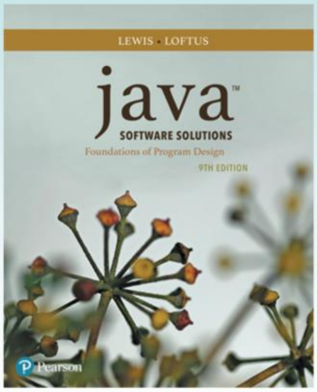
Chapter 7
Object-Oriented Design

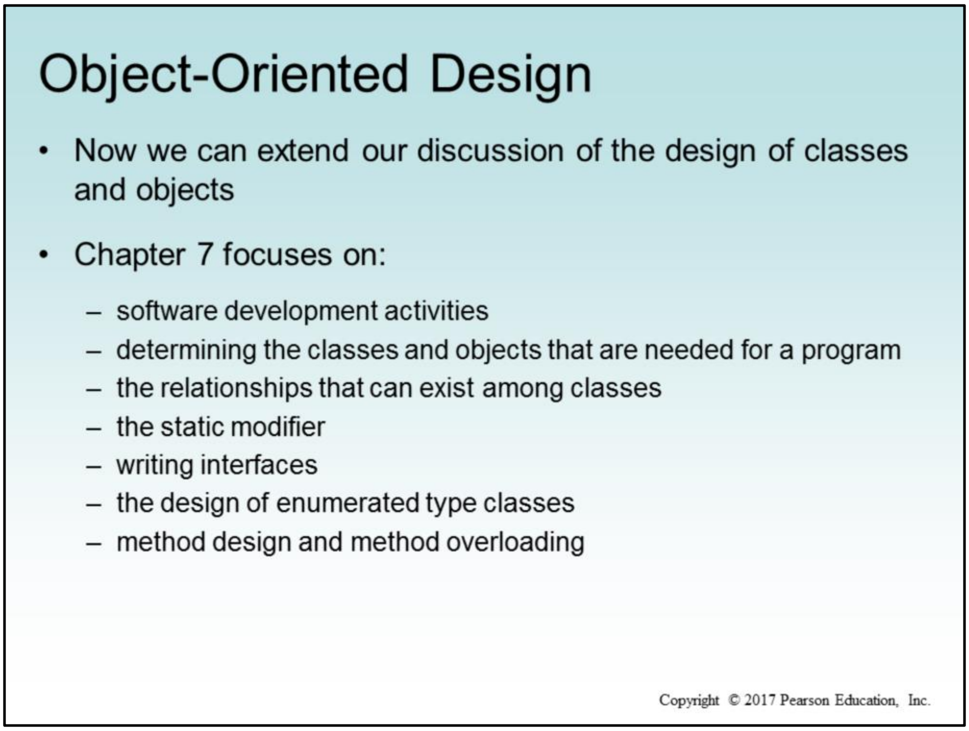


Java Software Solutions

Foundations of Program Design

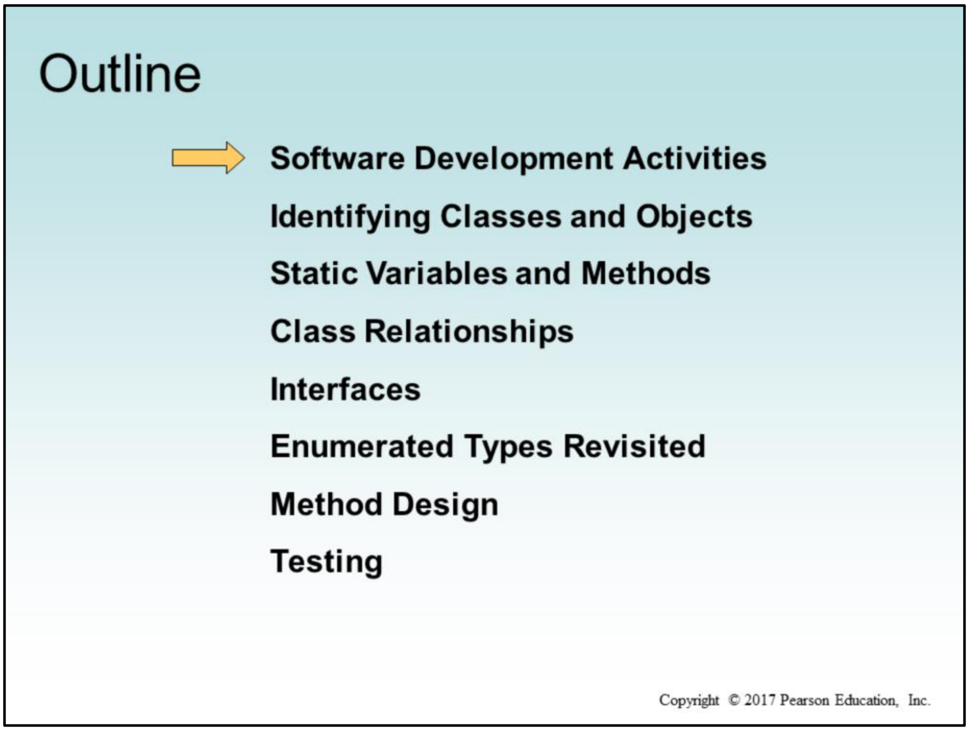
9th Edition

John Lewis
William Loftus



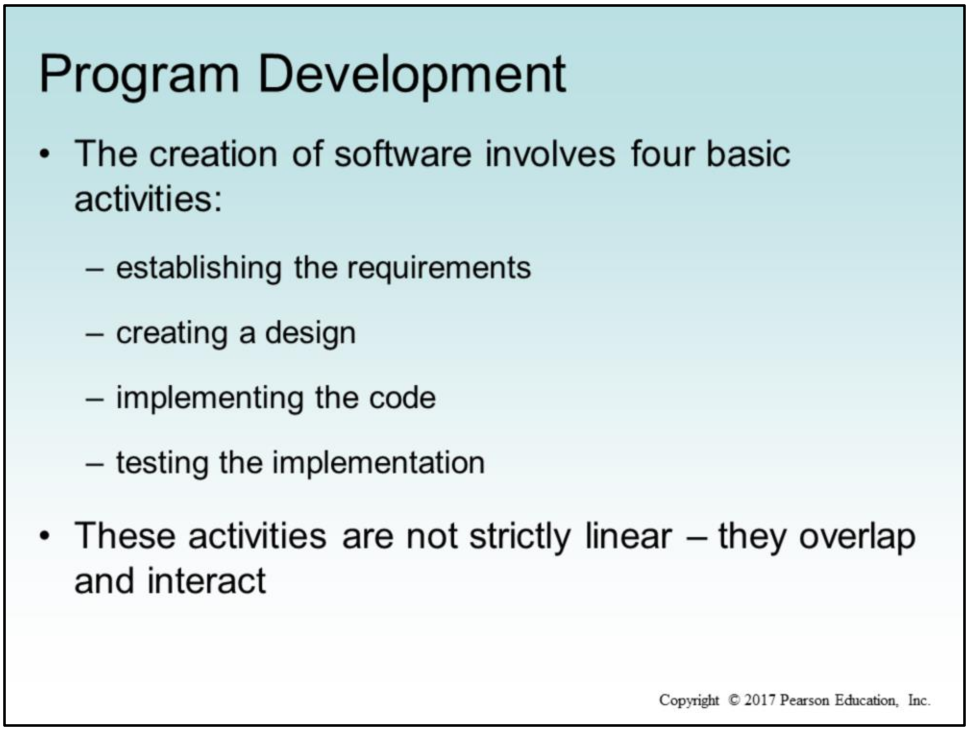
Object-Oriented Design

- Now we can extend our discussion of the design of classes and objects
- Chapter 7 focuses on:
  - software development activities
  - determining the classes and objects that are needed for a program
  - the relationships that can exist among classes
  - the static modifier
  - writing interfaces
  - the design of enumerated type classes
  - method design and method overloading



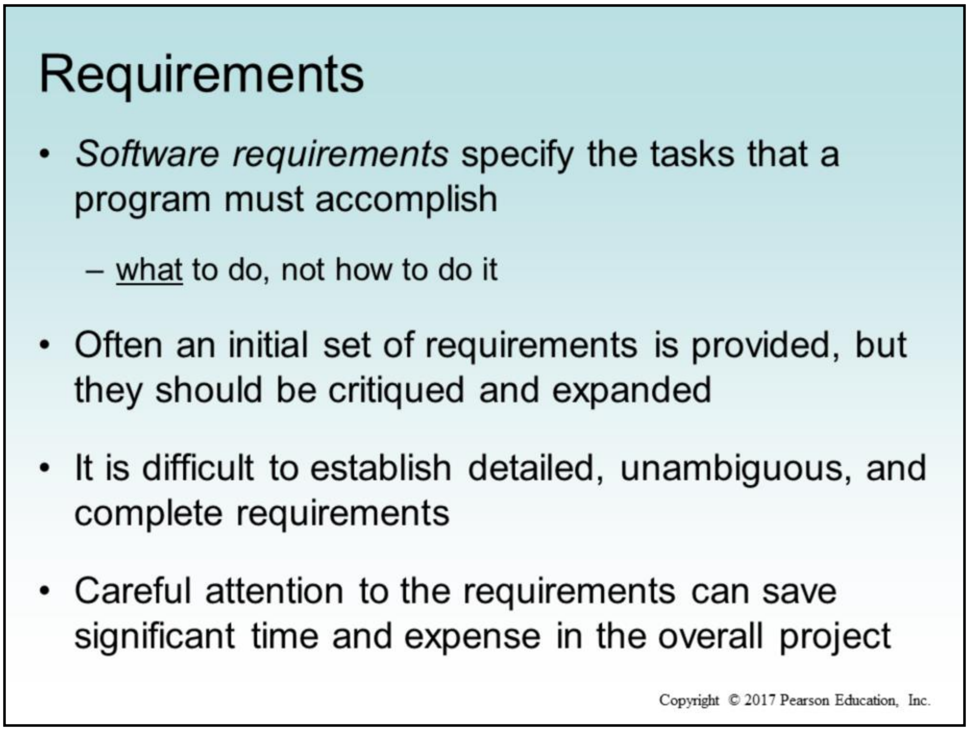
Outline

- ⇨ **Software Development Activities**
- **Identifying Classes and Objects**
- **Static Variables and Methods**
- **Class Relationships**
- **Interfaces**
- **Enumerated Types Revisited**
- **Method Design**
- **Testing**



