Agenda

- Continue OO concepts
- We're going to hold off the GUI examples for Chapter 3 and 4 until next week; I'd like to finish the discussion on classes and objects first

The Random Class

- The Random class is part of the java.util package
- It provides methods that generate pseudorandom numbers
- A Random object performs complicated calculations based on a seed value to produce a stream of seemingly random values
- Let's try a quick example (more complex one on page 126)
The Math Class

- The Math class is part of java.lang.
- The Math class contains methods that perform various mathematical functions, including absolute value, square root, exponentiation, and trigonometric functions.
- The methods of the Math class are static methods (also called class methods).
- Static methods can be invoked through the class name itself – no object of the Math class is needed.

```java
value = Math.cos(90) + Math.sqrt(delta);
```

Formatting Output

- As HW#2 demonstrated, Java will use too many decimal places by default.
- The Java standard class library contains classes that provide formatting capabilities to customize output.
- The NumberFormat class allows you to format values as currency or percentages.
- The DecimalFormat class allows you to format values based on a pattern.
- Both are part of the java.text package.

NumberFormat

- The NumberFormat class has static methods that return a formatter object:
  ```java
  NumberFormat.getCurrencyInstance()
  NumberFormat.getPercentInstance()
  ```
- Each formatter object has a method called format that returns a String with the specified information in the appropriate format.
**DecimalFormat**

- The `DecimalFormat` class can be used to format a floating point value in various ways.
- For example, you can specify that the number should be truncated to three decimal places.
- The constructor of the `DecimalFormat` class takes a string that represents a pattern for the formatted number.
- For HW#2, you could have used:
  ```java
  DecimalFormat df = new DecimalFormat("0.##")
  ```

**Wrapper Classes**

- Sometimes, we'll want to store the primitive objects inside a reference object.
- We'll learn how to make our own objects, but Java provides *wrapper classes* for this purpose.
- These wrapper classes also have a number of useful utility methods and attributes that work with the corresponding primitive type.
- Simple mnemonic: “capital form” of existing 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>
Creating a wrapper class object

- The following declaration creates an `Integer` object which represents the integer 40 as an object
  ```java
  Integer age = new Integer(40);
  ```
- An object of a wrapper class can be used in any situation where a primitive value will not suffice
- For example, some objects serve as containers of other objects
  - We’ll see this later in the semester

Utility methods in wrapper classes

- Wrapper classes also contain static methods that help manage the associated type
- For example, the `Integer` class contains a method to convert an integer stored in a `String` to an `int` value:
  ```java
  num = Integer.parseInt(str);
  ```
- The wrapper classes often contain useful constants as well
  - `MIN_VALUE` and `MAX_VALUE`, which hold the smallest and largest int values

Autoboxing

- Java 1.5 can auto-convert between a value of a primitive type and that of a reference type; this is called autoboxing
  ```java
  Integer obj;
  int num = 42;
  obj = num;
  ```
- The reverse conversion (called unboxing) also occurs automatically as needed
Writing our own classes

- The programs we’ve written in previous examples have used classes defined in the Java standard class library.
- Now we will begin to design programs that rely on classes that we write ourselves.
- The class that we’ve written so far contain only one method –– main –– and that’s just the starting point of a program.
- True object-oriented programming is based on defining classes that represent objects with well-defined characteristics and functionality.
  - Becoming an “expert” takes time.

Book’s example: Die

- Objects have state and behavior.
- Consider a six-sided die (singular of dice).
  - State: what’s the currently rolled value of the die?
  - Behavior: It can be (re-)rolled.
- We can represent a die in software by designing a class called Die that models this state and behavior.
  - The class serves as the blueprint for a die object.
- We can then instantiate as many die objects as we need for any particular program.

Classes

- A class can contain data declarations (state) and method declarations (functionality).
- So far, we haven’t used any data declarations, and only one method declaration: main.
- For our Die class, we can declare an int that represents the current value showing on the face.
- One of the methods would “roll” the die by setting that value to a random number between one and six.
Instance Variables

- We can not only create a variable inside a method, but *outside* it; these are called *instance variables*.
- Each object based on a class can have different values in these instance variables.
- Goal: for every Die that we create, we want an individual value for the die.
- The objects of a class share the method definitions, but each object has its own data space.

Instance Data

- If we have two Die objects in a DieRoller class, it might look like the following:

```
  die1 --> value: 5
  die2 --> value: 2
```

Method Declarations

- A method begins with a *declaration*, followed by code that will be executed when the method is *invoked* (called).
- When a method is invoked, the flow of control jumps to the method and executes its code.
- When complete, the flow returns to the place where the method was called and continues.
- The invocation may or may not *return* a value, depending on how the method is defined.
- Think of methods as (complex) mathematical functions.
Method Header

- A method declaration begins with a **method header**

```
int add (int num1, int num2)
```

- The parameter list specifies the type and name of each parameter
- The name of a parameter in the method declaration is called a **formal parameter**

Method Body

- The method header is followed by the **method body**

```
int add (int num1, int num2)
{
    int sum = num1 + num2;
    return sum;
}
```

- The return expression must be consistent with the return type
- `sum` is local data; it’s created each time the method is called, and is destroyed when it finishes executing

The return Statement

- The **return type** of a method indicates the type of value that the method sends back to the calling location
- A method that does not return a value has a **void** return type
- A **return statement** specifies the value that will be returned

```
return expression;
```

- Its expression must conform to the return type
Parameters

- When a method is called, the actual parameters in the invocation are copied into the formal parameters in the method header.

```java
result = obj.add(25, 30);
int add(int num1, int num2)
{
    int sum = num1 + num2;
    return sum;
}
```

Local Data

- As we’ve seen, local variables can be declared inside a method.
- The formal parameters of a method create automatic local variables when the method is invoked.
- When the method finishes, all local variables are destroyed (including the formal parameters).
- Keep in mind that instance variables, declared at the class level, exists as long as the object exists.

Constructors

- As mentioned previously, a constructor is a special method that is used to set up an object when it is initially created from a class.
- A constructor has the same name as the class, and no return type.
- Major bug alert!
- You can have a constructor set a default value or have it take an initial value for an object.
Defining the Die class

- Goal: design the `Die` class with other data and methods to make it a versatile and *reusable* resource
- That said, it doesn’t mean a program *has* to use all the features of a class
- Let’s write out what a possible Die class might be:
  - An int that has the face value
  - Methods to roll and set the die explicitly
  - Methods to get info on the die’s current value

Next time

- Continue Java OO concepts