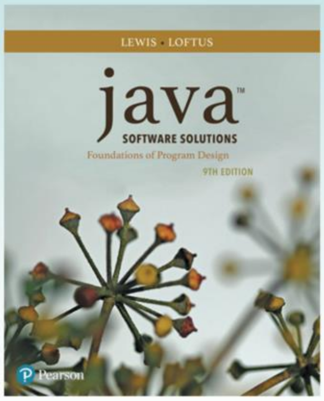
Chapter 3
Using Classes and Objects

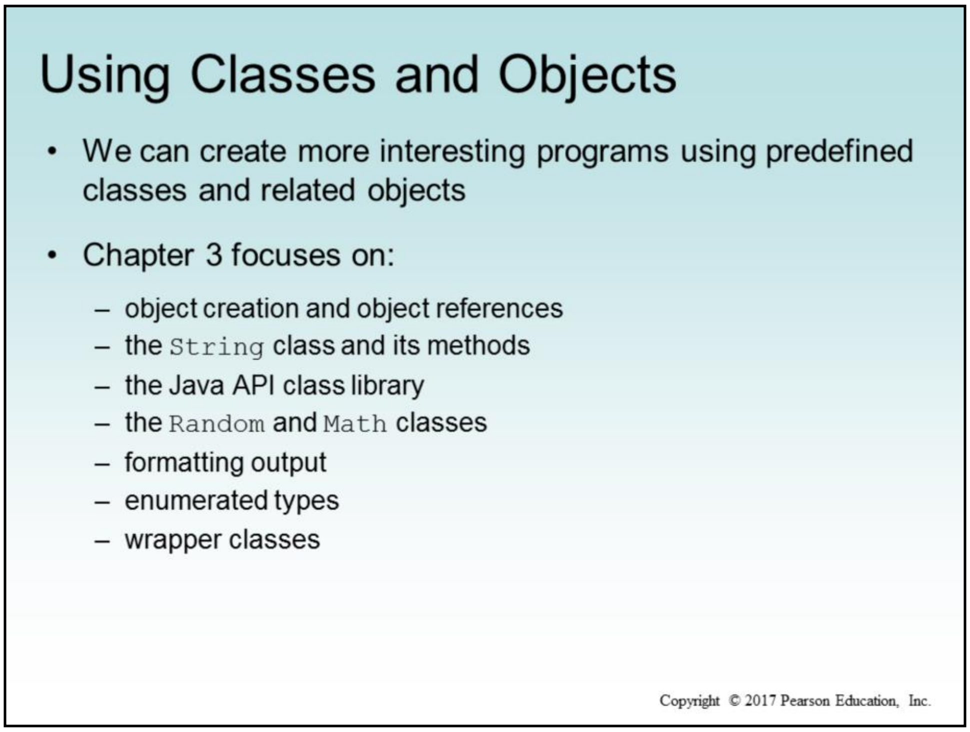


Java Software Solutions

Foundations of Program Design

9th Edition

John Lewis
William Loftus



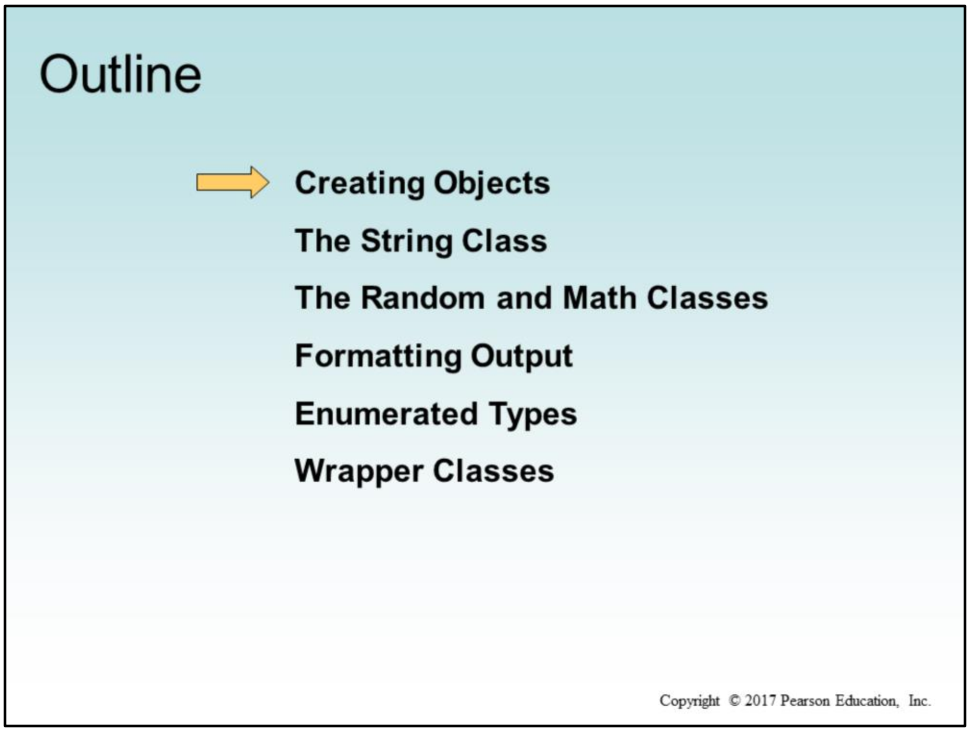
Using Classes and Objects

- We can create more interesting programs using predefined classes and related objects
- Chapter 3 focuses on:
  - object creation and object references
  - the `String` class and its methods
  - the Java API class library
  - the `Random` and `Math` classes
  - formatting output
  - enumerated types
  - wrapper classes

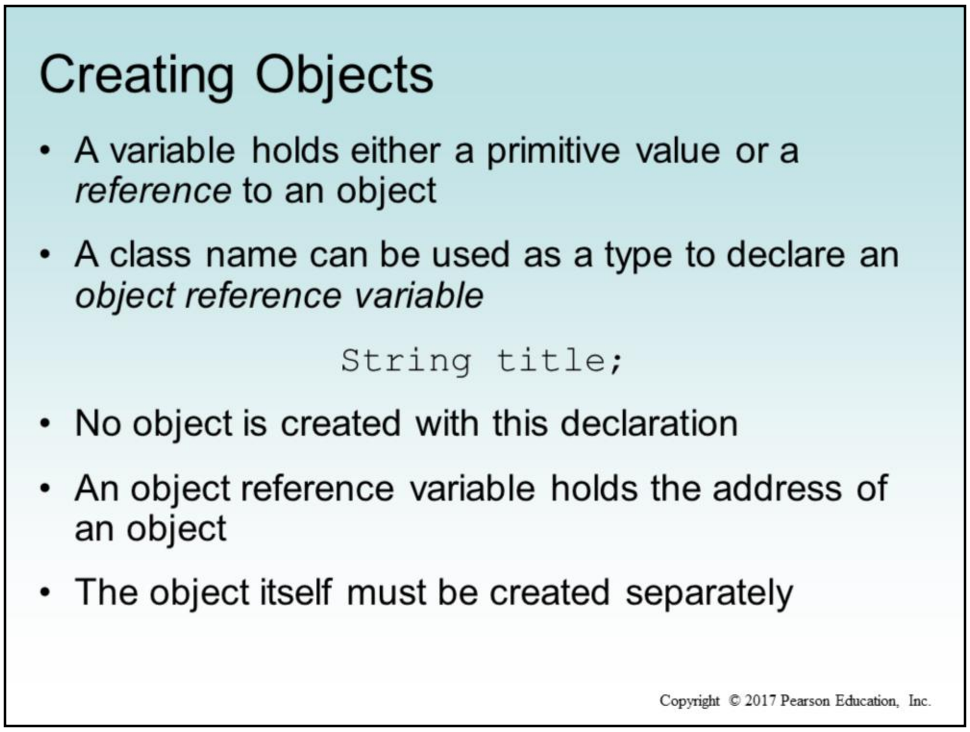
-In this chapter, we'll continue to study and use Java classes in our program

