Chapter 3: Using Classes and Objects
CS 121

Department of Computer Science
College of Engineering
Boise State University

January 3, 2017
Chapter 3 Topics

- Intro to Object-Oriented Programming
- Using Classes and Objects
- The Graphics class
- The Scanner class
- The Random class
- Formatting output
- The Math class
- The String class
- Wrapper classes and autoboxing
Brief Intro to Object-Oriented Programming

- Java is an **object-oriented** programming language.
- **Objects** can be 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.

**Variables** represent the object’s **state** (attributes, properties).

**Methods** define **behaviors** (functions, actions).
Classes and Objects

- 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, meaning the state and behaviours are wrapped together as a single unit. Encapsulation is one of the four fundamental Object-Oriented Programming (OOP) concepts. The other three are inheritance, polymorphism, and abstraction. (You will see these in CS 221).
Encapsulation

- A visual example of encapsulation of the `String` object.

  **State:**
  - Sequence of characters

  **Behavior:**
  - Get length
  - Convert to upper case
  - Convert to lower case

- An analogy of a “real-world” object that uses encapsulation.
For now, we will be *using* classes that other people have created.

After we grasp the basic concepts of how to use them, then we will see how to write our own classes.
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>
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)
We must **declare** and **initialize** our objects before we can use them.

This is very similar to what we did with primitive data type variables, except we must also **instantiate** our objects.

```java
int courseNumber = 121;
Color grass = new Color(0, 255, 0);
```

For example, we use the `new` operator to instantiate a new `Scanner` object, which is an instance of the `Scanner` class.

If we don’t use the `new` operator, and just declare an object, this does not create an instance of the class.

```java
Color color;  // this variable refers to // nothing (aka. null)!!
```
Instantiating an Object

String courseName = new String("CS 121");
    // more on Strings in a minute
Color grass = new Color(0, 255, 0);

- When we use the new operator, this calls the class constructor – a special method that sets up the object.
- The new object is an instance of the class “blueprint”.
We don’t have to use the `new` operator to create a String.

```java
String courseName = new String("CS 121");
// is the same as
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.
Reference Variables vs. Primitive Variables

- Primitive variables and object variables store different information.

- Primitive variables (e.g. int, char, boolean) contain the value itself.

  ```java
  int courseNumber = 121;
  ```

- The variable referring to an object is known as a reference variable.

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

- It holds the address (aka. reference) of where the actual object data is stored in memory. We sometimes say it “points to” the object.
Reference Variables: The Hulk

```java
int age = 52;
String name = new String("Bruce Banner");
String alterEgo = "The Hulk";
double health = 100.0;
int hits = 0;
```
Assignment Revisited

- **Recall:** The act of assignment takes a copy of a value (the Right-Hand-Side) and stores it in the target variable (the Left-Hand-Side).
- For primitive types, the value of the variable is copied.

```
Before:  
  hits | newHits
  0    | 5

Assignment: hits = newHits;

After:  
  hits | newHits
  5    | 5
```
Assignment Revisited

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

Before:
- `name`: address 0x01, "Bruce Banner"
- `alterEgo`: address 0x02, "The Hulk"

Assignment: `name = alterEgo;`

After:
- `name`: address 0x02, "Bruce Banner"
- `alterEgo`: address 0x02, "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.*
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.
After we instantiate an object, we can use the dot operator to invoke its methods (or behaviors).

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

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

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

int length = courseName.length();
// the value of length will be 6
```
Methods may also accept **parameters** that provide additional information that it may need to perform the requested behavior.

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

String newCourse = courseName.replace('1', '2');
// the value of newCourse will be "CS 222"

char first = courseName.charAt(0);
// the value of first will be 'C'
```
In-Class Exercise

- What is the difference between a **class** and an **object**?
- Objects are **encapsulated**. What does this mean?
- What does it mean to **instantiate** an object? Write one line of code to instantiate a `Scanner` to read from `System.in`.
- What is a **reference variable**?
- What is the value of a reference variable if the object it is supposed to refer to is not instantiated?
- How do you tell an object to perform an **action/behavior**?
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.
The Scanner class

- The first tool, or class, we will use from the Java API is the `Scanner`. It is used when we want our programs to interact with our users.
- The `Scanner` class is part of the `java.util` class library. It must be imported.
  
  ```java
  import java.util.Scanner;
  ```
- It provides methods for reading input values of various types.
- A `Scanner` object can read input from various sources (e.g. keyboard, file)
- See Java 8 API docs: [http://docs.oracle.com/javase/8/docs/api/java/util/Scanner.html](http://docs.oracle.com/javase/8/docs/api/java/util/Scanner.html)
Create a new `Scanner` object that reads from the keyboard.

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

- The `new` operator creates a new `Scanner` object.
- The `System.in` object represents keyboard input.
- After the object is created, we can invoke various input methods it provides.
Example

- **Example:** Convert TempConverter.java to interactive program.
- **Example:** Echo.java
By default, white space is used to separate input elements (called tokens).

White space includes:
- Space characters (‘ ’)
- Tab characters (‘	’)
- New line characters (‘\n’ and ‘\r’)

The next, nextInt, nextDouble, etc. methods of the Scanner class read and return the next input tokens.

See Scanner documentation for more details.
Example: GasMileage.java
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: **DiceRoll.java, RandomNumbers.java**
1. Given an `Random` object named `generator`, what range of values are produced by the following expressions?

   \[
   \text{generator.nextInt(25)} \\
   \text{generator.nextInt(10)} + 1 \\
   \text{generator.nextInt(50)} + 100 \\
   \text{generator.nextInt(10)} - 5 \\
   \text{generator.nextInt(21)} - 10
   \]

2. 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

3. Create a random color using the `Color` class and the `Random` class.

4. Create a random position within a grid of width, \( w \), and height, \( h \), using the `Random` class.
The `java.text` package provides several classes to format values for output.

In this course, we will focus on classes for formatting numbers.

- `NumberFormat`: formats numerical values (e.g. currency or percentage).
- `DecimalFormat`: (a sub-class of `NumberFormat`) formats decimal values based on a pattern.

The following code snippets that show the usage of the `NumberFormat` class.

The import statement will be at the top of the Java source file.

