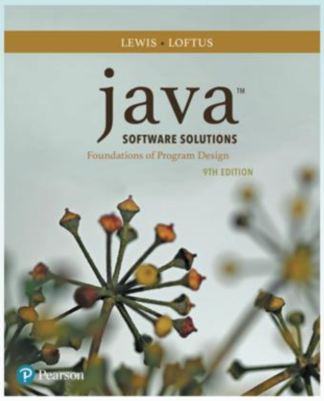
Chapter 4
Writing Classes

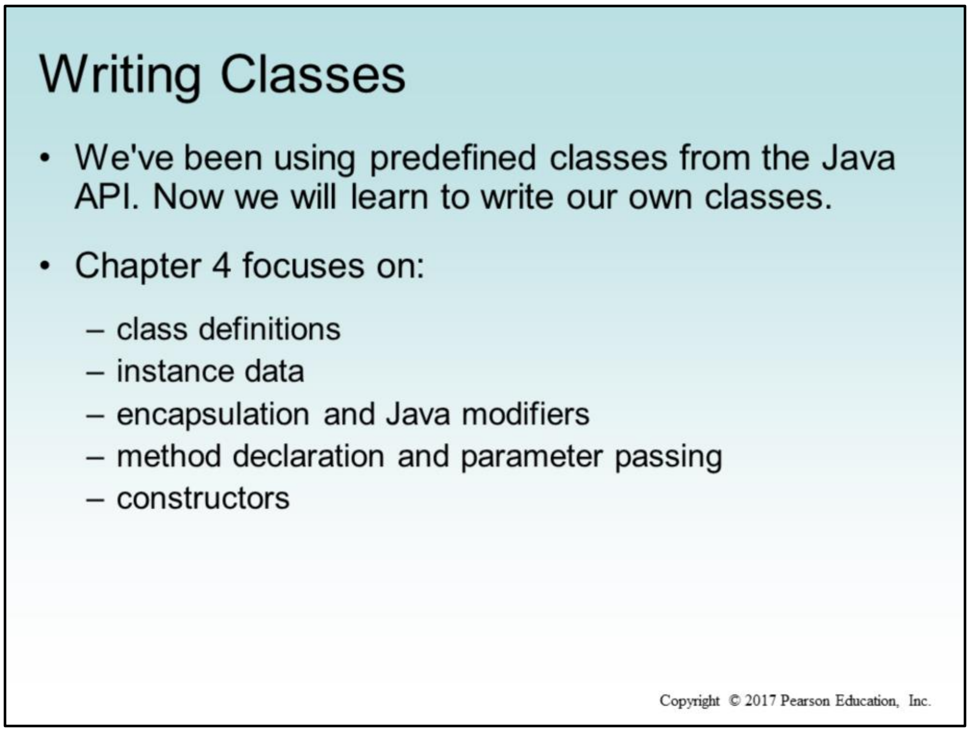


Java Software Solutions

Foundations of Program Design

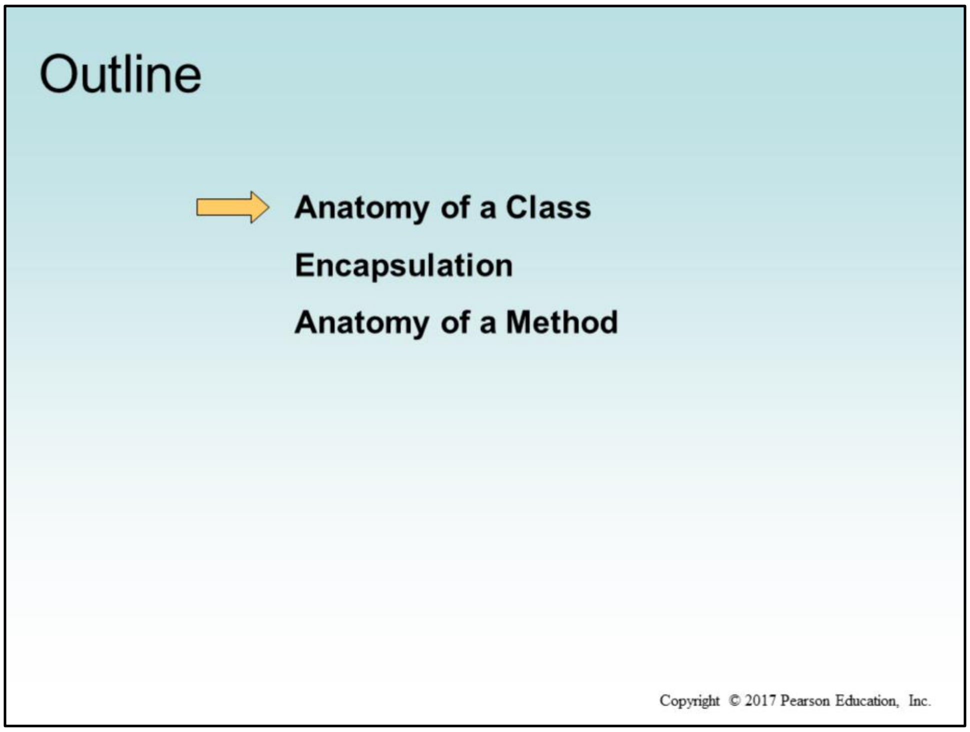
9th Edition

John Lewis
William Loftus



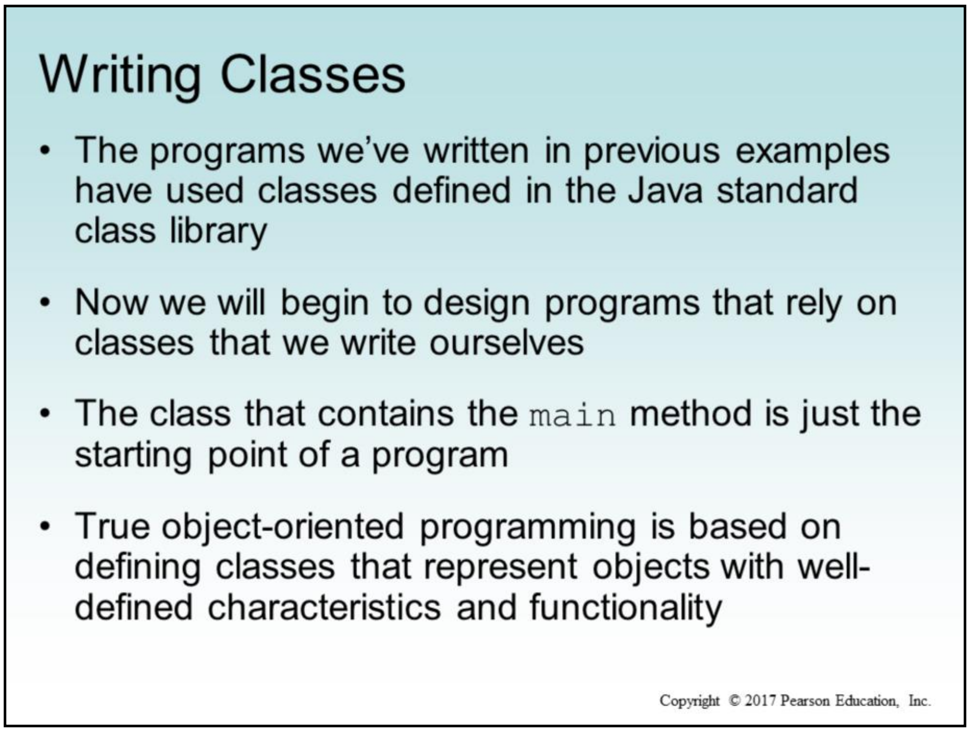
Writing Classes

- We've been using predefined classes from the Java API. Now we will learn to write our own classes.
- Chapter 4 focuses on:
  - class definitions
  - instance data
  - encapsulation and Java modifiers
  - method declaration and parameter passing
  - constructors



Outline

- Anatomy of a Class
- Encapsulation
- Anatomy of a Method



Writing 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 contains the `main` method is 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

-The only class we've written so far (with help from Eclipse) has been one with a **main** method