-Recall we've already learned about the String and Scanner class in previous chapters

-In this chapter, we'll be introduced to several new classes we can use in our programs



Outline



Creating Objects

- A variable holds either a primitive value or a *reference* to an object
- A class name can be used as a type to declare an *object reference variable*

```
String title;
```

- No object is created with this declaration
- An object reference variable holds the address of an object
- The object itself must be created separately

-The memory reserved for a **primitive** variable (e.g. int, double) stores the actual data

-The memory reserved for an **object** variable (e.g. String, Scanner) stores another address

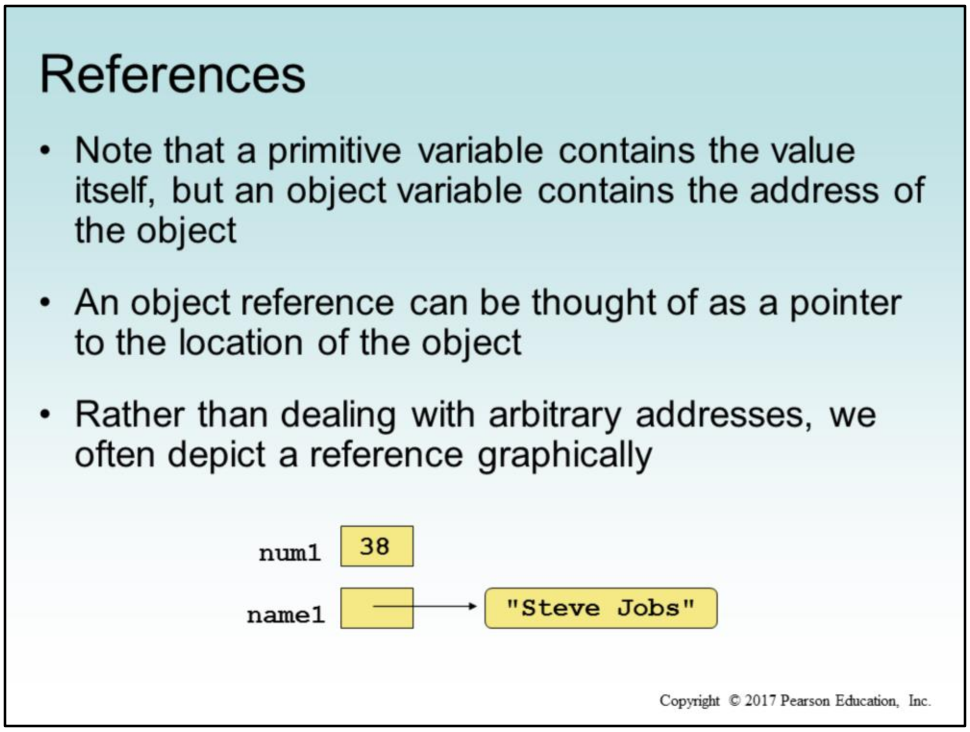
-This address is another area in memory storing the actual data that lives in the object

-In other words, this address, **refers** to the actual object living somewhere else in memory

-For this reason, an object variable is called an **object reference variable**

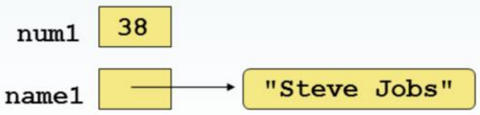
-We can also call an object reference variable a **pointer** since it points to another address

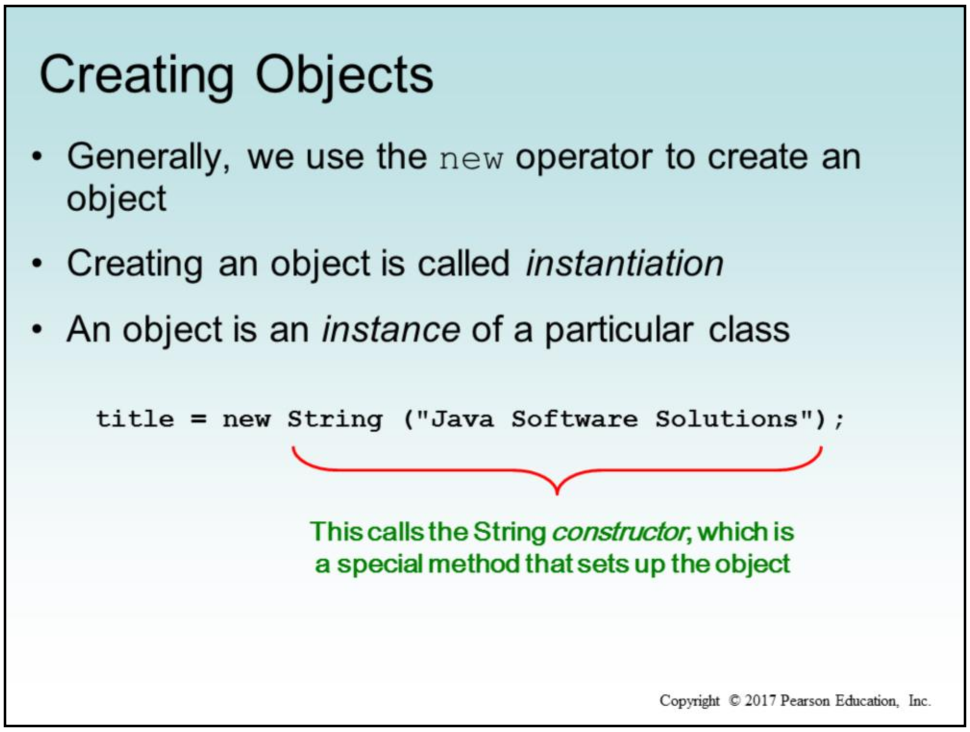
-In other words, one memory location contains an address to another memory address



References

- Note that a primitive variable contains the value itself, but an object variable contains the address of the object
- An object reference can be thought of as a pointer to the location of the object
- Rather than dealing with arbitrary addresses, we often depict a reference graphically





Creating Objects

- Generally, we use the `new` operator to create an object
- Creating an object is called *instantiation*
- An object is an *instance* of a particular class

```
title = new String ("Java Software Solutions");
```

