CSC 2014  Java Bootcamp

Lecture 5
Writing Classes & Methods
CREATING CLASSES
Classes and Objects

Recall that an object has *state* and *behavior*

Consider a six-sided die (singular of dice)

- It’s state can be defined as which face is showing
- It’s primary behavior is that it can be 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
A class can contain data declarations and method declarations

```java
int size, weight;
char category;
```

- Data declarations
- Method declarations
Classes

The values of the data define the state of an object created from the class.

The functionality of the methods define the behaviors of the object.

For our Die class, we might declare an integer 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.
The Die Class

The Die class contains two data values:

- A constant \texttt{MAX} that represents the maximum face value.
- An integer \texttt{faceValue} that represents the current face value.

The \texttt{roll} method uses the \texttt{random} method of the \texttt{Math} class to determine a new face value.

There are also methods to explicitly set and retrieve the current face value at any time.

See Refactor Example: Dice
The toString Method

All classes that represent objects should define a toString method.

The toString method returns a character string that represents the object in some way.

It is called automatically when an object is concatenated to a string or when it is passed to the println method.
Constructors

As mentioned previously, a constructor is a special method that is used to set up an object when it is initially created.

A constructor has the same name as the class.

The `Die` constructor is used to set the initial face value of each new die object to one.

We examine constructors in more detail later.
Data Scope

The *scope* of data is the area in a program in which that data can be referenced (used)

Data declared at the class level can be referenced by all methods in that class

Data declared within a method can be used only in that method

Data declared within a method is called *local data*

In the `Die` class, the variable `result` is declared inside the `toString` method -- it is local to that method and cannot be referenced anywhere else
The `faceValue` variable in the `Die` class is called *instance data* because each instance (object) that is created has its own version of it.

A class declares the type of the data, but it does not reserve any memory space for it.

Every time a `Die` object is created, a new `faceValue` variable is created as well.

The objects of a class share the method definitions, but each object has its own data space.

That's the only way two objects can have different states.
Instance Data

We can depict the two Die objects from the RollingDice program as follows:

Each object maintains its own faceValue variable, and thus its own state.
Encapsulation

An encapsulated object can be thought of as a *black box* -- its inner workings are hidden from the client.

The client invokes the interface methods of the object, which manages the instance data.
METHODS
Visibility Modifiers

In Java, we accomplish encapsulation through the appropriate use of *visibility modifiers*.

A *modifier* is a Java reserved word that specifies particular characteristics of a method or data.

We've used the `final` modifier to define constants.

Java has three visibility modifiers: public, protected, and private.

The `protected` modifier involves inheritance, which we will discuss later.
Visibility Modifiers

Members of a class that are declared with *public visibility* can be referenced anywhere.

Members of a class that are declared with *private visibility* can be referenced only within that class.

Members declared without a visibility modifier have *default visibility* and can be referenced by any class in the same package.
Visibility Modifiers

Public variables violate encapsulation because they allow the client to “reach in” and modify the values directly.

Therefore instance variables should not be declared with public visibility.

It is acceptable to give a constant public visibility, which allows it to be used outside of the class.

Public constants do not violate encapsulation because, although the client can access it, its value cannot be changed.
Visibility Modifiers

Methods that provide the object's services are declared with public visibility so that they can be invoked by clients.

Public methods are also called *service methods*.

A method created simply to assist a service method is called a *support method*.

Since a support method is not intended to be called by a client, it should not be declared with public visibility.
Visibility Modifiers

<table>
<thead>
<tr>
<th></th>
<th>public</th>
<th>private</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variables</td>
<td>Violate encapsulation</td>
<td>Enforce encapsulation</td>
</tr>
<tr>
<td>Methods</td>
<td>Provide services to clients</td>
<td>Support other methods in the class</td>
</tr>
</tbody>
</table>
Accessors and Mutators

Because instance data is private, a class usually provides services to access and modify data values.

An *accessor method* returns the current value of a variable.

A *mutator method* changes the value of a variable.

The names of accessor and mutator methods take the form `getX` and `setX`, respectively, where `X` is the name of the value.

They are sometimes called “getters” and “setters.”
Mutator Restrictions

The use of mutators gives the class designer the ability to restrict a client’s options to modify an object’s state.

A mutator is often designed so that the values of variables can be set only within particular limits.

For example, the setFaceValue mutator of the Die class should have restricted the value to the valid range (1 to MAX).
Let’s now examine method declarations in more detail.

A *method declaration* specifies the 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.
Method Control Flow

If the called method is in the same class, only the method name is needed
The called method is often part of another class or object
A method declaration begins with a *method header*

```
char calc (int num1, int num2, String message)
```

- **Return type**: `char`
- **Method name**: `calc`
- **Parameter list**: `int num1, int num2, String message`

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*.
The method header is followed by the method body

```java
char calc (int num1, int num2, String message) {
    int sum = num1 + num2;
    char result = message.charAt (sum);

    return result;
}
```

The return expression must be consistent with the return type. **sum** and **result** are local data. They are created each time the method is called, and are 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
char ch = obj.calc(25, count, "Hello");
```

```java
char calc(int num1, int num2, String message) {
    int sum = num1 + num2;
    char result = message.charAt(sum);
    return result;
}
```
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.
The this Reference

The this reference allows an object to refer to itself

That is, the this reference, used inside a method, refers to the object through which the method is being executed

Suppose the this reference is used in a method called `tryMe`, which is invoked as follows:

```java
obj1.tryMe();
obj2.tryMe();
```

- In the first invocation, the this reference refers to `obj1`; in the second it refers to `obj2`
The this reference

The *this* reference can be used to distinguish the instance variables of a class from corresponding method parameters with the same names.

The constructor of the `Account` class could have been written as follows:

```java
public Account (String name, long acctNumber,
               double balance)
{
    this.name = name;
    this.acctNumber = acctNumber;
    this.balance = balance;
}
```
Inheritance

Inheritance allows a software developer to derive a new class from an existing one.

The existing class is called the parent class, or superclass, or base class.

The derived class is called the child class or subclass.

As the name implies, the child inherits characteristics of the parent.

That is, the child class inherits the methods and data defined by the parent class.