Program Development

- The creation of software involves four basic activities:
  - establishing the requirements
  - creating a design
  - implementing the code
  - testing the implementation
- These activities are not strictly linear – they overlap and interact



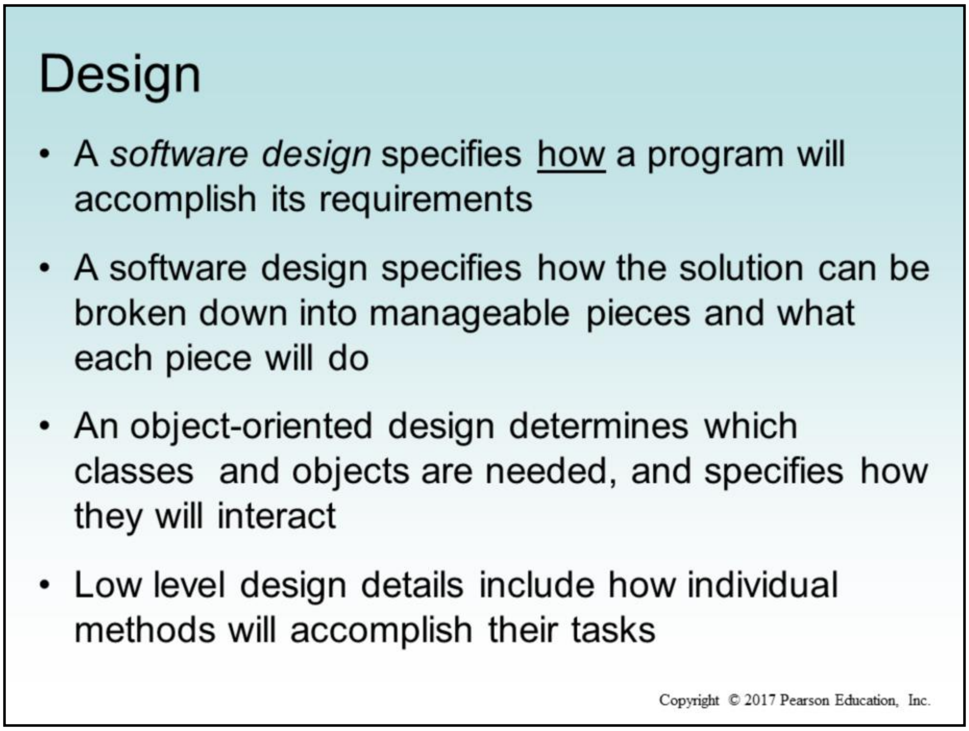
Requirements

- *Software requirements* specify the tasks that a program must accomplish
  - what to do, not how to do it
- Often an initial set of requirements is provided, but they should be critiqued and expanded
- It is difficult to establish detailed, unambiguous, and complete requirements
- Careful attention to the requirements can save significant time and expense in the overall project

-Note that our lab assignment descriptions act as the software requirements for our labs

-Note requirements are *what* to do, not *how* to do it

-A common mistake is to think about *how* instead of *what* too early in the development of a project

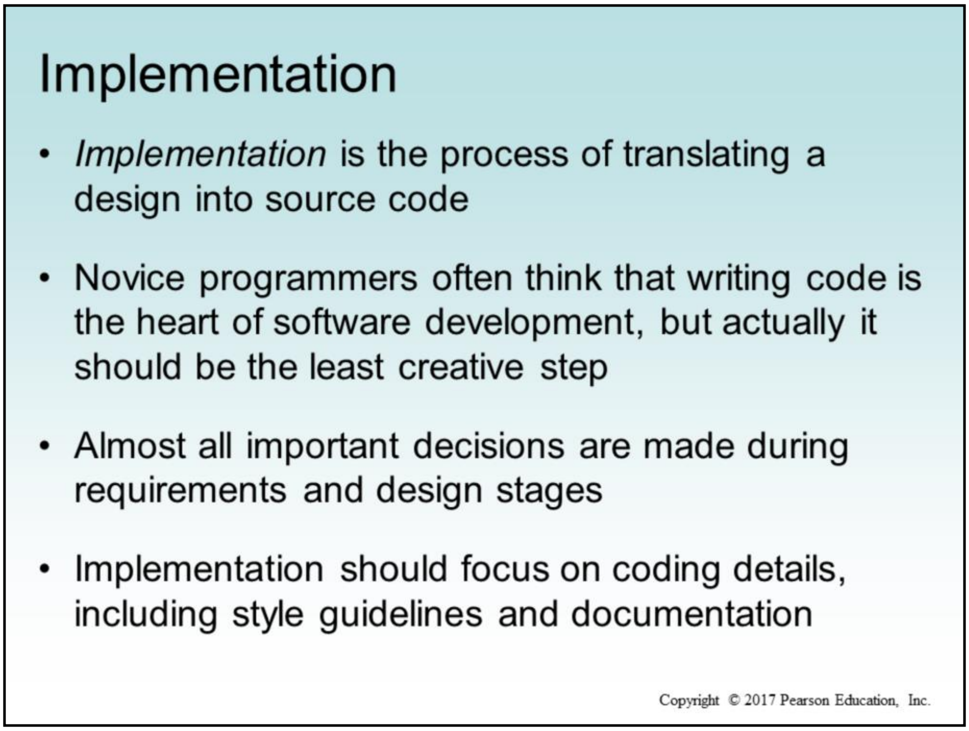


Design

- A *software design* specifies how a program will accomplish its requirements
- A software design specifies how the solution can be broken down into manageable pieces and what each piece will do
- An object-oriented design determines which classes and objects are needed, and specifies how they will interact
- Low level design details include how individual methods will accomplish their tasks

-A good habit is to design a program (e.g. classes, logic) on paper before attempting to code

-Careful design on paper **before** programming can save enormous amounts of time and problems!



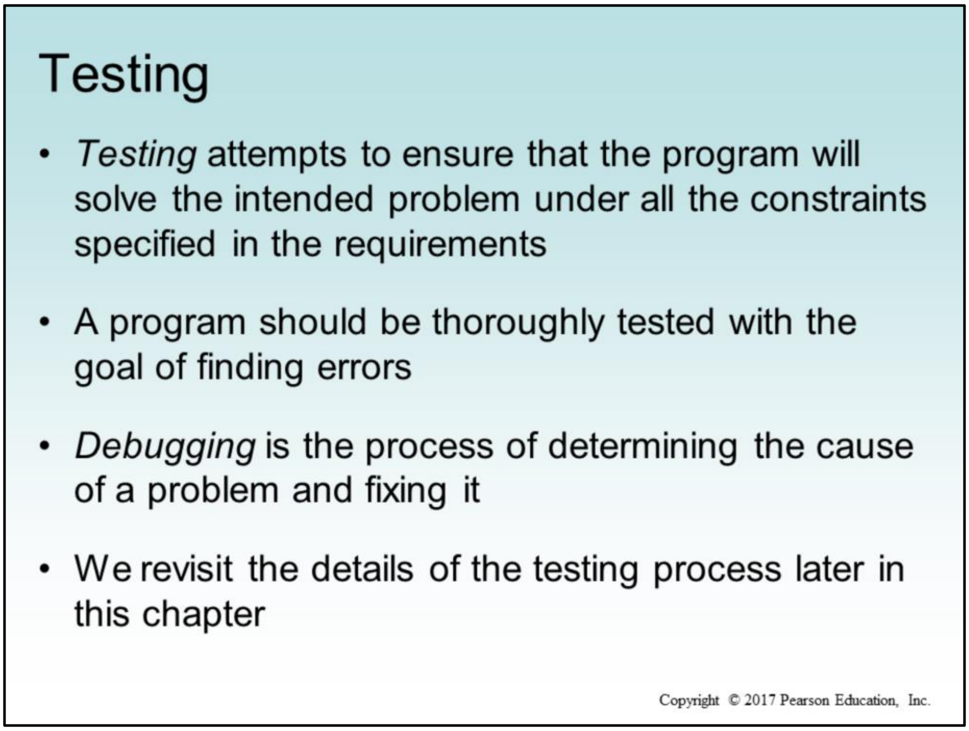
Implementation

- *Implementation* is the process of translating a design into source code
- Novice programmers often think that writing code is the heart of software development, but actually it should be the least creative step
- Almost all important decisions are made during requirements and design stages
- Implementation should focus on coding details, including style guidelines and documentation

-Note that implementation is simply a translation of a good design into source code