$\underbrace{\texttt{new String ("Java Software Solutions")}}_{\text{This calls the String constructor, which is a special method that sets up the object}}$

This calls the String *constructor*, which is a special method that sets up the object

-When the **new** operator is called, memory is reserved and its address stored in the object reference variable

-We call this **instantiation** (an object is created from the class blueprint)

-We say that an object is an **instance** of a class

-After memory has been created, the class constructor method is then called

-The purpose of this method is to complete the **construction** of the object in memory

-This involves such tasks as initializing information, opening files, allocating more memory, …

-Note that constructor methods have the same name as the class

-Note that they are specified along with the new operator

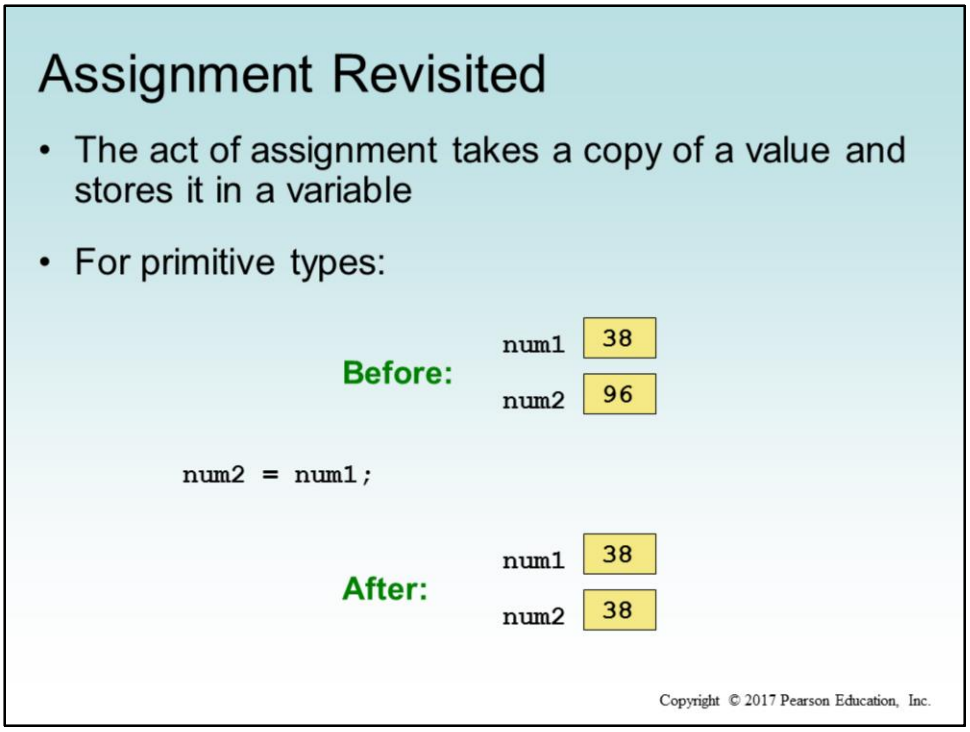
Invoking Methods

- We've seen that once an object has been instantiated, we can use the *dot operator* to invoke its methods

```
numChars = title.length()
```

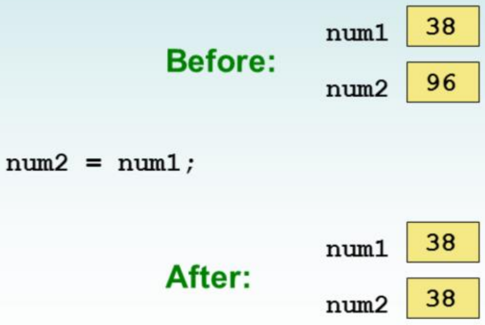
- A method may *return a value*, which can be used in an assignment or expression

- A method invocation can be thought of as asking an object to perform a service



Assignment Revisited

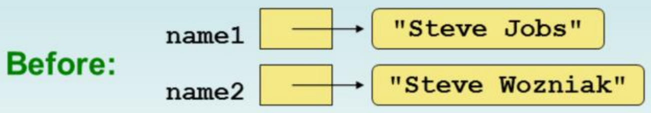
- The act of assignment takes a copy of a value and stores it in a variable
- For primitive types:



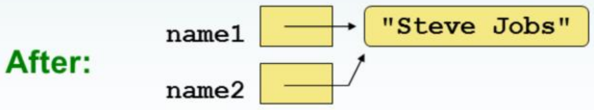
```
num2 = num1;
```

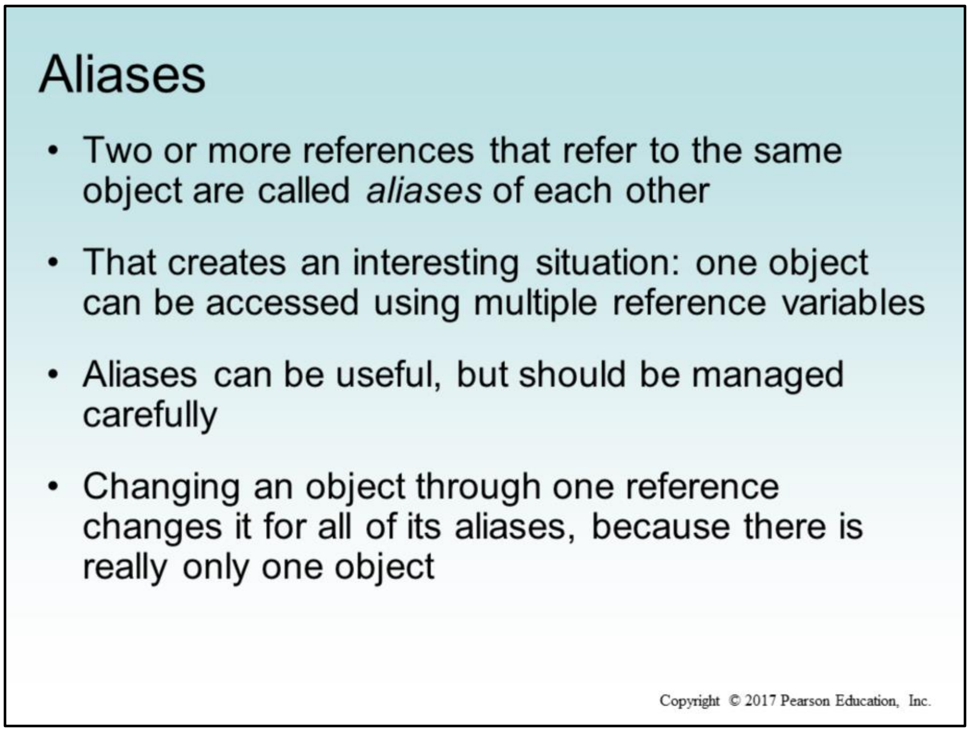
Reference Assignment

- For object references, assignment copies the address:



