enum Suit {CLUBS, DIAMONDS, HEARTS, SPADES};
```

The enumeration lists all possible values for the specified type.

Now we can declare a variable of the enumerated type and assign it values:

```java
Suit card = Suit.DIAMONDS;
```

Enumerated values are type-safe, that is, only valid assignments are the listed values.

**In-Class Exercise: EX 3.12.** Write a declaration for an enumerated type that represents the days of the week. Declare a variable of this type and assign it a day of the week.
Internally, each value of an enumerated type is stored as an integer, called its **ordinal value**. We can access the ordinal value with the `ordinal()` method.

```java
System.out.println(card.ordinal());
```

The first value in an enumerated type has an ordinal value of zero, the second one, and so on.

However, we cannot assign a numeric value to an enumerated type, even if it corresponds to a valid ordinal value.

Example: `SuitTest.java`, `IceCream.java`

Note that `RoundingMode` that we used earlier is an `enum` defined in the `java.math` package.
Wrapper Classes (1)

- The `java.lang` package contains wrapper classes corresponding to each primitive type.

<table>
<thead>
<tr>
<th>primitive</th>
<th>wrapper</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>Byte</td>
</tr>
<tr>
<td>short</td>
<td>Short</td>
</tr>
<tr>
<td>int</td>
<td>Integer</td>
</tr>
<tr>
<td>long</td>
<td>Long</td>
</tr>
<tr>
<td>float</td>
<td>Float</td>
</tr>
<tr>
<td>double</td>
<td>Double</td>
</tr>
<tr>
<td>char</td>
<td>Character</td>
</tr>
<tr>
<td>boolean</td>
<td>Boolean</td>
</tr>
<tr>
<td>void</td>
<td>Void</td>
</tr>
</tbody>
</table>

- See below for the relationship between the wrapper object and the primitive type:

- An object of a wrapper class can be used any place where we need to store a primitive value as an object.
The wrapper classes contain useful static methods as well as constants related to the base primitive type.

For example, the minimum int value is `Integer.MIN_VALUE` and the maximum int value is `Integer.MAX_VALUE`.

Example: `PrimitiveTypes.java`

For example, the `parseInt` method converts an integer stored as a `String` into an int value. Here is a typical usage to convert input from a user to an integer.

```java
Scanner scan = new Scanner(System.in);
String input = scan.nextLine();
int num = Integer.parseInt(input);
```
Wrapper Classes (3)

- Selected methods from the `Integer` class.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Integer(int value)</code></td>
<td>Constructor: builds a new <code>Integer</code> object that stores the specified value.</td>
</tr>
<tr>
<td><code>static parseInt(String s)</code></td>
<td>Returns an <code>int</code> value corresponding to the value stored in the string <code>s</code>.</td>
</tr>
<tr>
<td><code>static toBinaryString(int i)</code></td>
<td>Returns the string representation of integer <code>i</code> in the corresponding base.</td>
</tr>
<tr>
<td><code>static toOctalString(int i)</code></td>
<td>Returns the string representation of integer <code>i</code> in the corresponding base.</td>
</tr>
<tr>
<td><code>static toHexString(int i)</code></td>
<td>Returns the string representation of integer <code>i</code> in the corresponding base.</td>
</tr>
</tbody>
</table>

- Similar methods and many more are available for all the wrapper classes. Explore the javadocs for the wrapper classes.
Autoboxing

- **Autoboxing** is the automatic conversion of a primitive value to a corresponding wrapper object.

```java
Integer obj;
int num = 100;
obj = num;
```

- The assignment creates the corresponding wrapper `Integer` object. So it is equivalent to the following statement.

```java
obj = new Integer(num);
```

- The reverse conversion (**unboxing**) also happens automatically as needed.
Summary

- Understand the difference between primitive type variables and reference variables.
- Creating and using objects.
- Using `String`, `Math`, `Random`, `Scanner` classes.
- Formatting output using `NumberFormat`, `DecimalFormat` and `Formatter` classes.
- Using `enum` types to define a collection of constants.
- Wrapper classes and autoboxing: `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Character`, `Boolean`
Exercises

- Read Chapter 3.
- **Recommended Homework:**
  - Projects: PP 3.2, 3.3, 3.5.