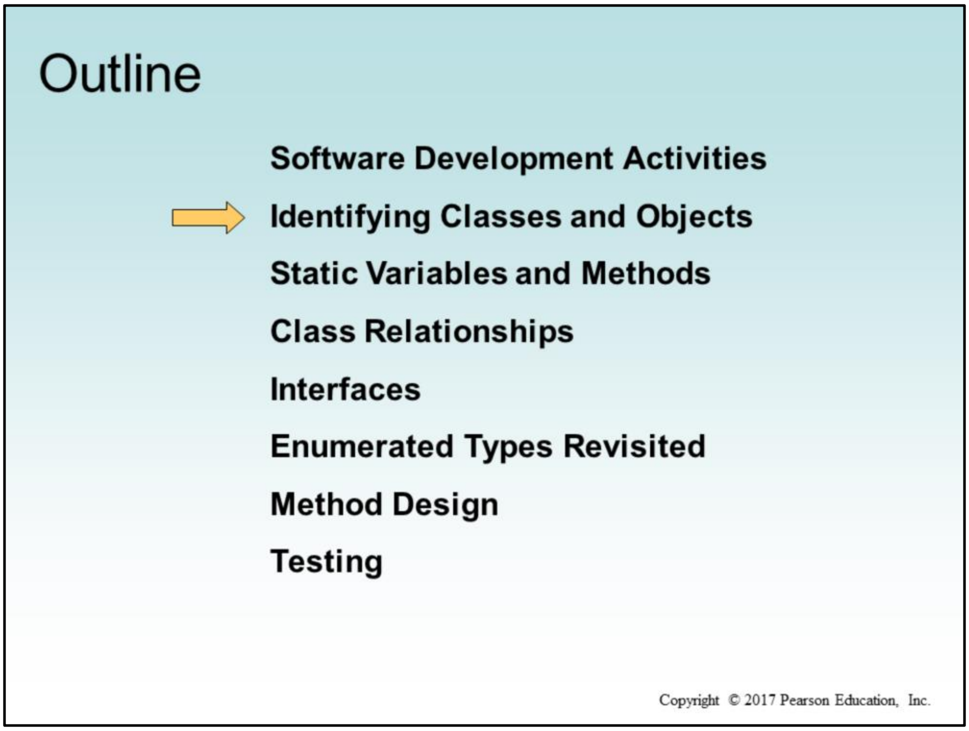
-There should not be any guess work in this process if the design is solid

-Rule: implement **only** after a thorough design



Testing

- *Testing* attempts to ensure that the program will solve the intended problem under all the constraints specified in the requirements
- A program should be thoroughly tested with the goal of finding errors
- *Debugging* is the process of determining the cause of a problem and fixing it
- We revisit the details of the testing process later in this chapter



Outline

Software Development Activities

➡ Identifying Classes and Objects

Static Variables and Methods

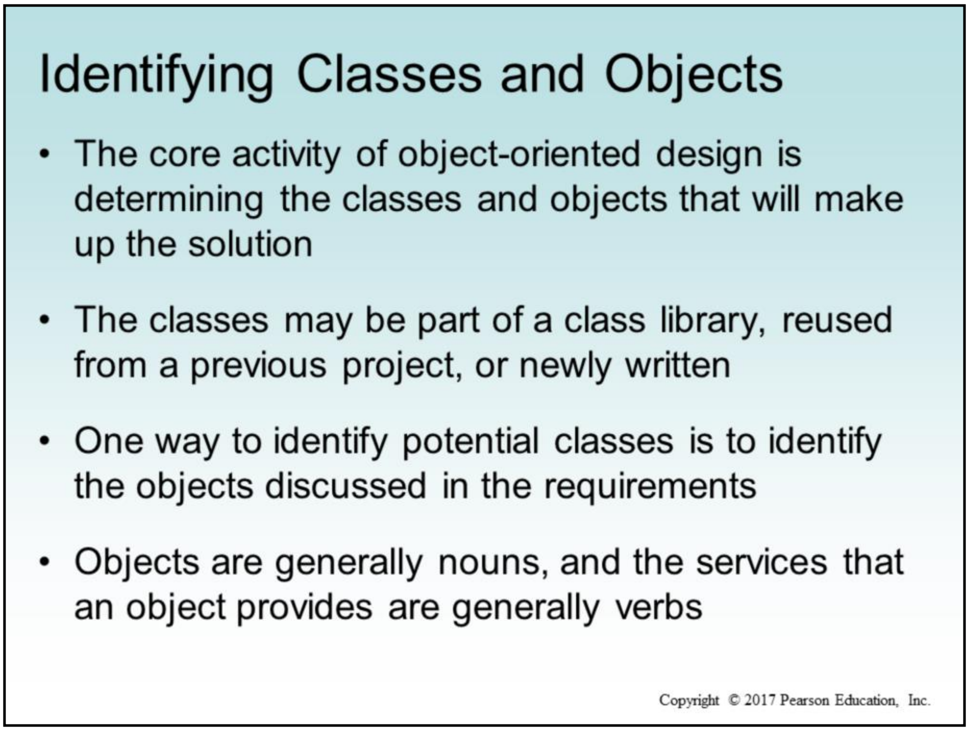
Class Relationships

Interfaces

Enumerated Types Revisited

Method Design

Testing

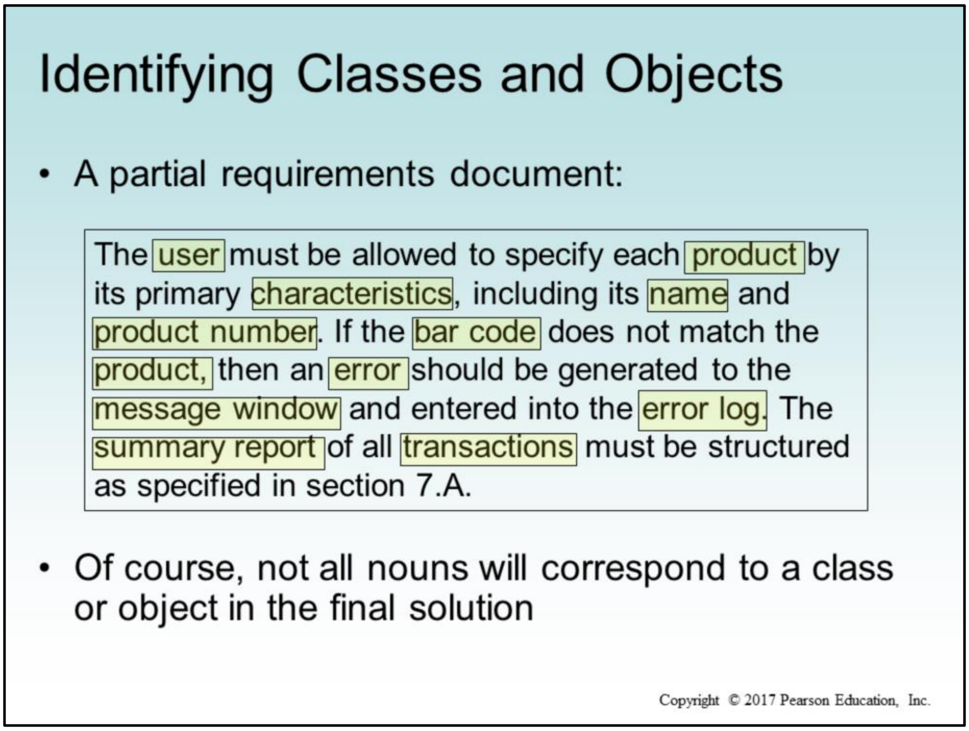


Identifying Classes and Objects

- The core activity of object-oriented design is determining the classes and objects that will make up the solution
- The classes may be part of a class library, reused from a previous project, or newly written
- One way to identify potential classes is to identify the objects discussed in the requirements
- Objects are generally nouns, and the services that an object provides are generally verbs

-After identifying candidates for classes in a description, first look in the Java API

-The class may have already been implemented in the Java API

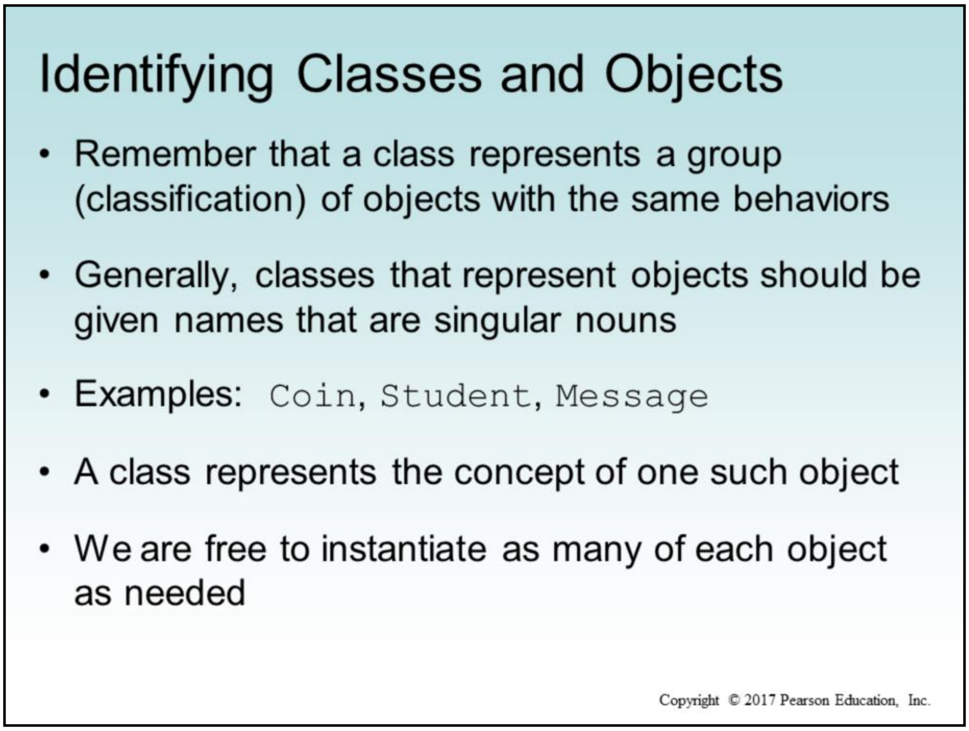


Identifying Classes and Objects

- A partial requirements document:

> The user must be allowed to specify each product by its primary characteristics, including its name and product number. If the bar code does not match the product, then an error should be generated to the message window and entered into the error log. The summary report of all transactions must be structured as specified in section 7.A.