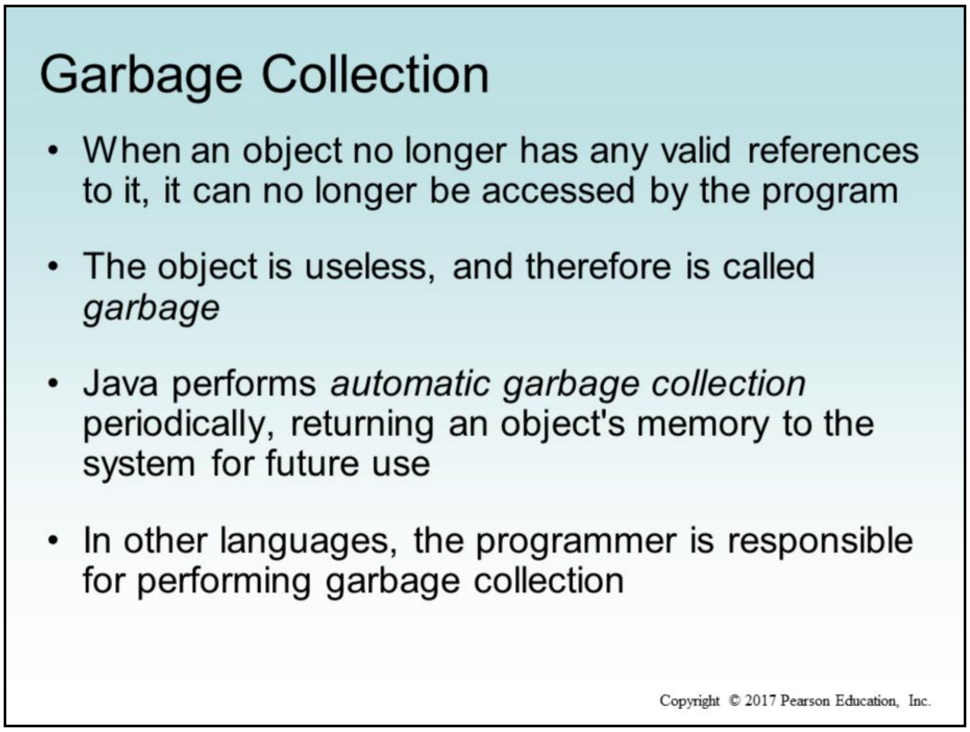
```
name2 = name1;
```





Aliases

- Two or more references that refer to the same object are called *aliases* of each other
- That creates an interesting situation: one object can be accessed using multiple reference variables
- Aliases can be useful, but should be managed carefully
- Changing an object through one reference changes it for all of its aliases, because there is really only one object



Garbage Collection

- When an object no longer has any valid references to it, it can no longer be accessed by the program
- The object is useless, and therefore is called *garbage*
- Java performs *automatic garbage collection* periodically, returning an object's memory to the system for future use
- In other languages, the programmer is responsible for performing garbage collection

-Garbage is memory allocated for an object that is no longer referenced

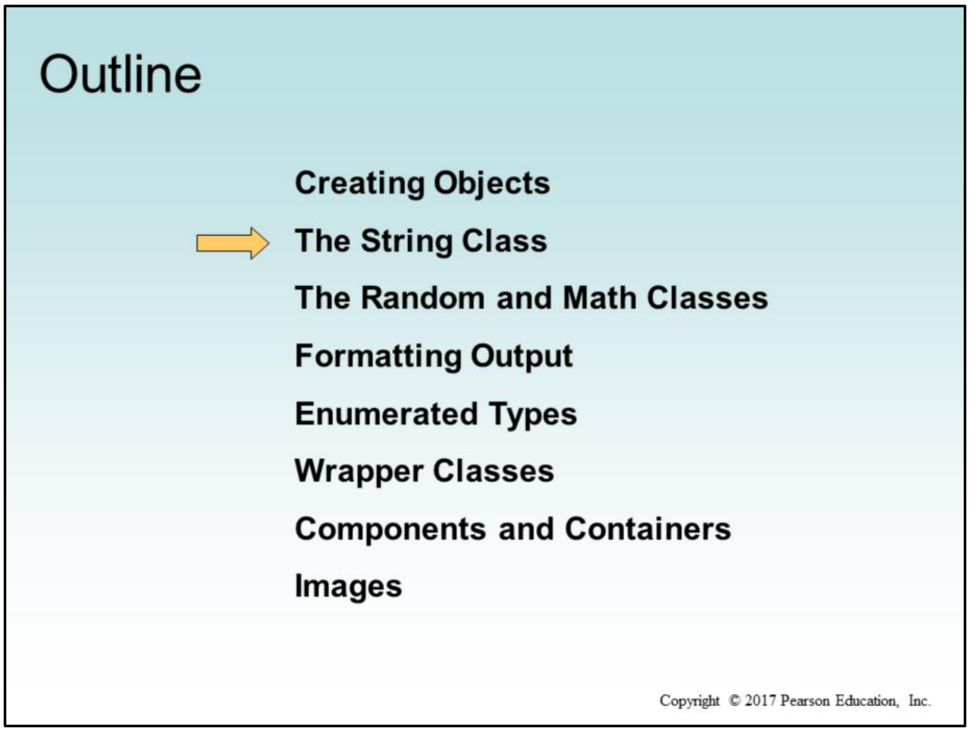
-Java will automatically free up such memory for future use

-In C++, the programmer must keep track of memory allocated

-If the memory is no longer used, the C++ programmer must free it

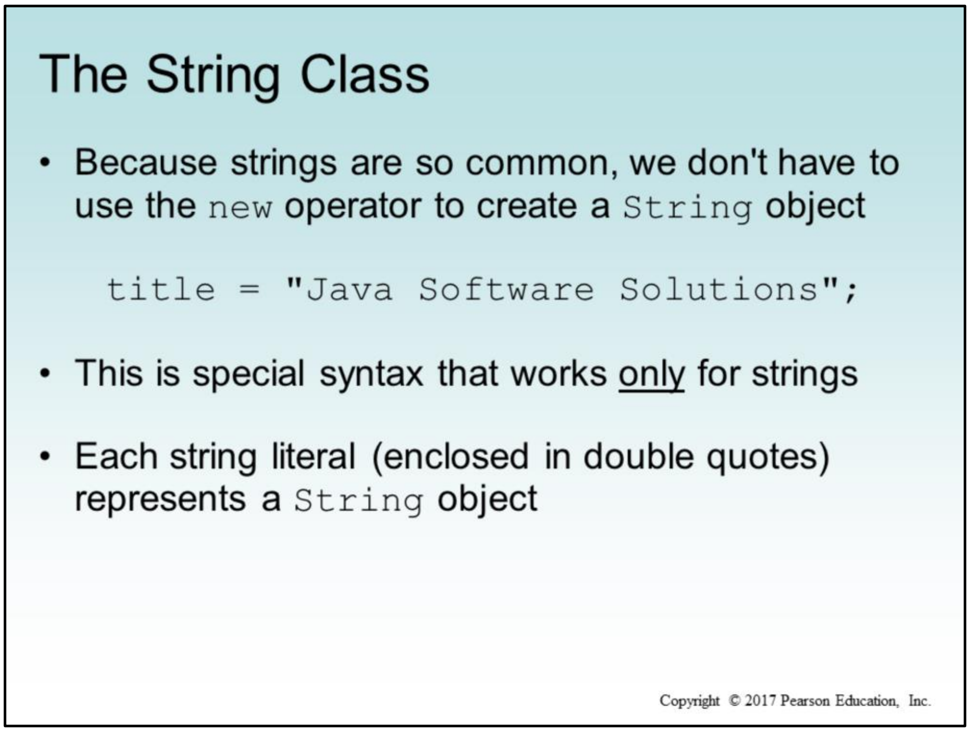
-Automatic garbage collection does not exist in C++

-This is a wonderful feature of Java!