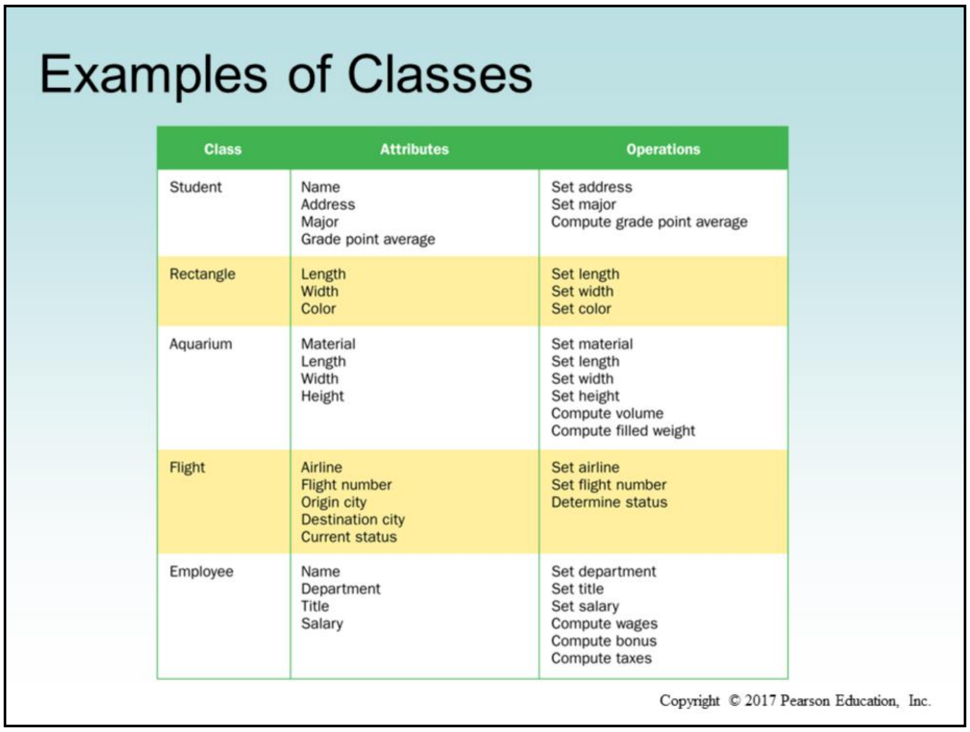
-We sometimes refer to such classes written for this purpose as **drivers**

-These classes exist simply to **drive** the execution of our program

-The main method in the class is the place where program execution begins

-Such a class doesn't really embody a true object-oriented design describing some data model

-In this chapter, we'll design and write classes in the true sense of object-oriented design



Examples of Classes

Class	Attributes	Operations
Student	Name Address Major Grade point average	Set address Set major Compute grade point average
Rectangle	Length Width Color	Set length Set width Set color
Aquarium	Material Length Width Height	Set material Set length Set width Set height Compute volume Compute filled weight
Flight	Airline Flight number Origin city Destination city Current status	Set airline Set flight number Determine status
Employee	Name Department Title Salary	Set department Set title Set salary Compute wages Compute bonus Compute taxes

-The **state** of an object is the value of all it **knows** about at a particular instance in time

-For example, at any instant in time, YOU have a **state**

-The values of your blood pressure, heart rate, weight, location, etc. define your state

-This state is typically different from other people and may change at any time

-Object's live in the same way during the execution of a program

-An object stores it state values in **attributes** (variables, fields) in memory

-Similarly, an object can **do** things, or perform actions or has **behaviors**

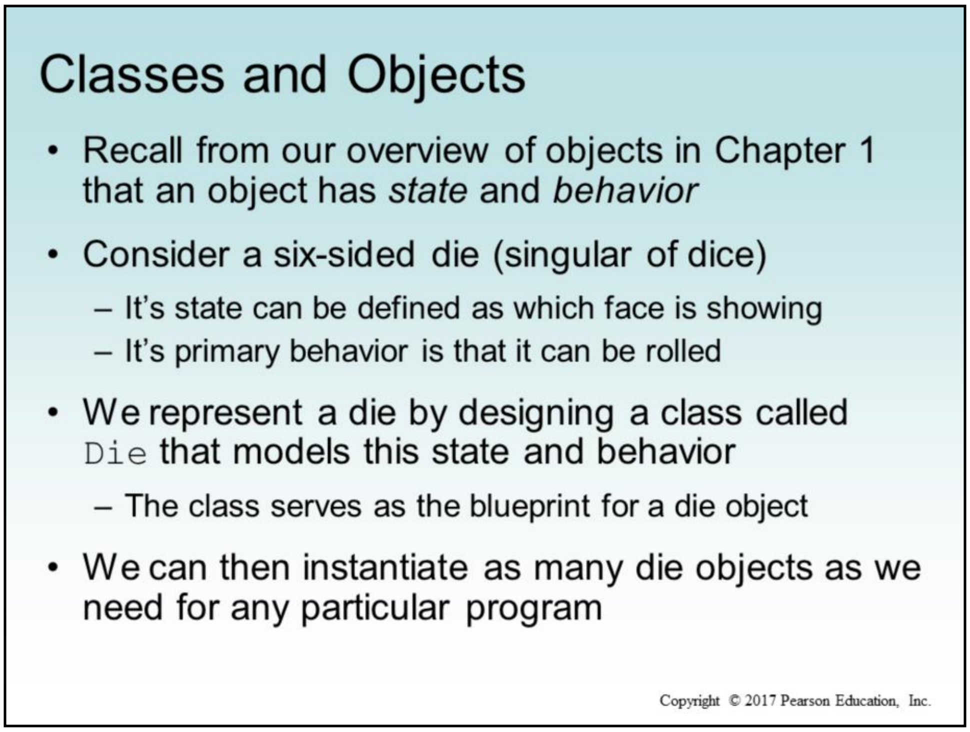
-The instructions defining such behaviors are contained in the class **methods**

-attributes store what an object **knows**, methods define what it **does**

-Oftentimes, the methods can change the state of the object

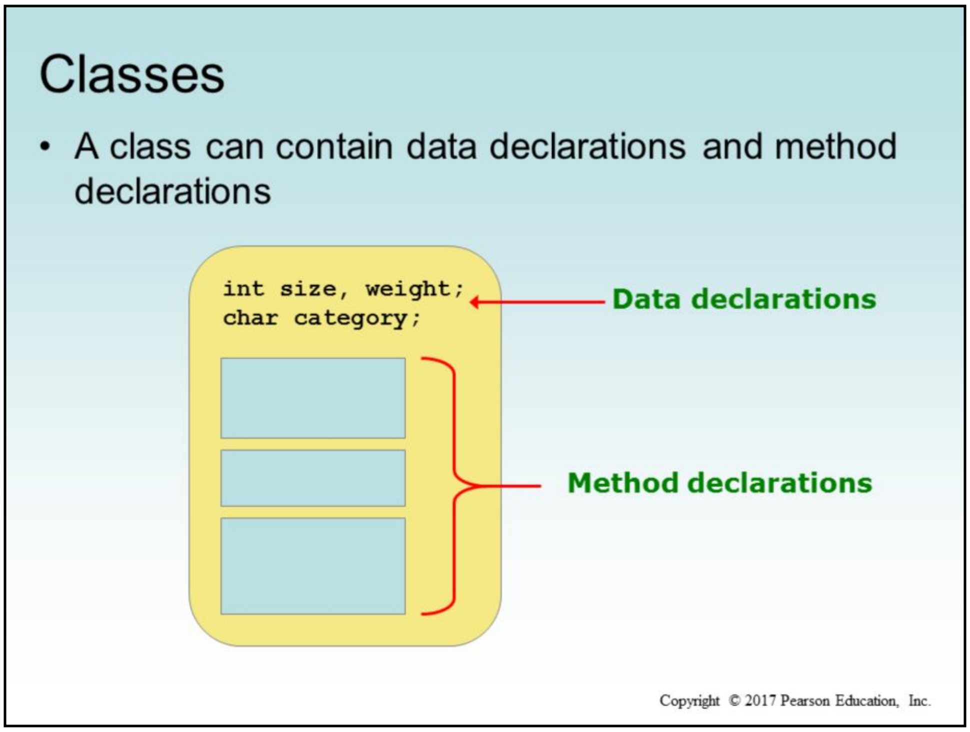
-Again, consider the analogy to people

-You are an object – if you go jogging (behavior), your heart rate changes (state)