- Of course, not all nouns will correspond to a class or object in the final solution



Identifying Classes and Objects

- Remember that a class represents a group (classification) of objects with the same behaviors
- Generally, classes that represent objects should be given names that are singular nouns
- Examples: `Coin`, `Student`, `Message`
- A class represents the concept of one such object
- We are free to instantiate as many of each object as needed

-A class should describe and encapsulate the description of a single entity

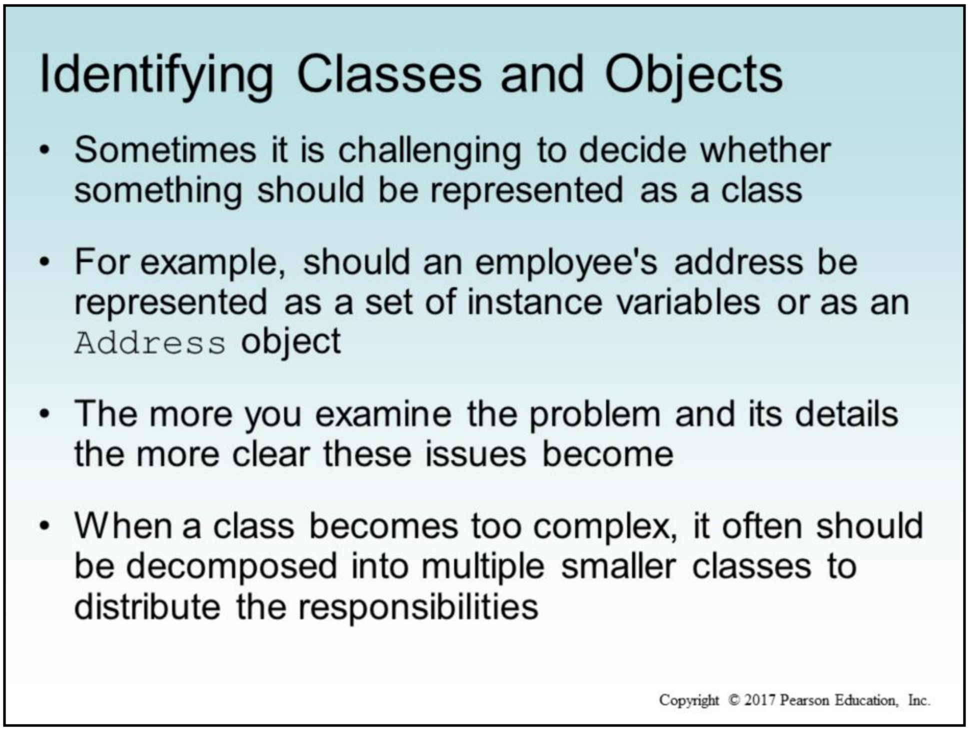
-Avoid designing classes that describe multiple items

-For example, if the description is “write a program to add two fractions”

-Write a class **Fraction** to describe a single fraction, not a class **Fractions** that describes two fractions

-Why? The answer is reusability! You want to write classes that you can reuse

-You’ll reuse a **Fraction** class more than a **Fractions** class



Identifying Classes and Objects

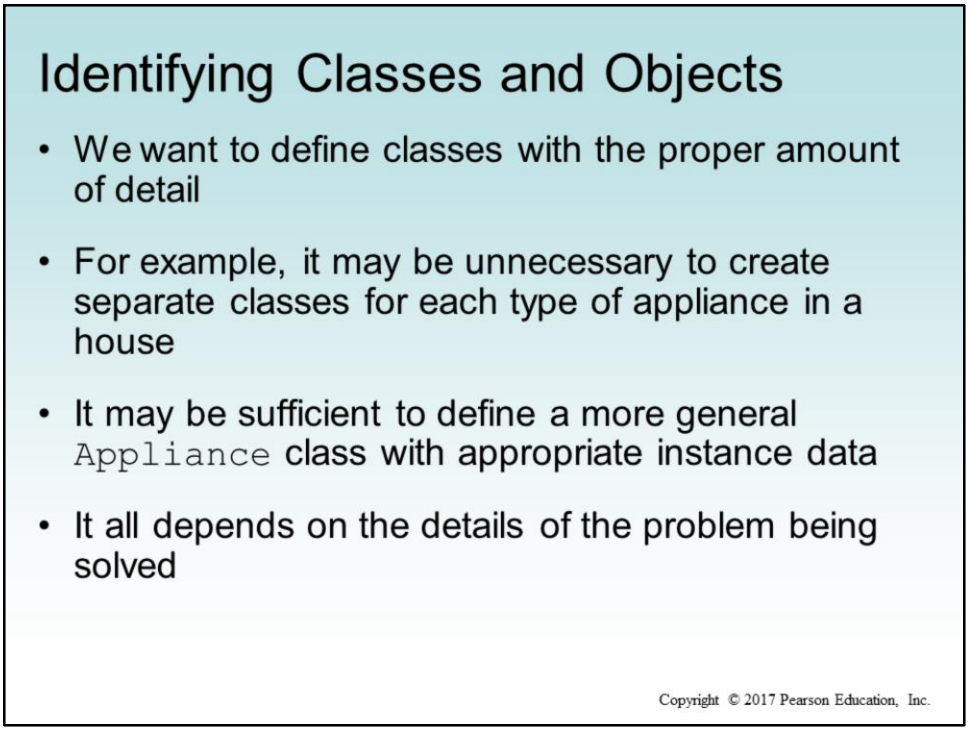
- Sometimes it is challenging to decide whether something should be represented as a class
- For example, should an employee's address be represented as a set of instance variables or as an `Address` object
- The more you examine the problem and its details the more clear these issues become
- When a class becomes too complex, it often should be decomposed into multiple smaller classes to distribute the responsibilities

-Oftentimes listing what a class “knows” about will help identify new class candidates

-Consider the design of Clock class

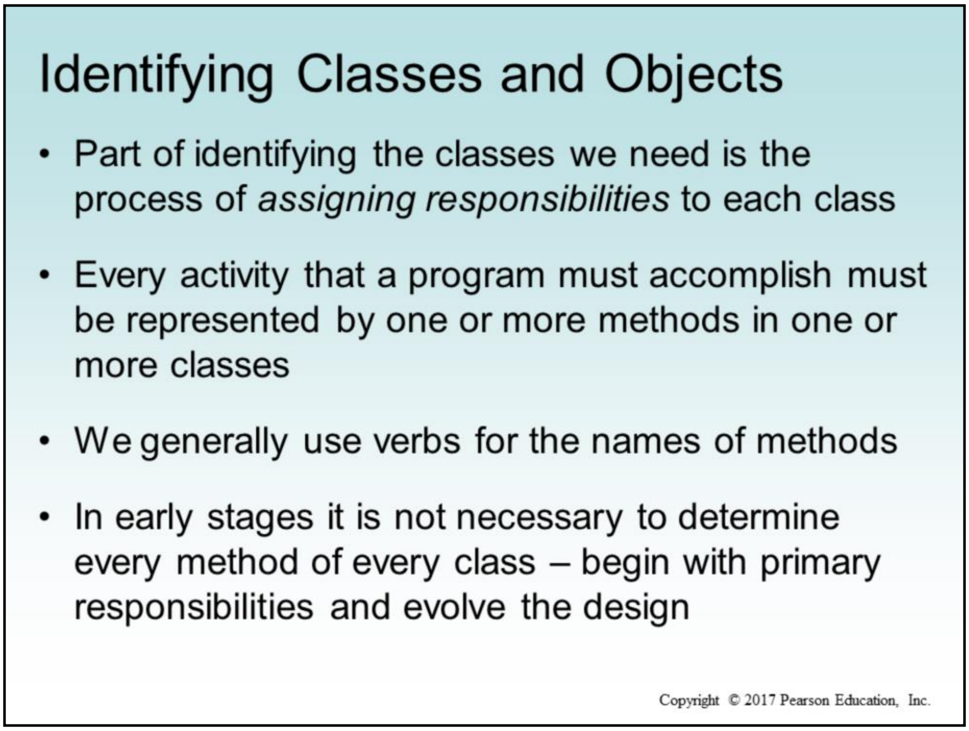
-Among the attributes of what a Clock knows, we might list an alarm

-If we can’t store information for an alarm in primitive data types, this could become another class!