Outline

- Creating Objects
- ➡ The String Class
- The Random and Math Classes
- Formatting Output
- Enumerated Types
- Wrapper Classes
- Components and Containers
- Images

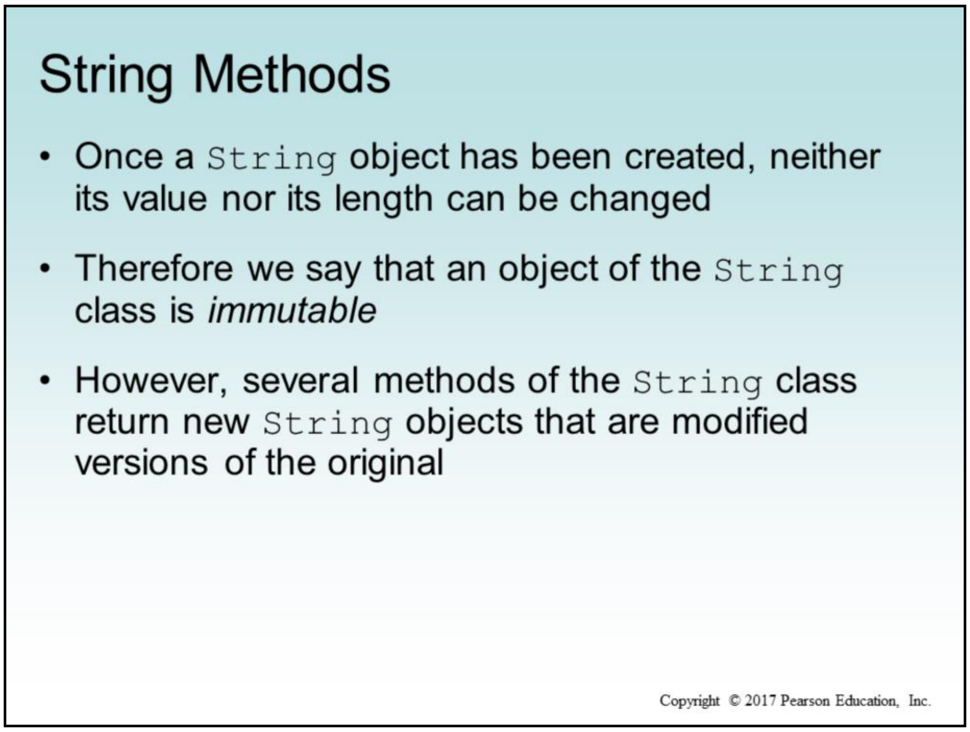


The String Class

- Because strings are so common, we don't have to use the `new` operator to create a `String` object

```
title = "Java Software Solutions";
```

- This is special syntax that works <u>only</u> for strings
- Each string literal (enclosed in double quotes) represents a `String` object



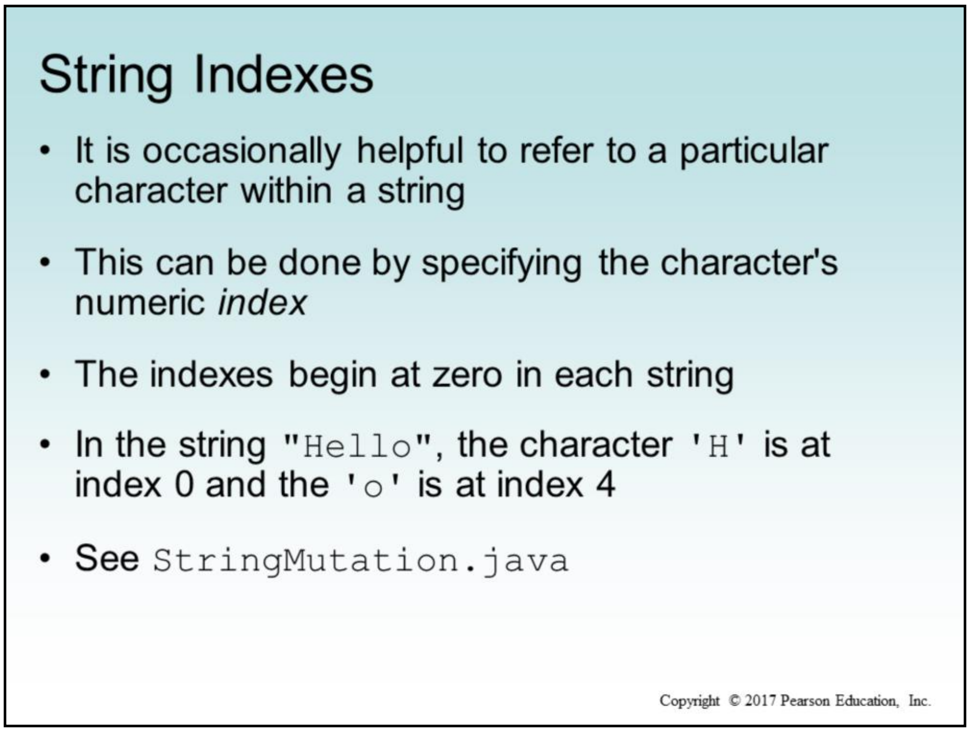
String Methods

- Once a `String` object has been created, neither its value nor its length can be changed
- Therefore we say that an object of the `String` class is *immutable*
- However, several methods of the `String` class return new `String` objects that are modified versions of the original

-**Immutable**: state of the object cannot be changed

-When a String method returns another String object, the original was never changed