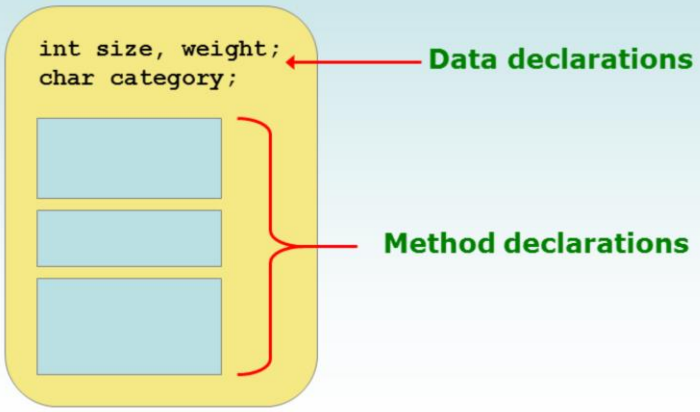
Classes and Objects

- Recall from our overview of objects in Chapter 1 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 represent a die 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



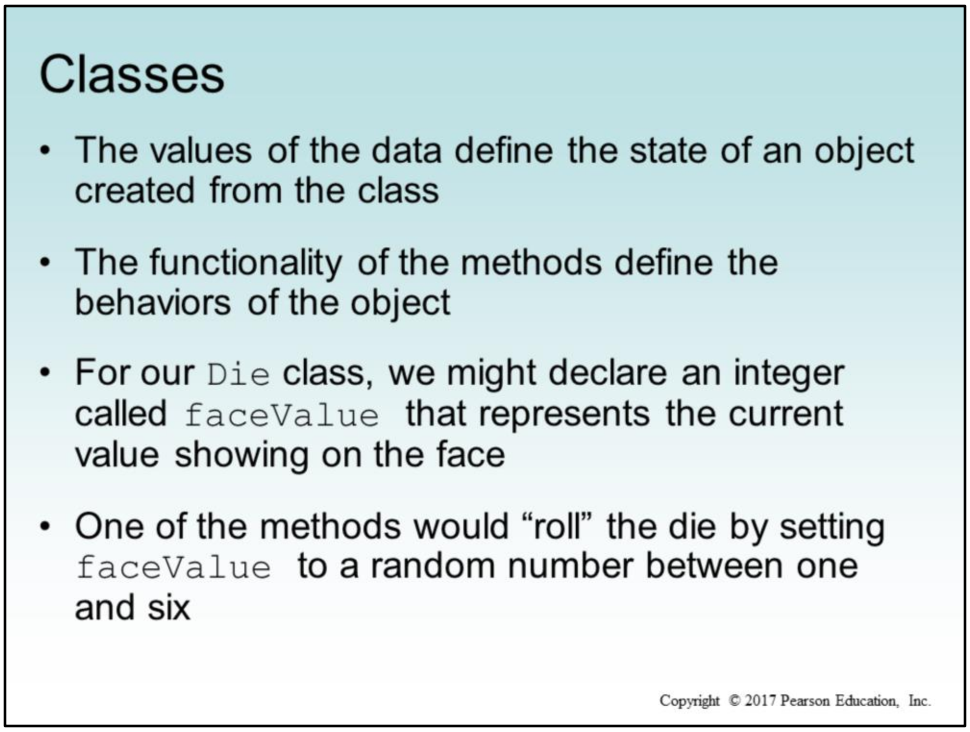
Classes

- A class can contain data declarations and method declarations


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

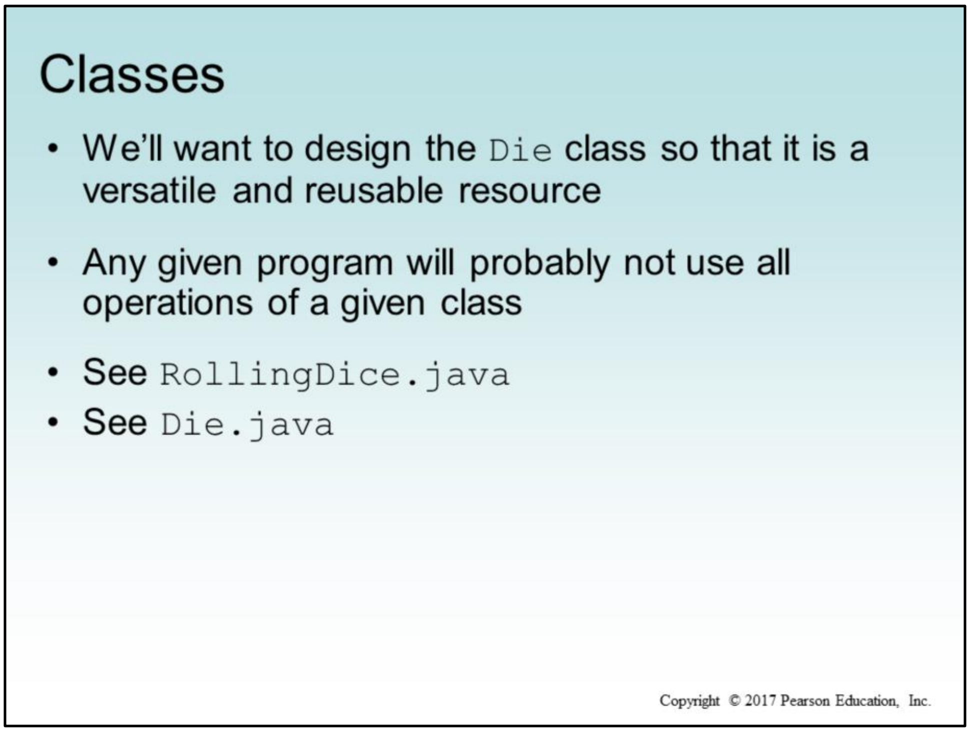
-The data and methods we write in our class are called **members**

-This refers to the fact that they are declared **within** the class itself (members of the class)



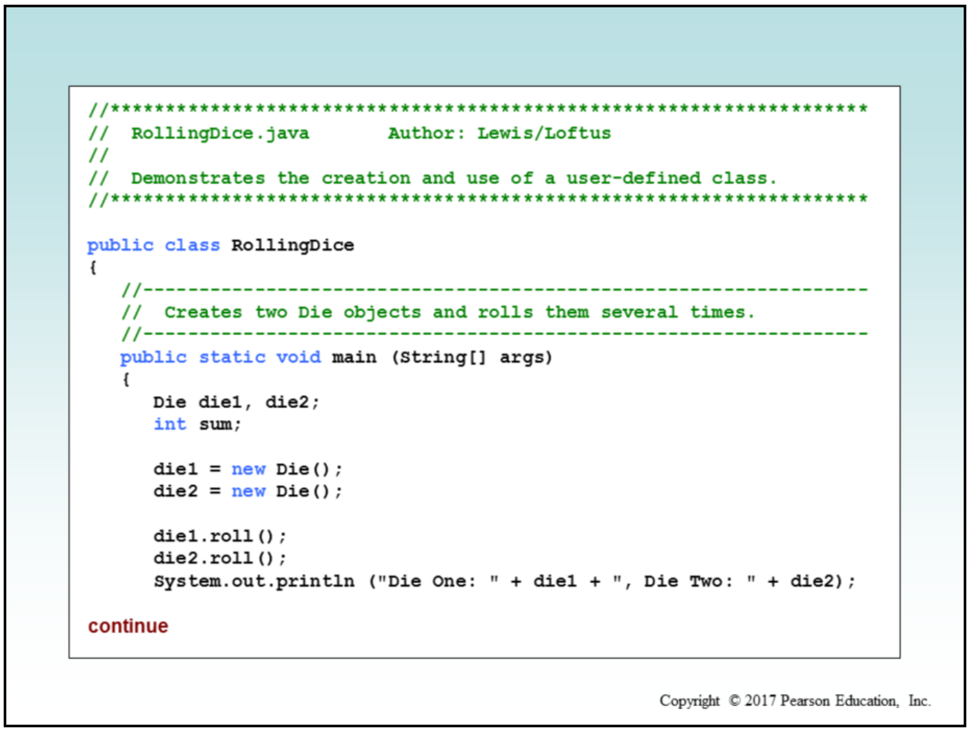
Classes

- 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 called `faceValue` that represents the current value showing on the face
- One of the methods would “roll” the die by setting `faceValue` to a random number between one and six