Identifying Classes and Objects

- We want to define classes with the proper amount of detail
- For example, it may be unnecessary to create separate classes for each type of appliance in a house
- It may be sufficient to define a more general `Appliance` class with appropriate instance data
- It all depends on the details of the problem being solved



Identifying Classes and Objects

- Part of identifying the classes we need is the process of *assigning responsibilities* to each class
- Every activity that a program must accomplish must be represented by one or more methods in one or more classes
- We generally use verbs for the names of methods
- In early stages it is not necessary to determine every method of every class – begin with primary responsibilities and evolve the design

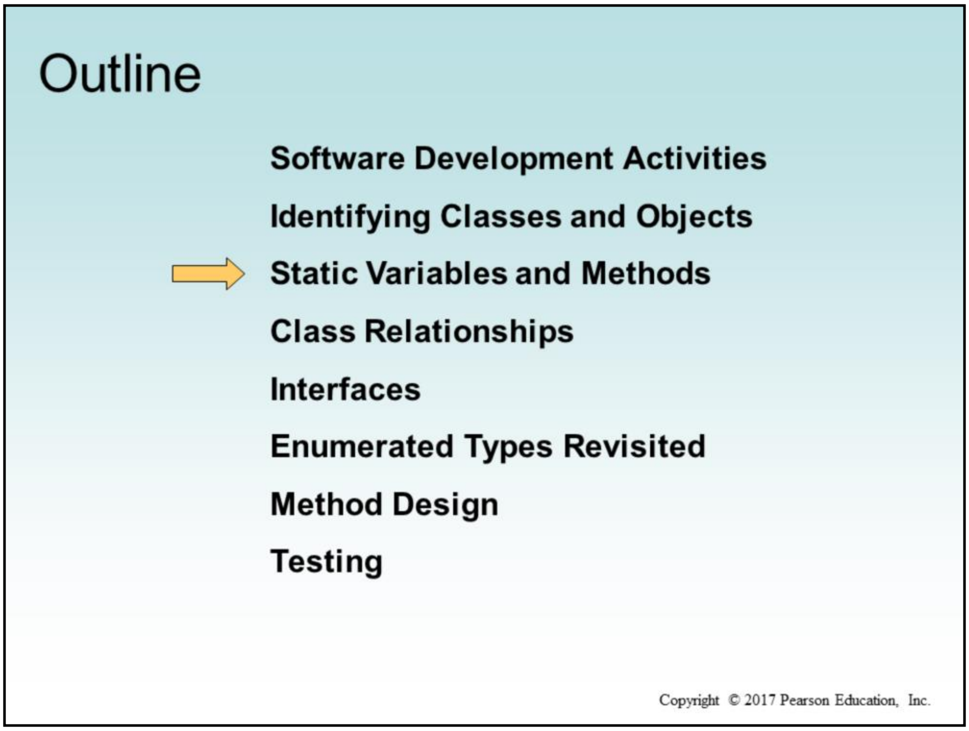
-Think of a program as a set of interconnecting modules (similar to Lego pieces)

-Each Lego piece has a function and purpose lending to the overall function of the program

-When we design a class, we are designing a specific Lego piece

-The great thing about object-oriented design is that we can reuse a Lego piece

-We might use one piece in the design of one program, then reuse it for another



Outline

Software Development Activities

Identifying Classes and Objects

⇨ Static Variables and Methods

Class Relationships

Interfaces

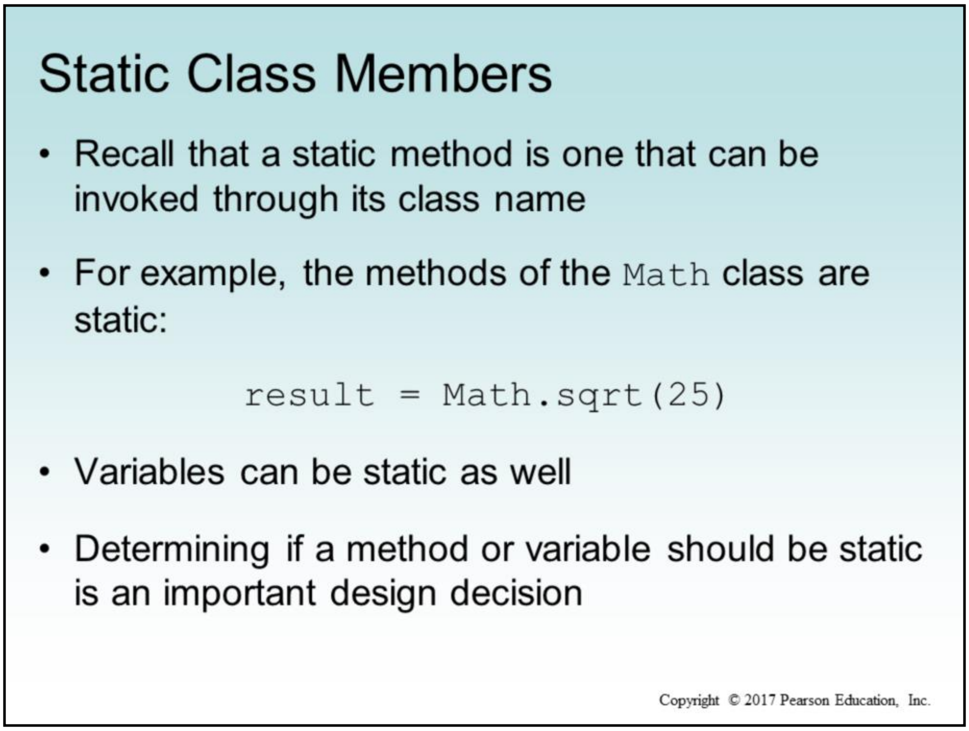
Enumerated Types Revisited

Method Design

Testing

-Next, let’s look at Java language considerations that go into the design and implementation of a class

-We begin by looking at the static modifier and its use with variables and methods

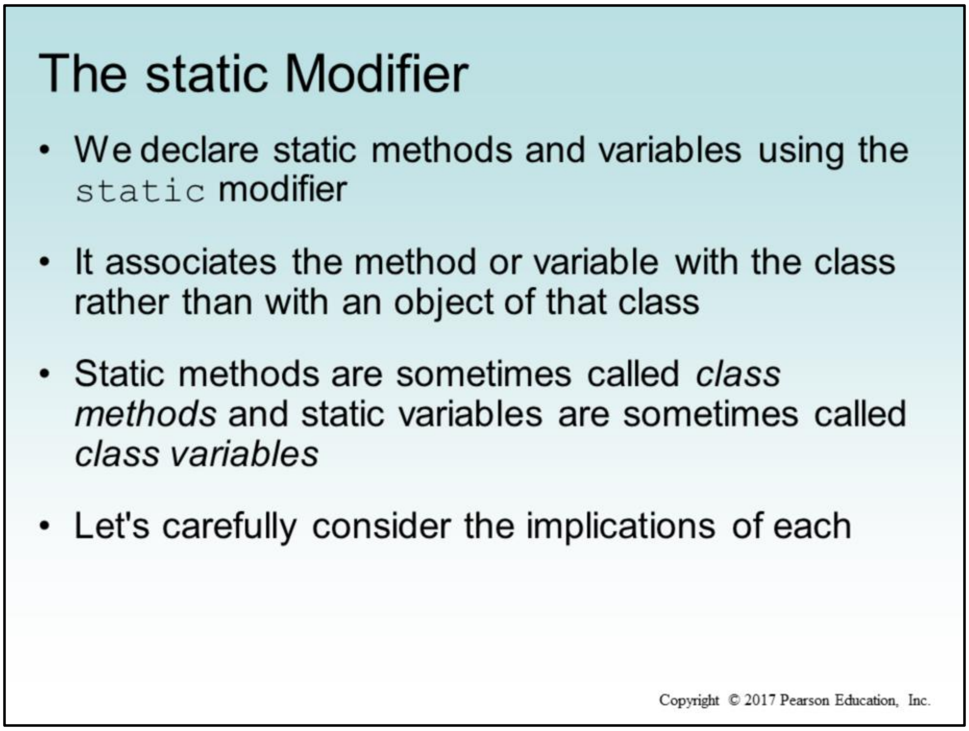


Static Class Members

- Recall that a static method is one that can be invoked through its class name
- For example, the methods of the `Math` class are static:

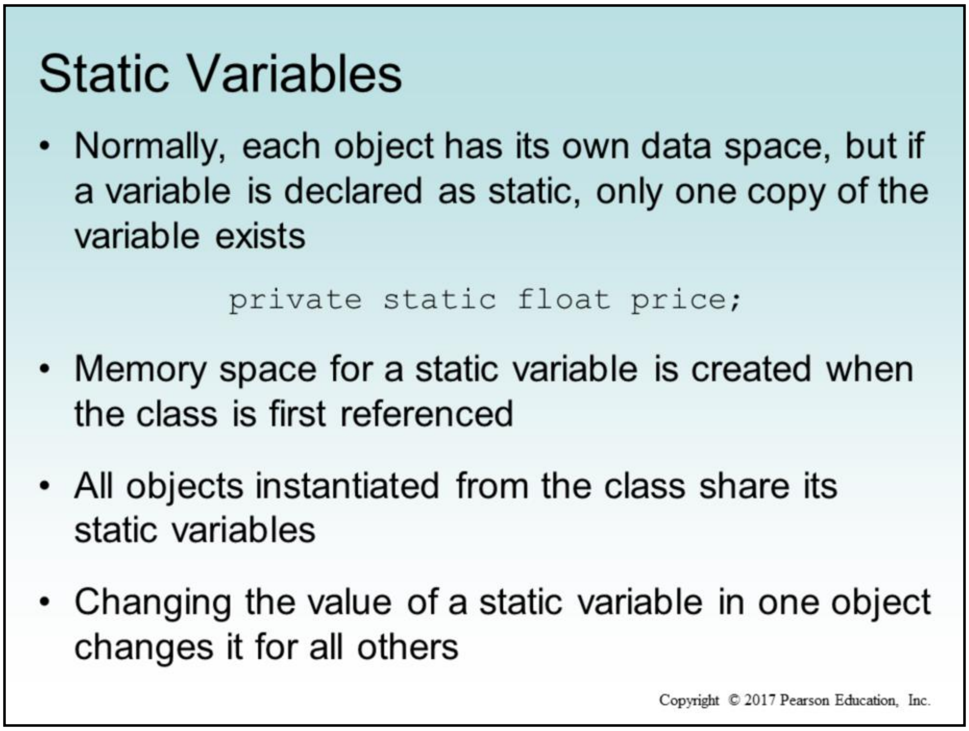
```
result = Math.sqrt(25)
```

- Variables can be static as well
- Determining if a method or variable should be static is an important design decision