-Such methods create new String objects **based on** the original String

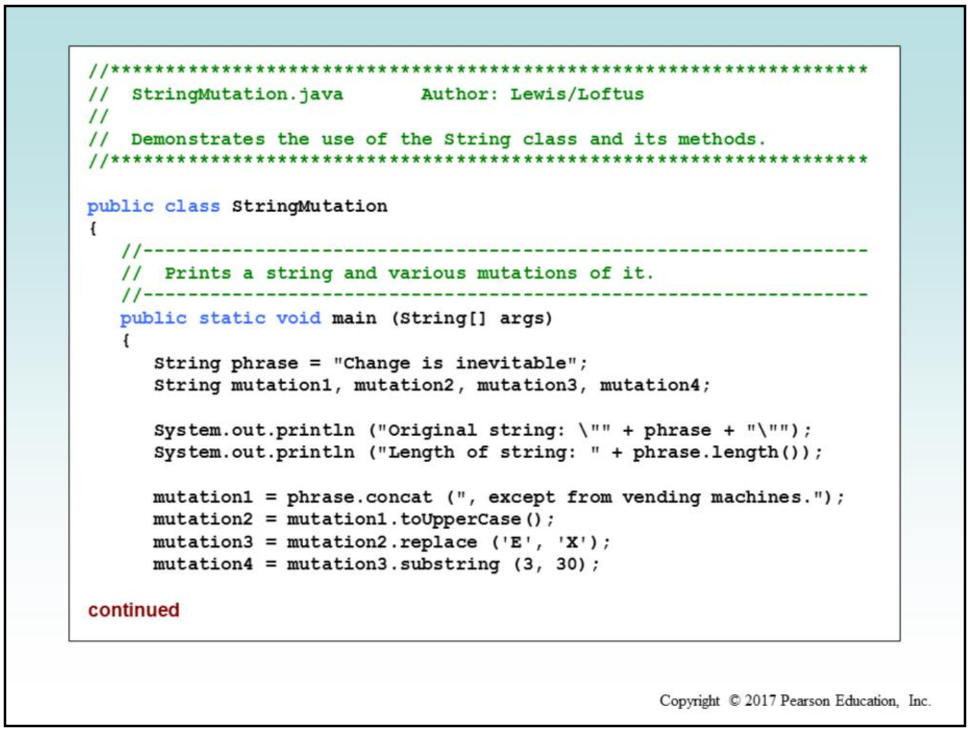


String Indexes

- It is occasionally helpful to refer to a particular character within a string
- This can be done by specifying the character's numeric *index*
- The indexes begin at zero in each string
- In the string `"Hello"`, the character `'H'` is at index 0 and the `'o'` is at index 4
- See `StringMutation.java`

-Note character index starts at 0, **not** 1

-Indices range from 0 to (number of characters in String – 1)



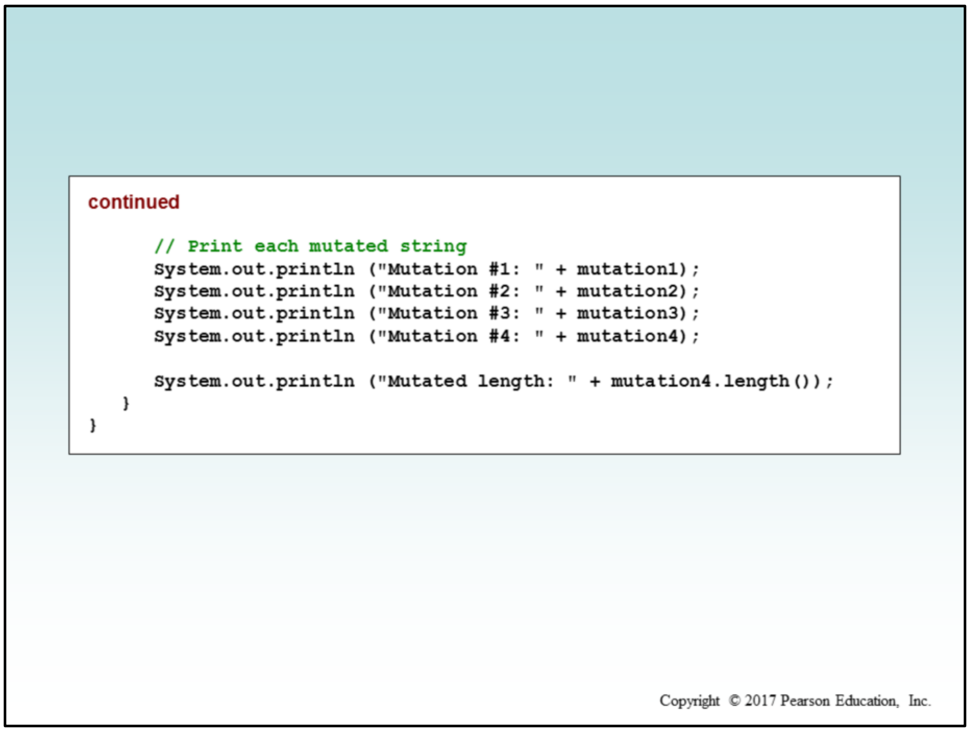
```
//********************************************************************
//  StringMutation.java       Author: Lewis/Loftus
//
//  Demonstrates the use of the String class and its methods.
//********************************************************************

public class StringMutation
{
   //-----------------------------------------------------------------
   //  Prints a string and various mutations of it.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      String phrase = "Change is inevitable";
      String mutation1, mutation2, mutation3, mutation4;

      System.out.println ("Original string: \"" + phrase + "\"");
      System.out.println ("Length of string: " + phrase.length());

      mutation1 = phrase.concat (", except from vending machines.");
      mutation2 = mutation1.toUpperCase();
      mutation3 = mutation2.replace ('E', 'X');
      mutation4 = mutation3.substring (3, 30);

continued
```

-Demonstrates using some of the String methods (listed in Fig. 3.1)

-Note that methods such as concat, toUpperCase, replace actually return new String objects

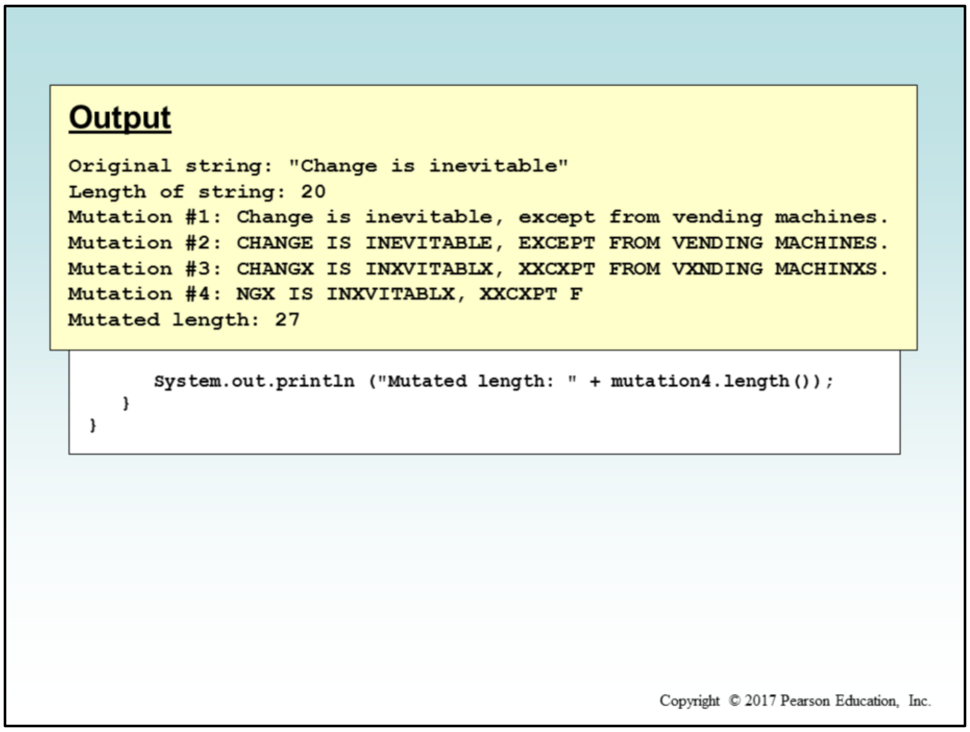
-Note that methods operate on the **object that called the method**