Classes

- We'll want to design the `Die` class so that it is a versatile and reusable resource
- Any given program will probably not use all operations of a given class
- See `RollingDice.java`
- See `Die.java`



```
//********************************************************************
//  RollingDice.java       Author: Lewis/Loftus
//
//  Demonstrates the creation and use of a user-defined class.
//********************************************************************

public class RollingDice
{
   //-----------------------------------------------------------------
   //  Creates two Die objects and rolls them several times.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      Die die1, die2;
      int sum;

      die1 = new Die();
      die2 = new Die();

      die1.roll();
      die2.roll();
      System.out.println ("Die One: " + die1 + ", Die Two: " + die2);
```

continue

-Note the class RollingDice is our driver program (similar to our programs up to this point)

-Notice how we are creating objects from a class named Die

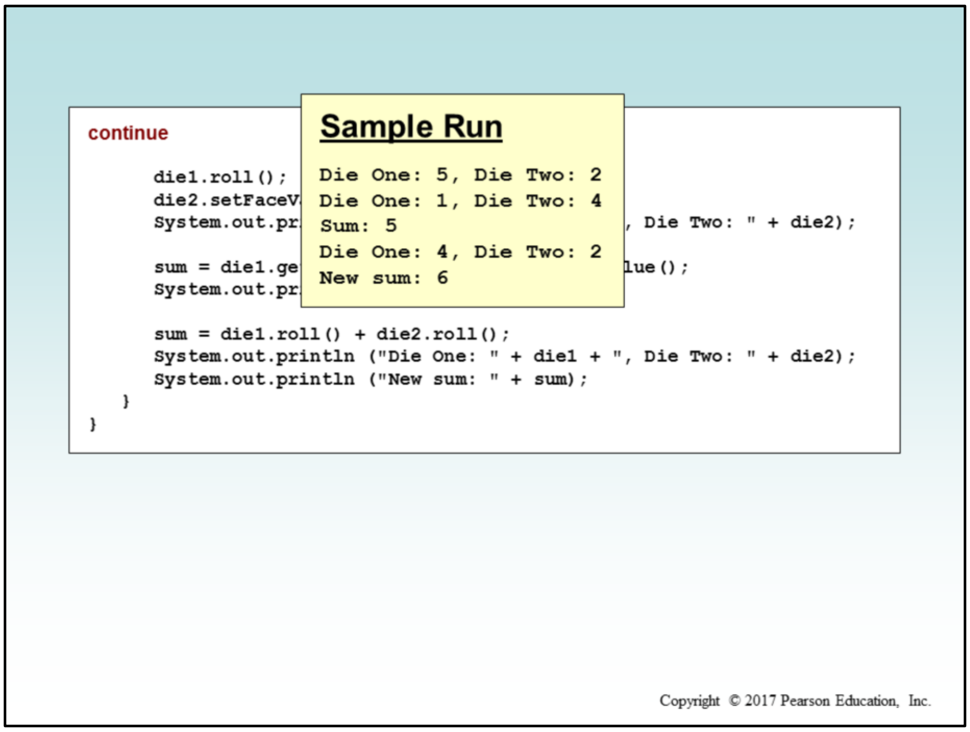
-This is the class that we write, not part of the Java standard library

-Note how we use such objects in a similar manner as objects from Java standard library classes continue

```
      die1.roll();
      die2.setFaceValue(4);
      System.out.println ("Die One: " + die1 + ", Die Two: " + die2);

      sum = die1.getFaceValue() + die2.getFaceValue();
      System.out.println ("Sum: " + sum);

      sum = die1.roll() + die2.roll();
      System.out.println ("Die One: " + die1 + ", Die Two: " + die2);
      System.out.println ("New sum: " + sum);
   }
}
```



continue

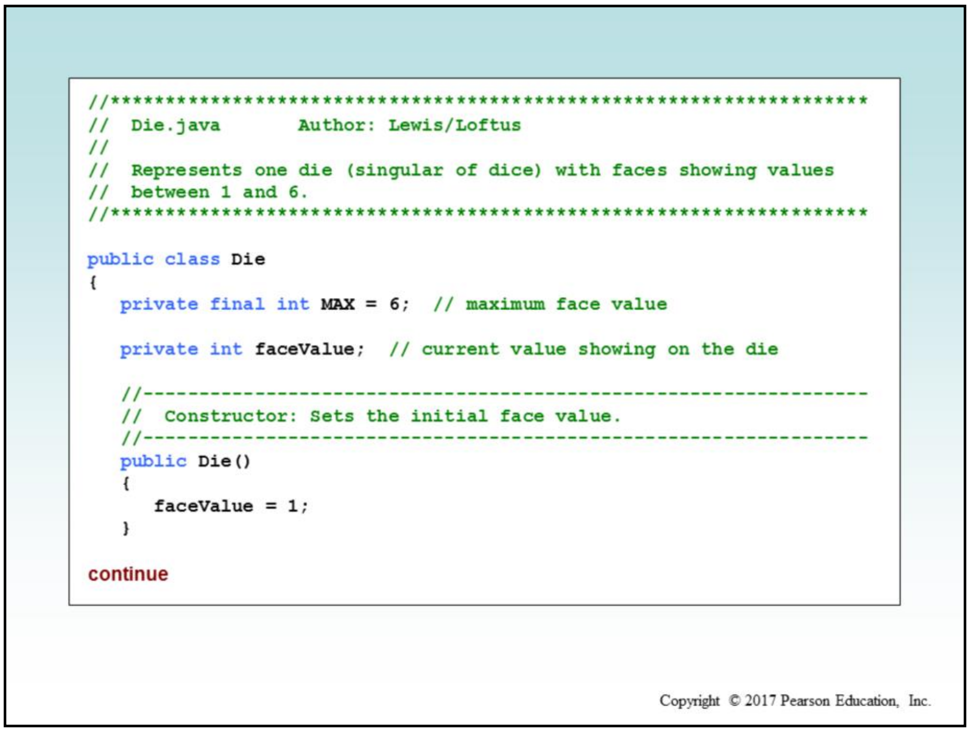
```
      die1.roll();
      die2.setFaceV
      System.out.pr                          , Die Two: " + die2);

      sum = die1.ge                          lue();
      System.out.pr

      sum = die1.roll() + die2.roll();
      System.out.println ("Die One: " + die1 + ", Die Two: " + die2);
      System.out.println ("New sum: " + sum);
   }
}
```

Sample Run

```
Die One: 5, Die Two: 2
Die One: 1, Die Two: 4
Sum: 5
Die One: 4, Die Two: 2
New sum: 6
```



```
//********************************************************************
//  Die.java       Author: Lewis/Loftus
//
//  Represents one die (singular of dice) with faces showing values
//  between 1 and 6.
//********************************************************************

public class Die
{
   private final int MAX = 6;  // maximum face value

   private int faceValue;  // current value showing on the die

   //-----------------------------------------------------------------
   //  Constructor: Sets the initial face value.
   //-----------------------------------------------------------------
   public Die()
   {
      faceValue = 1;
   }
```

continue

-This is the class that we write named Die (written in a separate Java source file with the same name)

-You create one in Eclipse as before, just without a main method

-We first declare variables (attributes) making up what it **knows** (MAX, faceValue)

-We then declare and define methods describing what it can **do** (e.g. roll, getFaceValue)

-Note also a special method with the same name as the class named Die()

-This method is called a **constructor**

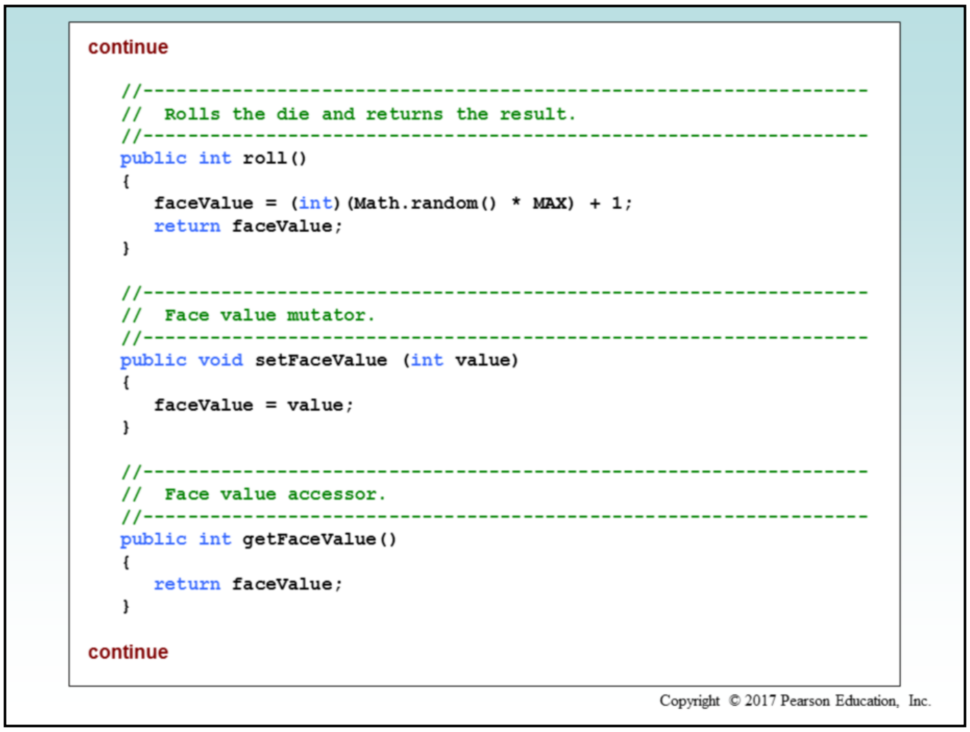
-Recall that a constructor is the method that is automatically called after the object is created in memory