The static Modifier

- We declare static methods and variables using the `static` modifier
- It associates the method or variable with the class rather than with an object of that class
- Static methods are sometimes called *class methods* and static variables are sometimes called *class variables*
- Let's carefully consider the implications of each



Static Variables

- Normally, each object has its own data space, but if a variable is declared as static, only one copy of the variable exists

```
private static float price;
```

- Memory space for a static variable is created when the class is first referenced
- All objects instantiated from the class share its static variables
- Changing the value of a static variable in one object changes it for all others

-Note that **multiple** copies of **instance** variables exist for multiple objects of the same class

-Each object gets a copy of an instance variable in memory

-However, only a **single** copy of a **static** variable exists for multiple objects of the same class

-Each object refers to the **same** static variable in memory

-In other words, multiple objects all **share** the same static variable

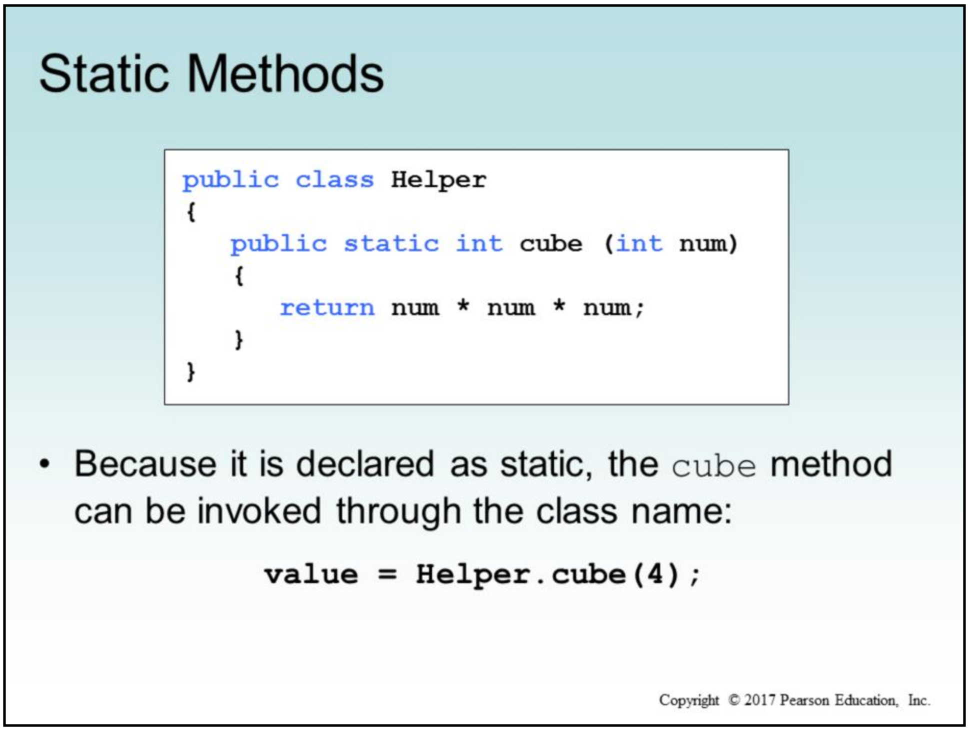
-Note that memory is allocated (reserved) for a static variable upon instantiation of the first object

-Local variables (those declared in a method) cannot be static

-Only class variables (those declared in the class and outside any method) can be static

-Note that **constants** are often defined as static (e.g. final static double PI)

-Since constants don’t allow their values changed, no need to have multiple copies



Static Methods

```
public class Helper
{
   public static int cube (int num)
   {
      return num * num * num;
   }
}
```

- Because it is declared as static, the `cube` method can be invoked through the class name:

```
value = Helper.cube(4);
```

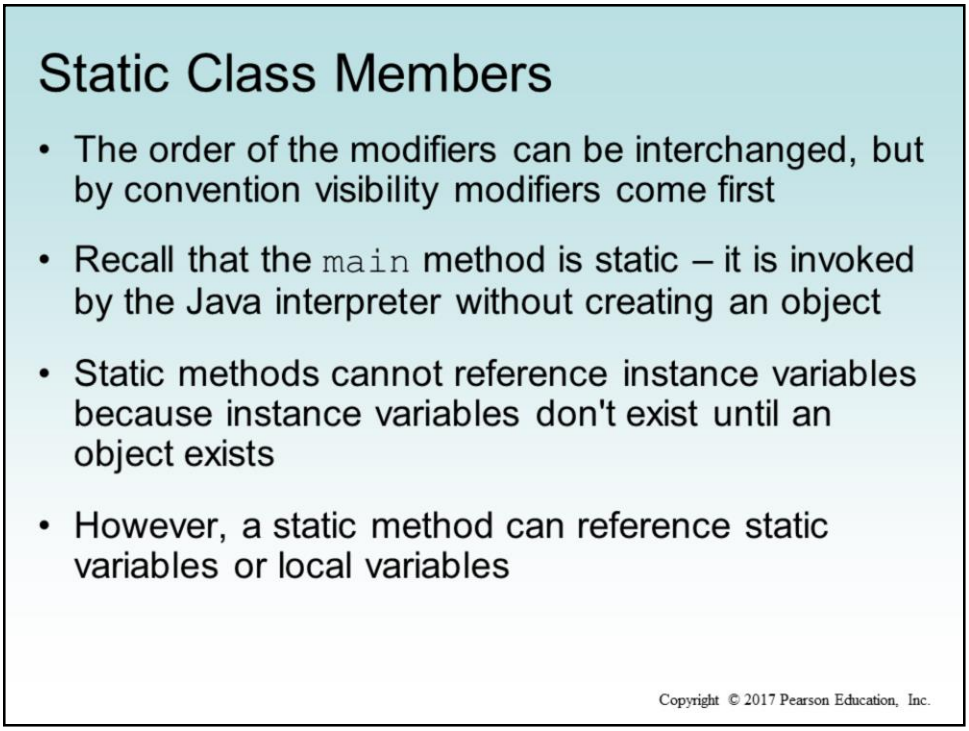
-Note that we do not need to create a new object to use static methods of a class

-Instead, we can call them using the name of the class, so instead of:

```
Math m = new Math();
m.sqrt(25);
```

-We can simply use:

```
Math.sqrt(25);
```



Static Class Members

- The order of the modifiers can be interchanged, but by convention visibility modifiers come first
- Recall that the `main` method is static – it is invoked by the Java interpreter without creating an object
- Static methods cannot reference instance variables because instance variables don't exist until an object exists
- However, a static method can reference static variables or local variables

-Recall visibility modifiers are public, private, and protected, these typically come first in a declaration

-Recall that the main method is static:

```
public static void main(String[] args)
```

-The main method is static so the program can call it without needing to create a new object

-Since static methods are not called through an object, they **cannot** use instance variables of a class

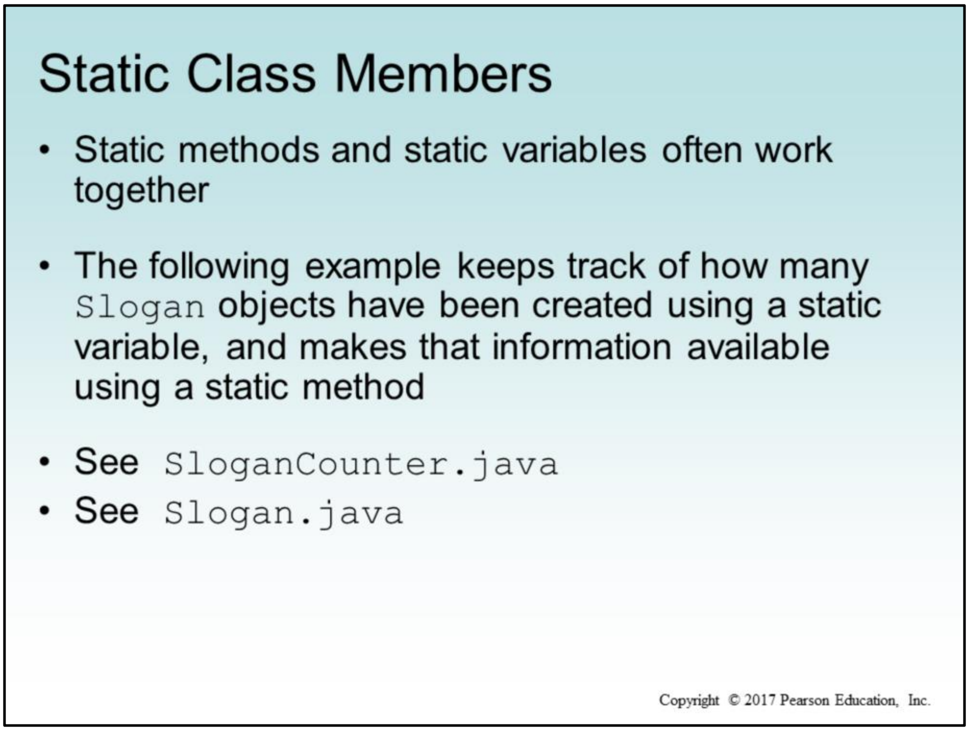
-A static method would not know with which object to associate the instance variables!