-For example, mutation1.toUpperCase() operates on the object mutation1



continued

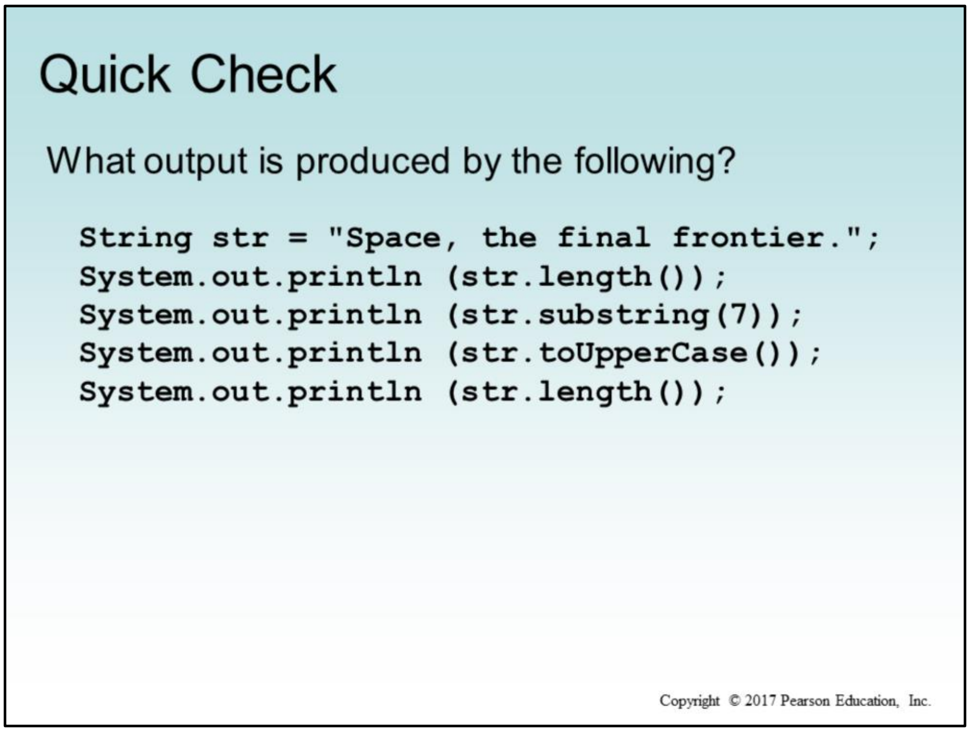
```
      // Print each mutated string
      System.out.println ("Mutation #1: " + mutation1);
      System.out.println ("Mutation #2: " + mutation2);
      System.out.println ("Mutation #3: " + mutation3);
      System.out.println ("Mutation #4: " + mutation4);

      System.out.println ("Mutated length: " + mutation4.length());
   }
}
```



Output

```
Original string: "Change is inevitable"
Length of string: 20
Mutation #1: Change is inevitable, except from vending machines.
Mutation #2: CHANGE IS INEVITABLE, EXCEPT FROM VENDING MACHINES.
Mutation #3: CHANGX IS INXVITABLX, XXCXPT FROM VXNDING MACHINXS.
Mutation #4: NGX IS INXVITABLX, XXCXPT F
Mutated length: 27
```

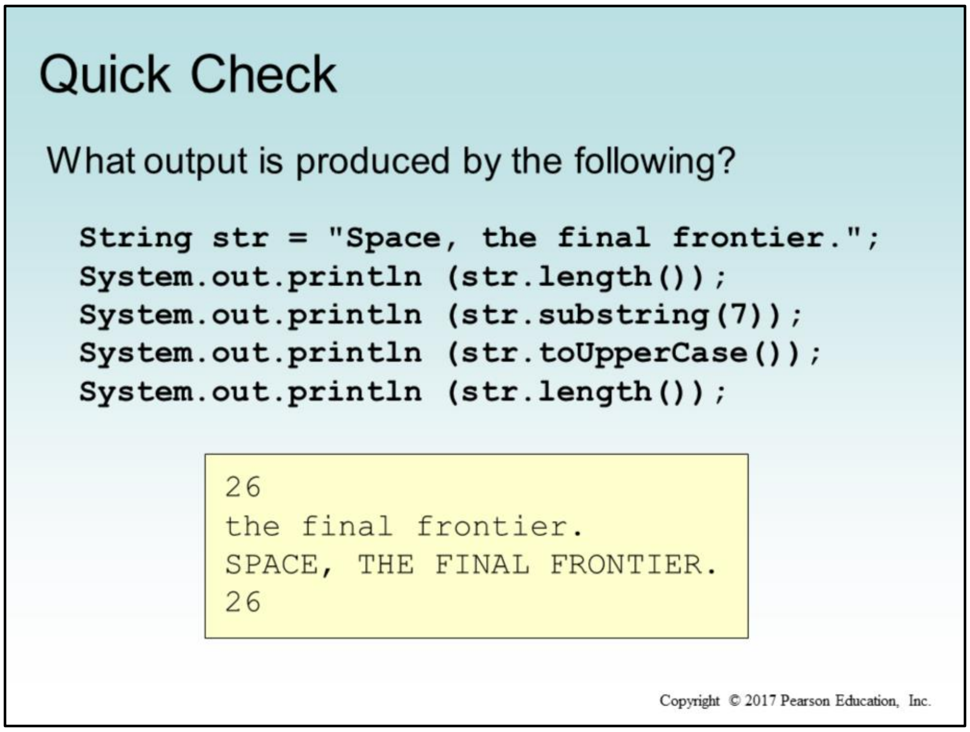
```
      System.out.println ("Mutated length: " + mutation4.length());
   }
}
```



Quick Check

What output is produced by the following?

```
String str = "Space, the final frontier.";
System.out.println (str.length());
System.out.println (str.substring(7));
System.out.println (str.toUpperCase());
System.out.println (str.length());
```