-This is called after the new operator is used to create the object in memory

-Once the object is created, the constructor is then called

-The constructor typically initializes (assigns) attributes in the class with default values

-In our example, we initialize the faceValue to 1 in the constructor


continue

```
   //-----------------------------------------------------------------
   //  Rolls the die and returns the result.
   //-----------------------------------------------------------------
   public int roll()
   {
      faceValue = (int)(Math.random() * MAX) + 1;
      return faceValue;
   }

   //-----------------------------------------------------------------
   //  Face value mutator.
   //-----------------------------------------------------------------
   public void setFaceValue (int value)
   {
      faceValue = value;
   }

   //-----------------------------------------------------------------
   //  Face value accessor.
   //-----------------------------------------------------------------
   public int getFaceValue()
   {
      return faceValue;
   }
```

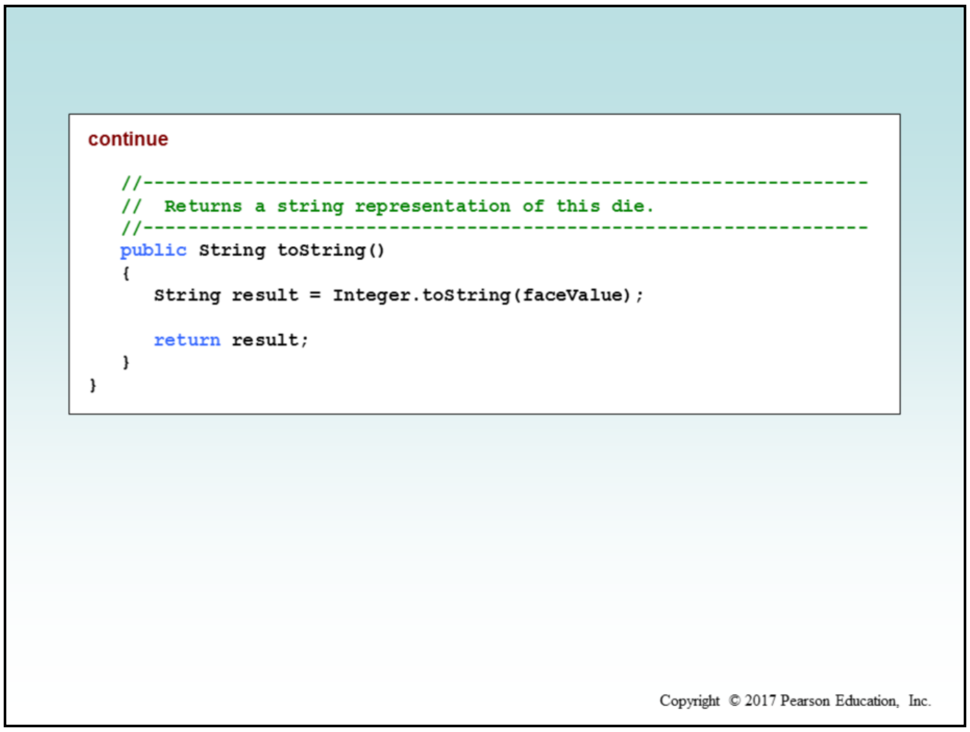
continue

-Look at the first lines of each of the method definitions (e.g. public int roll())

-These lines inform the compiler about important information about each of the methods

-We'll be studying such method definitions in detail later

-Note also the documentation that is specified to help explain the methods



continue

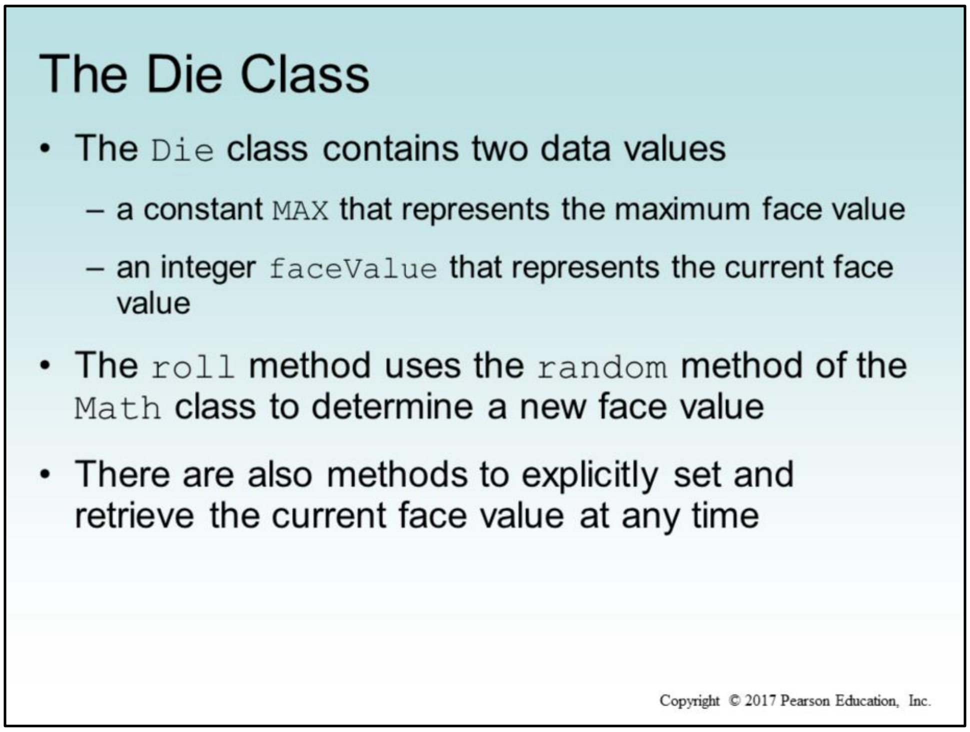
```
   //-----------------------------------------------------------------
   //  Returns a string representation of this die.
   //-----------------------------------------------------------------
   public String toString()
   {
      String result = Integer.toString(faceValue);

      return result;
   }
}
```

-As we'll see later, the method toString () is a special kind of **inherited** method

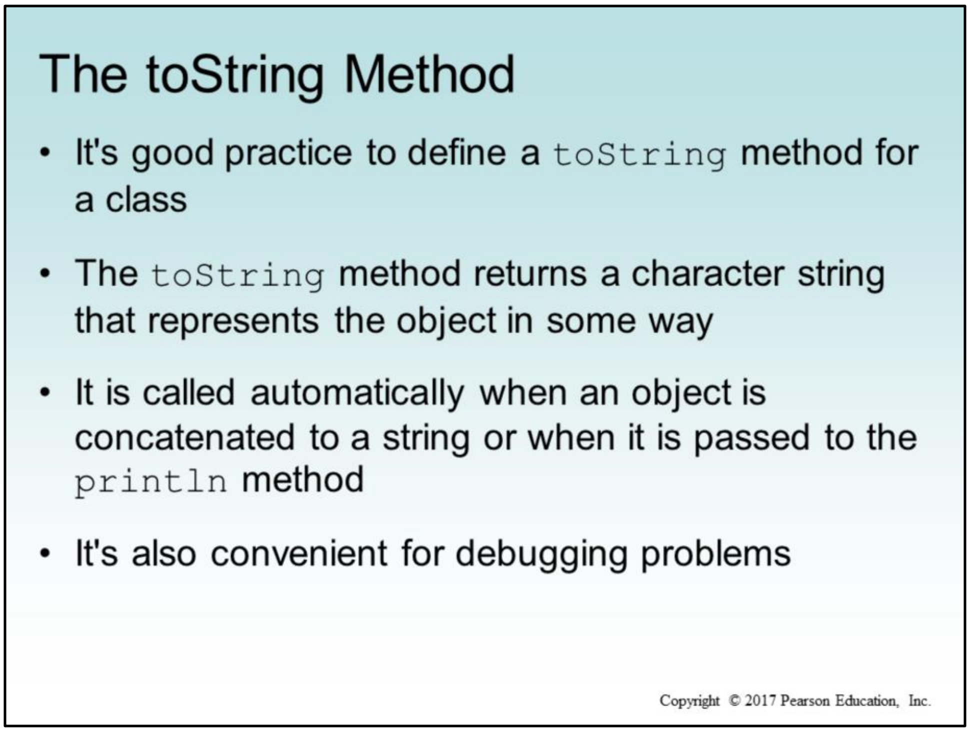
-Its purpose is to create a textual representation for information in the class