-An error indicating that a static method is attempting to use a non-static variable will result

-Static methods **can** reference static variables however

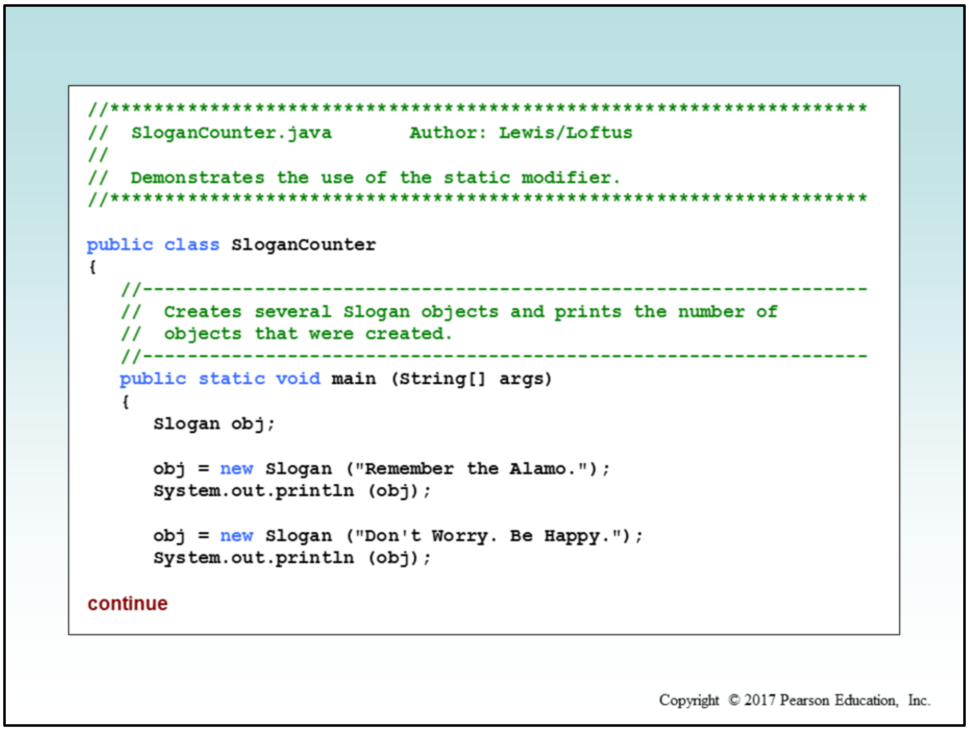
-As a static method, main cannot use any instance variables declared in the class

-This is why we typically do not declare instance variables in our “driver” classes



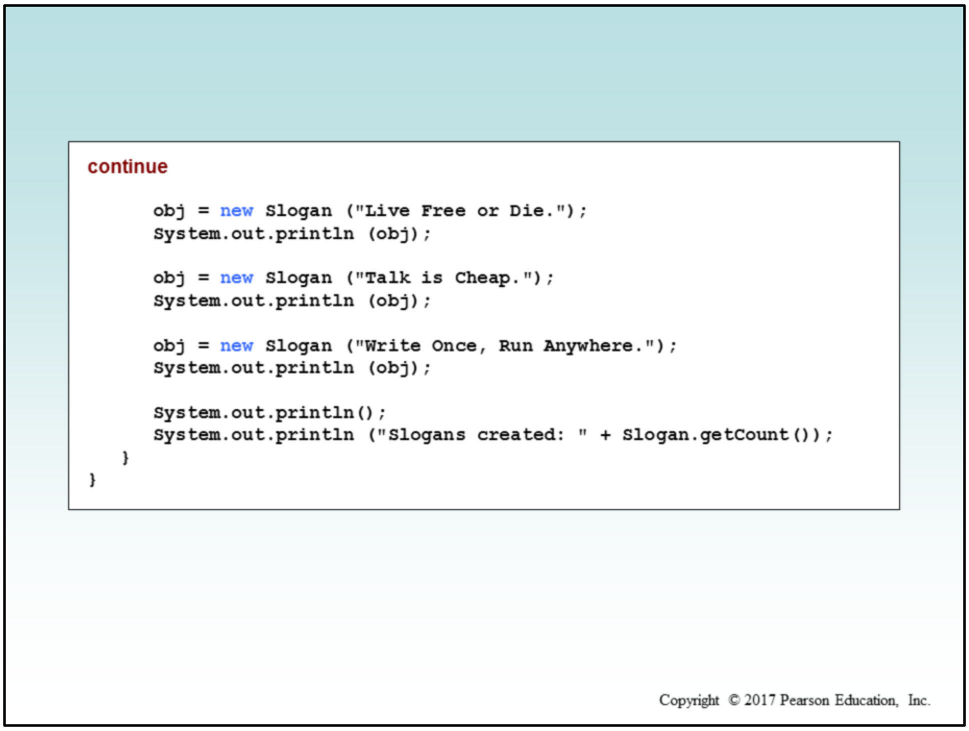
Static Class Members

- Static methods and static variables often work together
- The following example keeps track of how many `Slogan` objects have been created using a static variable, and makes that information available using a static method
- See `SloganCounter.java`
- See `Slogan.java`



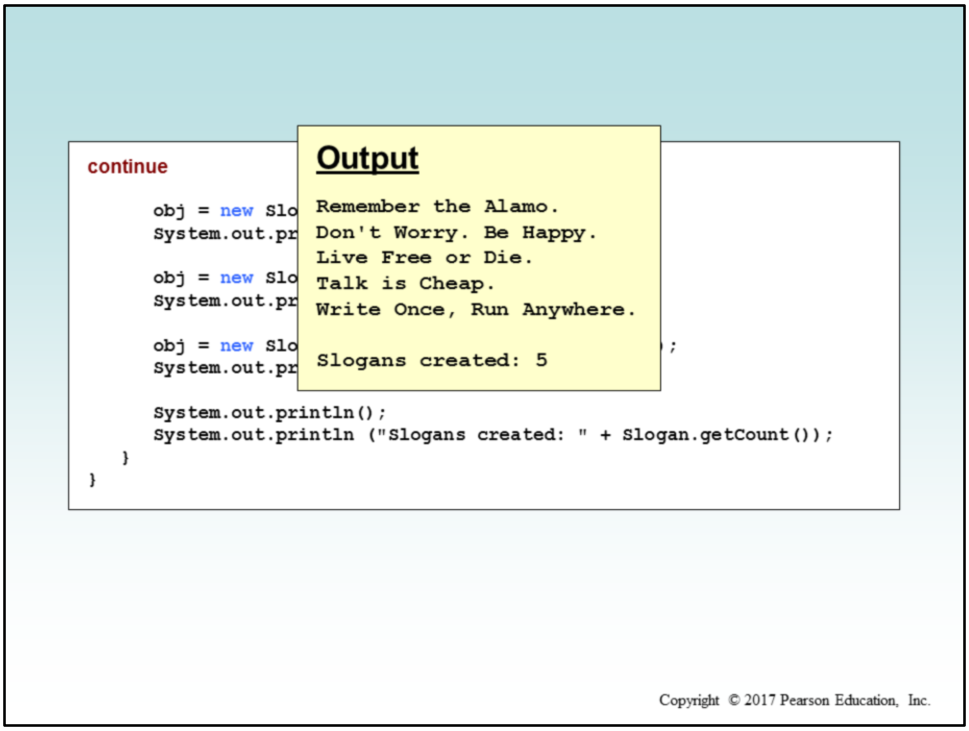
```java
//********************************************************************
//  SloganCounter.java       Author: Lewis/Loftus
//
//  Demonstrates the use of the static modifier.
//********************************************************************

public class SloganCounter
{
   //-----------------------------------------------------------------
   //  Creates several Slogan objects and prints the number of
   //  objects that were created.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      Slogan obj;

      obj = new Slogan ("Remember the Alamo.");
      System.out.println (obj);

      obj = new Slogan ("Don't Worry. Be Happy.");
      System.out.println (obj);
```

continue



continue

```
      obj = new Slogan ("Live Free or Die.");
      System.out.println (obj);

      obj = new Slogan ("Talk is Cheap.");
      System.out.println (obj);

      obj = new Slogan ("Write Once, Run Anywhere.");
      System.out.println (obj);

      System.out.println();
      System.out.println ("Slogans created: " + Slogan.getCount());
   }
}
```