```java
import java.text.NumberFormat;

NumberFormat currFmt = NumberFormat.getCurrencyInstance();
double amount = 1150.45;
System.out.println("Amount: " + currFmt.format(amount));

NumberFormat percFmt = NumberFormat.getPercentInstance();
double passRate = .8845;
System.out.println("Amount: " + percFmt.format(passRate));
```

Example: `BasicNumberFormat.java, 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: [http://docs.oracle.com/javase/8/docs/api/java/math/RoundingMode.html](http://docs.oracle.com/javase/8/docs/api/java/math/RoundingMode.html)

A code snippet that shows the usage:

```java
import java.text.DecimalFormat;

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

fmt.setRoundingMode(RoundingMode.CEILING);
System.out.println("Amount: " + fmt1.format(amount)); // shows 110.343
```

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

Example: `BasicDecimalFormat.java`
Commonly used symbols in the pattern:

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>digit (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: CircleStatsDecimalFormat.java

We can set minimum and maximum limits on integer and fractional digits. For more information, see the javadocs for the DecimalFormat class.
In-Class Exercise

What do the following patterns accomplish?

1. "##.###"
2. "00.###"
3. "###,###"
4. "000,000"
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\n", 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>Conversion</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,
The **Math** Class

- The **Math** 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);
```
- Examples: **Quadratic.java**, **TrigDemo.java**
**Selected Methods in the Math class**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>static int <code>abs(int num)</code></td>
<td>Returns the absolute value of a number</td>
</tr>
<tr>
<td>static double <code>sqrt(double num)</code></td>
<td>Returns the square root of a number</td>
</tr>
<tr>
<td>static double <code>ceil(double num)</code></td>
<td>Returns the smallest integer greater than or equal to the number</td>
</tr>
<tr>
<td>static double <code>floor(double num)</code></td>
<td>Returns the largest integer less than or equal to the number</td>
</tr>
<tr>
<td>static double <code>log(double num)</code></td>
<td>Returns the natural logarithm of a number</td>
</tr>
<tr>
<td>static double <code>log10(double num)</code></td>
<td>Returns the base 10 logarithm of a number</td>
</tr>
<tr>
<td>static double <code>pow(double num, double power)</code></td>
<td>Returns <code>num</code> raised to the power of <code>power</code></td>
</tr>
<tr>
<td>static double <code>min(double num1, double num2)</code></td>
<td>Returns the smaller of two numbers</td>
</tr>
<tr>
<td>static double <code>max(double num1, double num2)</code></td>
<td>Returns the larger of two numbers</td>
</tr>
<tr>
<td>static int <code>min(int num1, int num2)</code></td>
<td>Returns the smaller of two integers</td>
</tr>
<tr>
<td>static int <code>max(int num1, int num2)</code></td>
<td>Returns the larger of two integers</td>
</tr>
<tr>
<td>static double <code>sin(double angleInRadians)</code></td>
<td>Returns the sine of an angle</td>
</tr>
<tr>
<td>static double <code>cos(double angleInRadians)</code></td>
<td>Returns the cosine of an angle</td>
</tr>
<tr>
<td>static double <code>tan(double angleInRadians)</code></td>
<td>Returns the tangent of an angle</td>
</tr>
<tr>
<td>static double <code>toRadians(double angleInDegrees)</code></td>
<td>Converts an angle from degrees to radians</td>
</tr>
<tr>
<td>static double <code>toDegrees(double angleInRadians)</code></td>
<td>Converts an angle from radians to degrees</td>
</tr>
</tbody>
</table>
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>int <code>length()</code></td>
<td></td>
</tr>
<tr>
<td>char <code>charAt(int index)</code></td>
<td></td>
</tr>
<tr>
<td>String <code>toLowerCase()</code></td>
<td></td>
</tr>
<tr>
<td>String <code>toUpperCase()</code></td>
<td></td>
</tr>
<tr>
<td>String <code>trim()</code></td>
<td></td>
</tr>
<tr>
<td>boolean <code>equals(String str)</code></td>
<td></td>
</tr>
<tr>
<td>boolean <code>equalsIgnoreCase(String str)</code></td>
<td></td>
</tr>
<tr>
<td>int <code>compareTo(String str)</code></td>
<td></td>
</tr>
<tr>
<td>String <code>concat(String str)</code></td>
<td></td>
</tr>
<tr>
<td>String <code>replace(char oldChar, char newChar)</code></td>
<td></td>
</tr>
<tr>
<td>String <code>substring(int offset, int endIndex)</code></td>
<td>returns a string that equals the substring from index <code>offset</code> to <code>endIndex - 1</code></td>
</tr>
<tr>
<td>int <code>indexOf(char ch)</code></td>
<td></td>
</tr>
<tr>
<td>int <code>indexOf(String str)</code></td>
<td></td>
</tr>
</tbody>
</table>

- `returns a string that equals the substring from index offset to endIndex - 1`
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);
```
Example: StringPlay.java

What output is produced by the following code?

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());
Wrapper Classes (1)

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

<table>
<thead>
<tr>
<th>Primitive Type</th>
<th>Wrapper Class</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:

```
int num = 100;
Integer obj = new Integer(num);
```

- 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);
```
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></td>
</tr>
<tr>
<td><code>static toHexString(int i)</code></td>
<td></td>
</tr>
</tbody>
</table>

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