-In our example above, this method converts the faceValue into a String



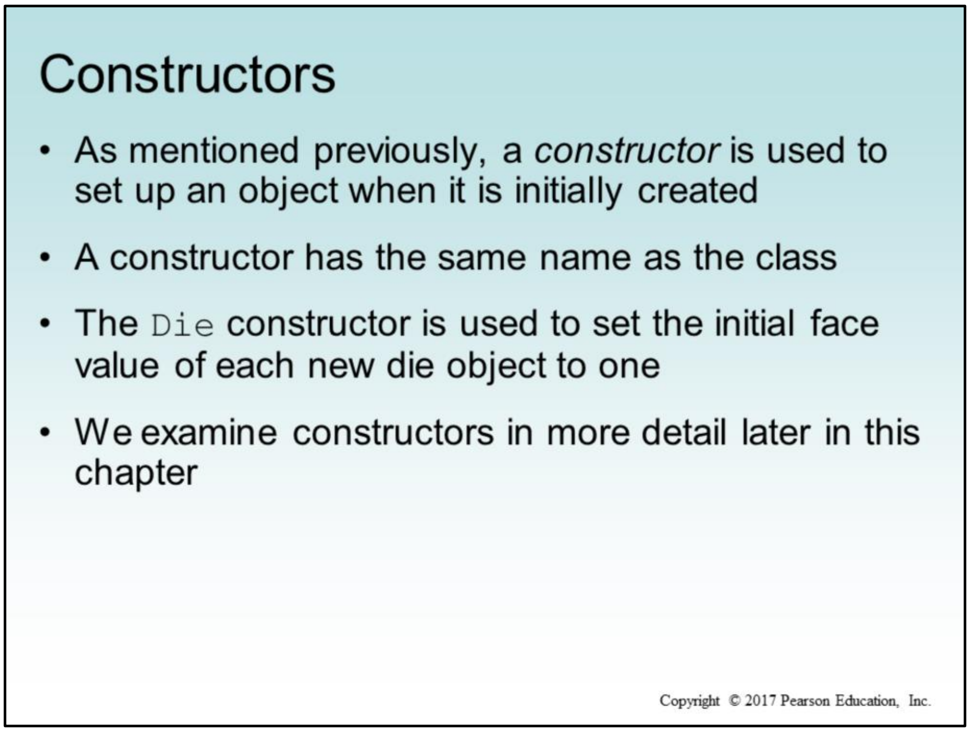
The Die Class

- The `Die` class contains two data values
  - a constant `MAX` that represents the maximum face value
  - an integer `faceValue` that represents the current face value
- The `roll` method uses the `random` method of the `Math` class to determine a new face value
- There are also methods to explicitly set and retrieve the current face value at any time



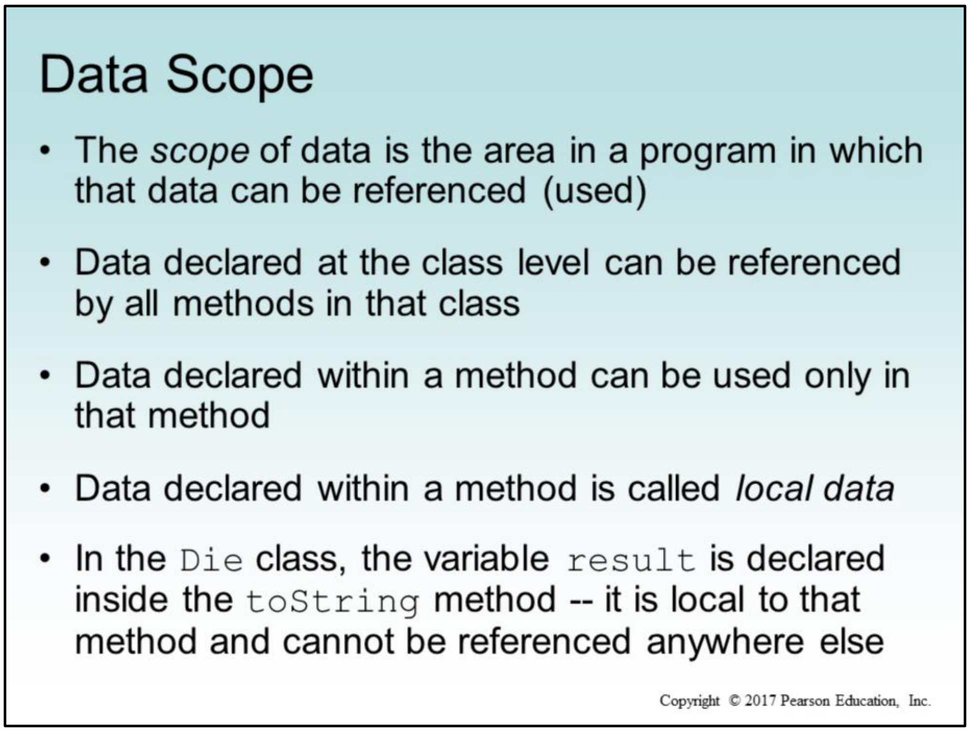
The toString Method

- It's good practice to define a `toString` method for a class
- 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
- It's also convenient for debugging problems



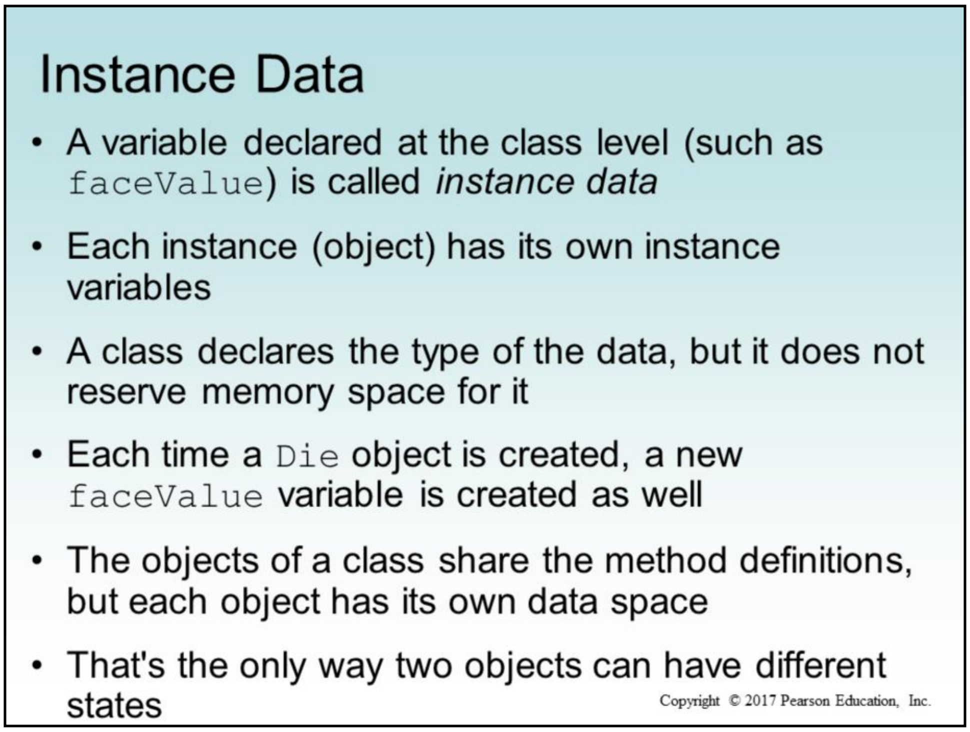
Constructors

- As mentioned previously, a *constructor* 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 in this chapter