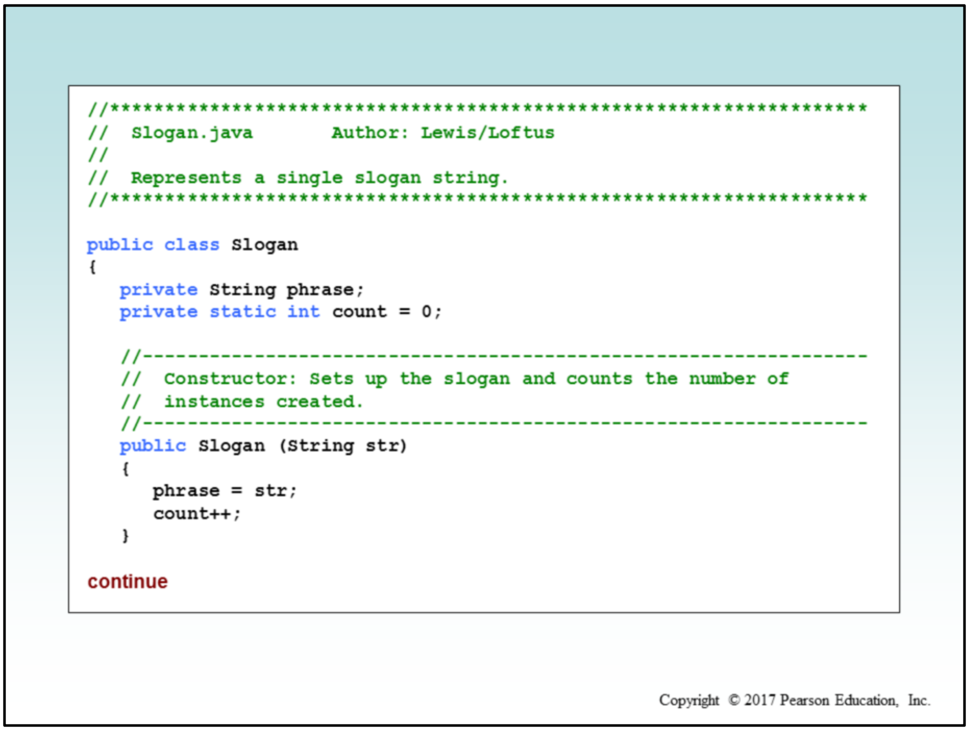


continue

```
      obj = new Slo
      System.out.pr

      obj = new Slo
      System.out.pr

      obj = new Slo                                   );
      System.out.pr

      System.out.println();
      System.out.println ("Slogans created: " + Slogan.getCount());
   }
}
```

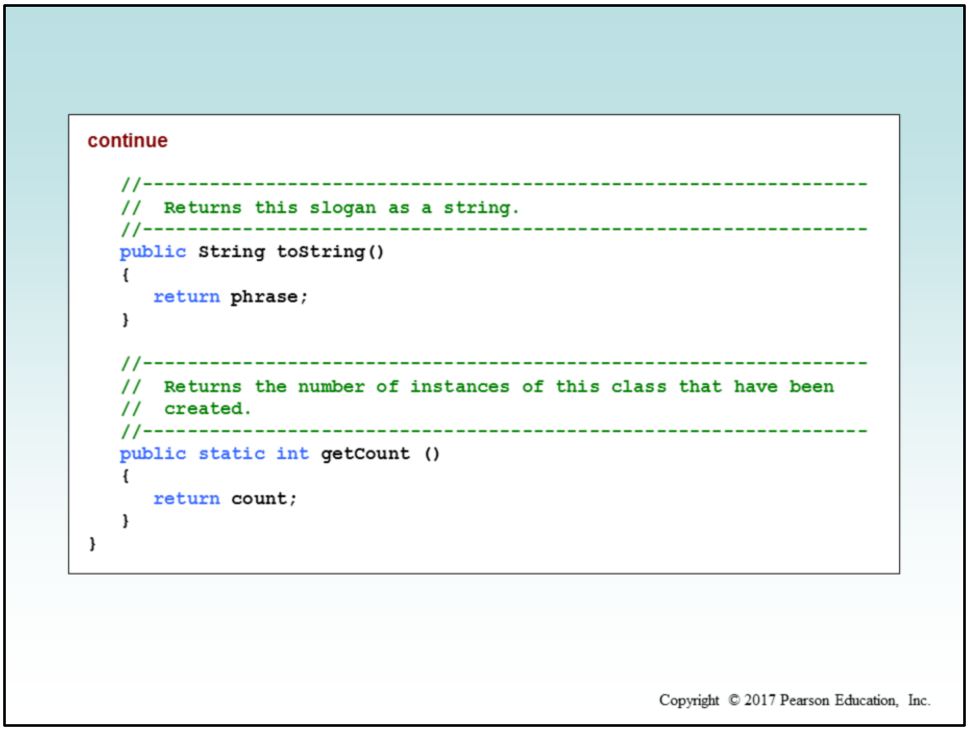
Output

```
Remember the Alamo.
Don't Worry. Be Happy.
Live Free or Die.
Talk is Cheap.
Write Once, Run Anywhere.

Slogans created: 5
```



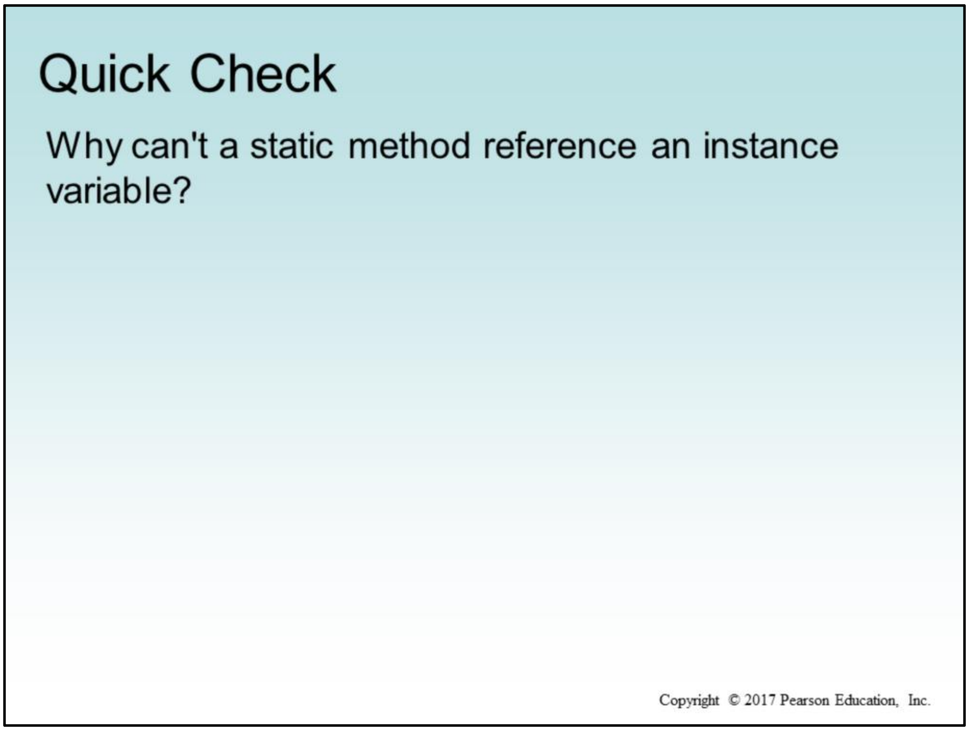
```java
//********************************************************************
//  Slogan.java       Author: Lewis/Loftus
//
//  Represents a single slogan string.
//********************************************************************

public class Slogan
{
   private String phrase;
   private static int count = 0;

   //-----------------------------------------------------------------
   //  Constructor: Sets up the slogan and counts the number of
   //  instances created.
   //-----------------------------------------------------------------
   public Slogan (String str)
   {
      phrase = str;
      count++;
   }
```

continue



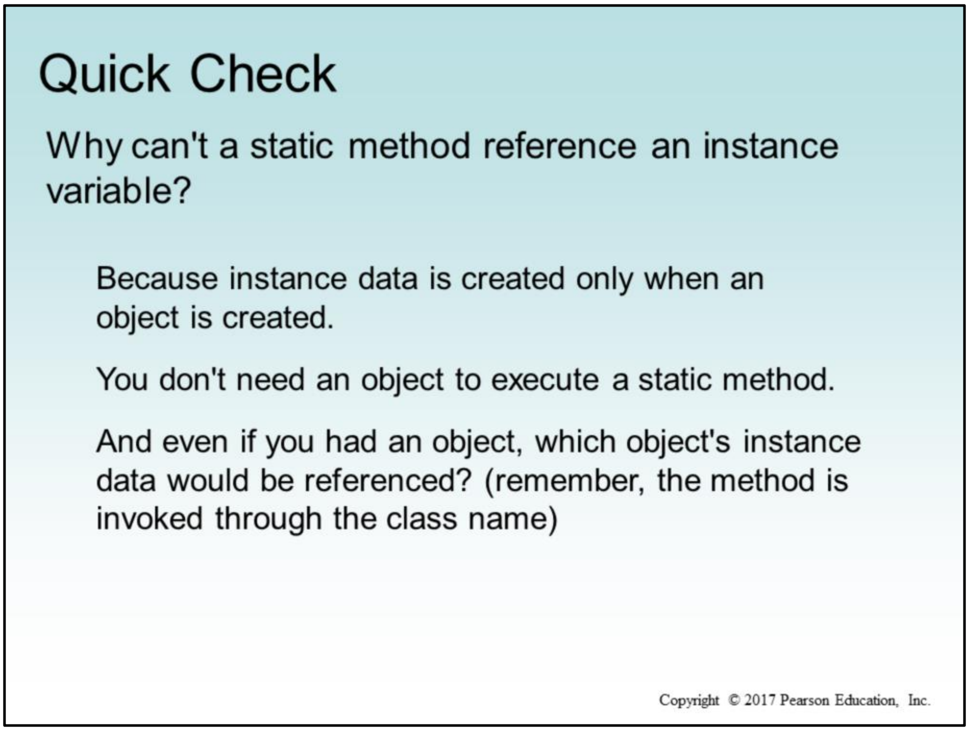
continue

```
   //----------------------------------------------------------------
   //  Returns this slogan as a string.
   //----------------------------------------------------------------
   public String toString()
   {
      return phrase;
   }

   //----------------------------------------------------------------
   //  Returns the number of instances of this class that have been
   //  created.
   //----------------------------------------------------------------
   public static int getCount ()
   {
      return count;
   }
}
```



Quick Check

Why can't a static method reference an instance variable?



Quick Check

Why can't a static method reference an instance variable?

Because instance data is created only when an object is created.

You don't need an object to execute a static method.

And even if you had an object, which object's instance data would be referenced? (remember, the method is invoked through the class name)