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



Instance Data

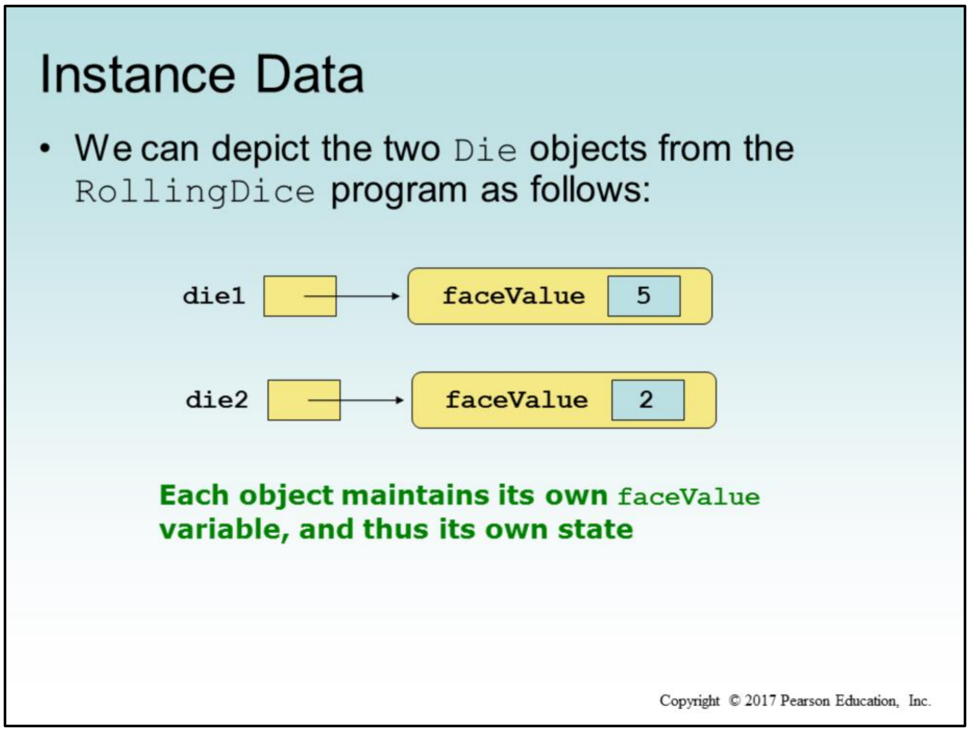
- A variable declared at the class level (such as `faceValue`) is called *instance data*
- Each instance (object) has its own instance variables
- A class declares the type of the data, but it does not reserve memory space for it
- Each time a `Die` object is created, a new `faceValue` variable is created 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

-Each object created by the new operator is given memory in which to store values for each of the attributes in the class

-In our example, two Die objects were created in memory

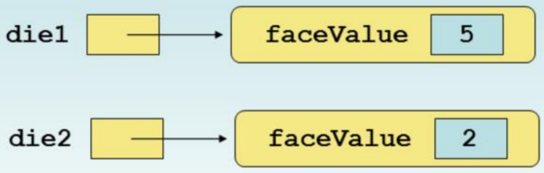
-Each could store different faceValues in their respective areas in memory

-We refer to such data as **instance** data since it is data stored in an object **instantiation**

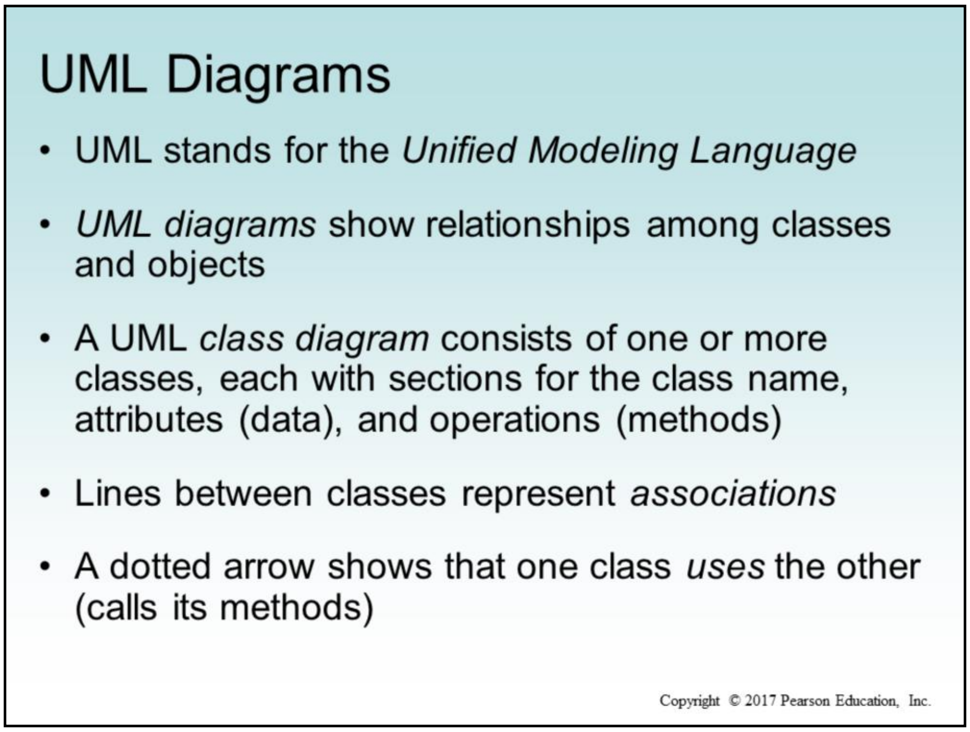


Instance Data

- 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



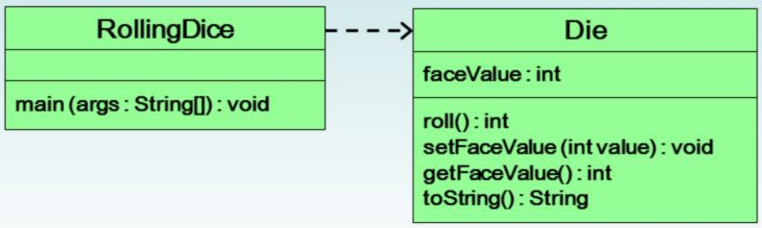
UML Diagrams

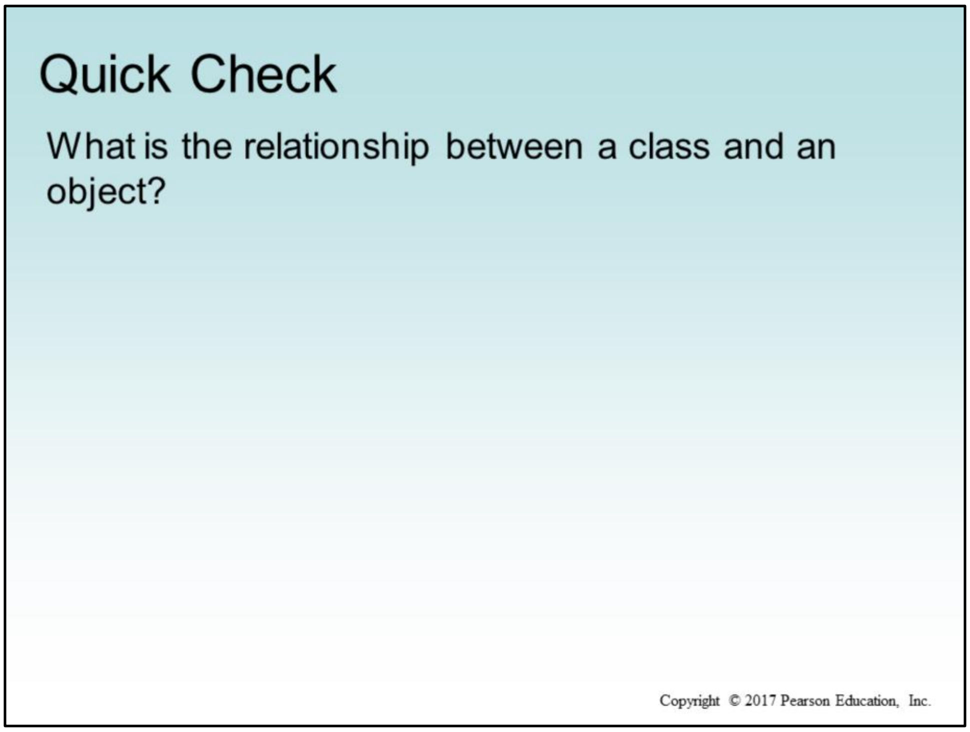
- UML stands for the *Unified Modeling Language*
- *UML diagrams* show relationships among classes and objects
- A UML *class diagram* consists of one or more classes, each with sections for the class name, attributes (data), and operations (methods)
- Lines between classes represent *associations*
- A dotted arrow shows that one class *uses* the other (calls its methods)

-UML diagrams are used to describe object-oriented designs… in any language, not just Java

UML Class Diagrams

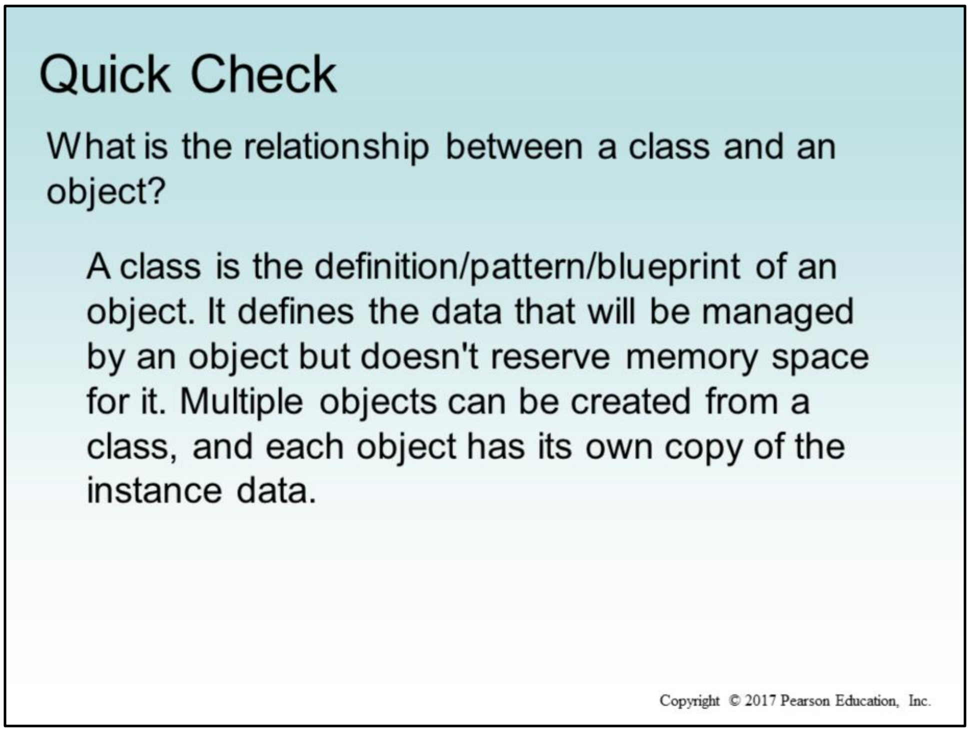
- A UML class diagram for the `RollingDice` program:





Quick Check

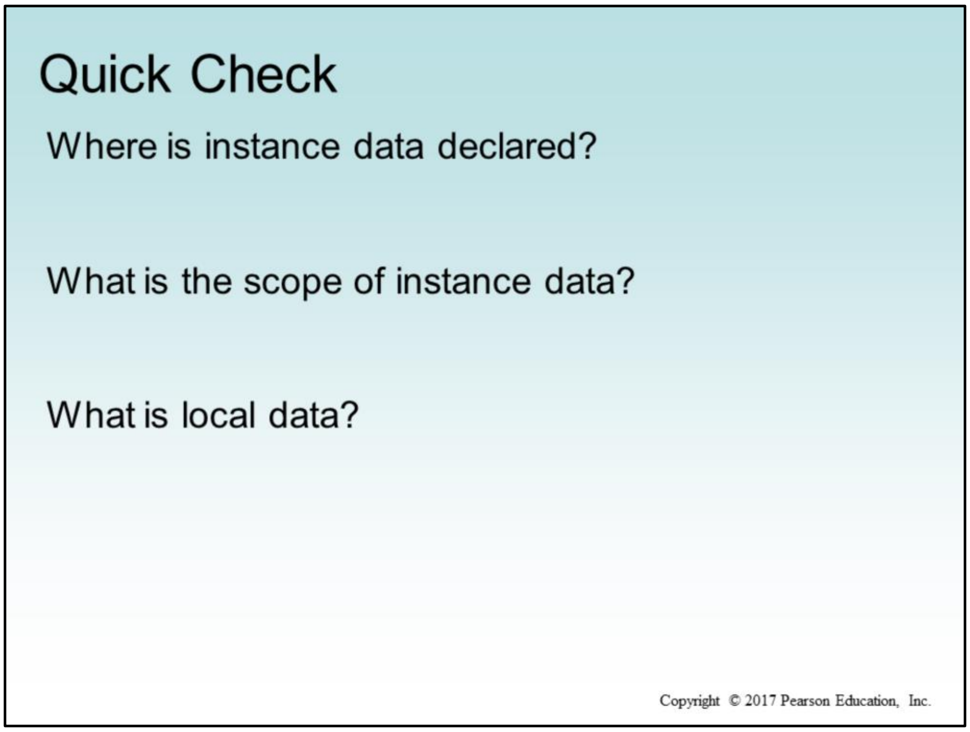
What is the relationship between a class and an object?



Quick Check

What is the relationship between a class and an object?

A class is the definition/pattern/blueprint of an object. It defines the data that will be managed by an object but doesn't reserve memory space for it. Multiple objects can be created from a class, and each object has its own copy of the instance data.

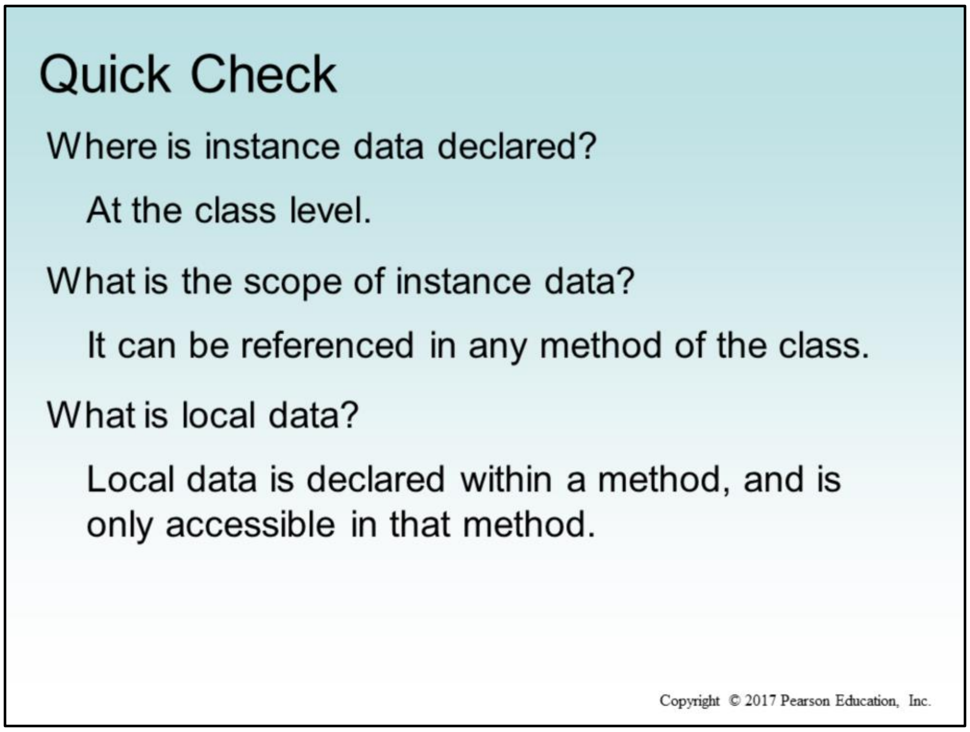


Quick Check

Where is instance data declared?

What is the scope of instance data?

What is local data?



Quick Check

Where is instance data declared?

At the class level.

What is the scope of instance data?

It can be referenced in any method of the class.

What is local data?

Local data is declared within a method, and is only accessible in that method.