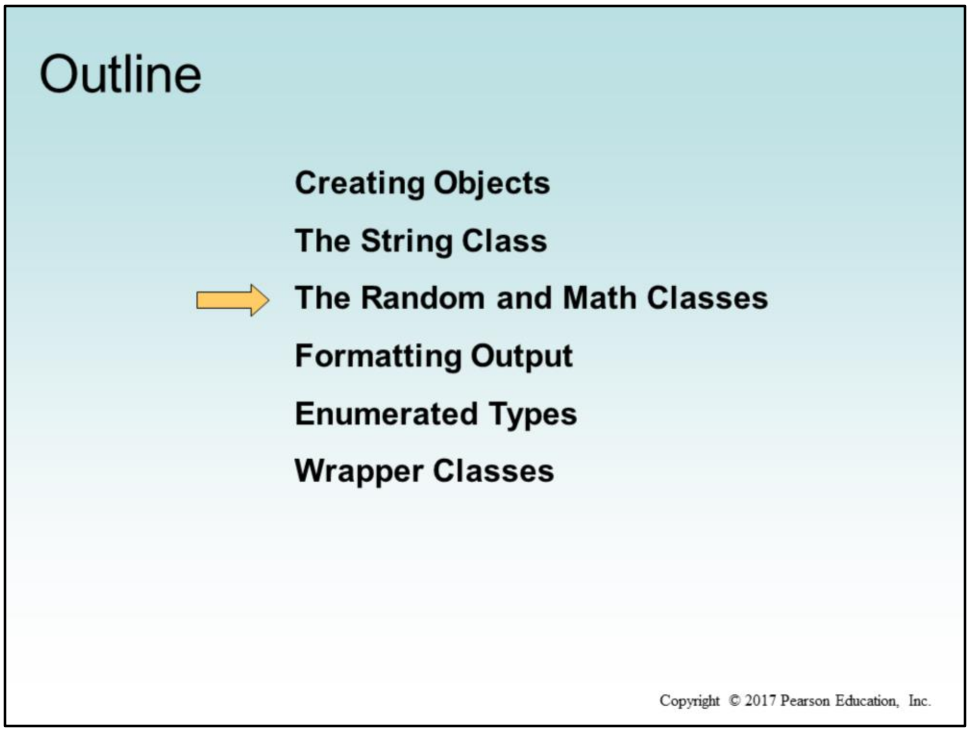


Quick Check

What output is produced by the following?

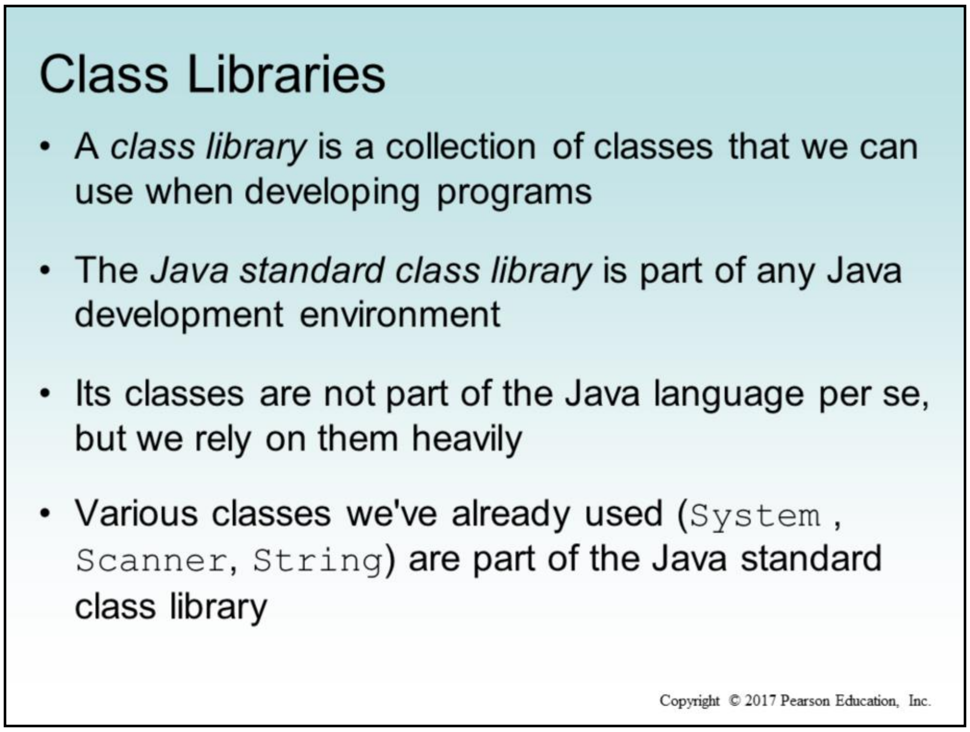
```
String str = "Space, the final frontier.";
System.out.println (str.length());
System.out.println (str.substring(7));
System.out.println (str.toUpperCase());
System.out.println (str.length());
```

```
26
the final frontier.
SPACE, THE FINAL FRONTIER.
26
```



Outline

- Creating Objects
- The String Class
- ➡ The Random and Math Classes
- Formatting Output
- Enumerated Types
- Wrapper Classes

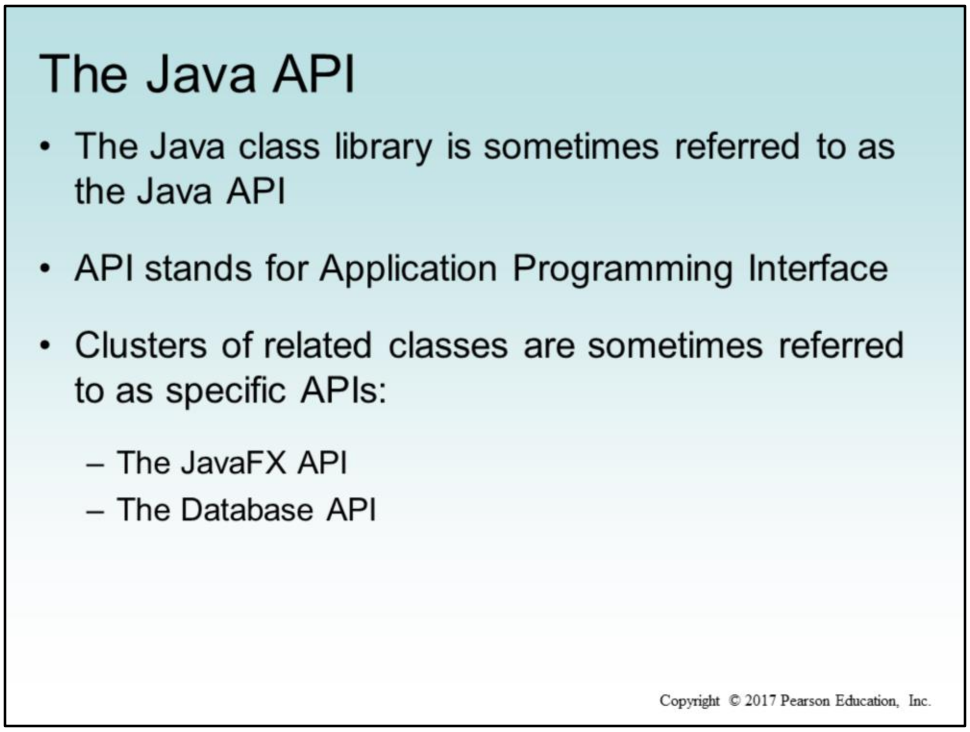


Class Libraries

- A *class library* is a collection of classes that we can use when developing programs
- The *Java standard class library* is part of any Java development environment
- Its classes are not part of the Java language per se, but we rely on them heavily
- Various classes we've already used (`System` , `Scanner`, `String`) are part of the Java standard class library

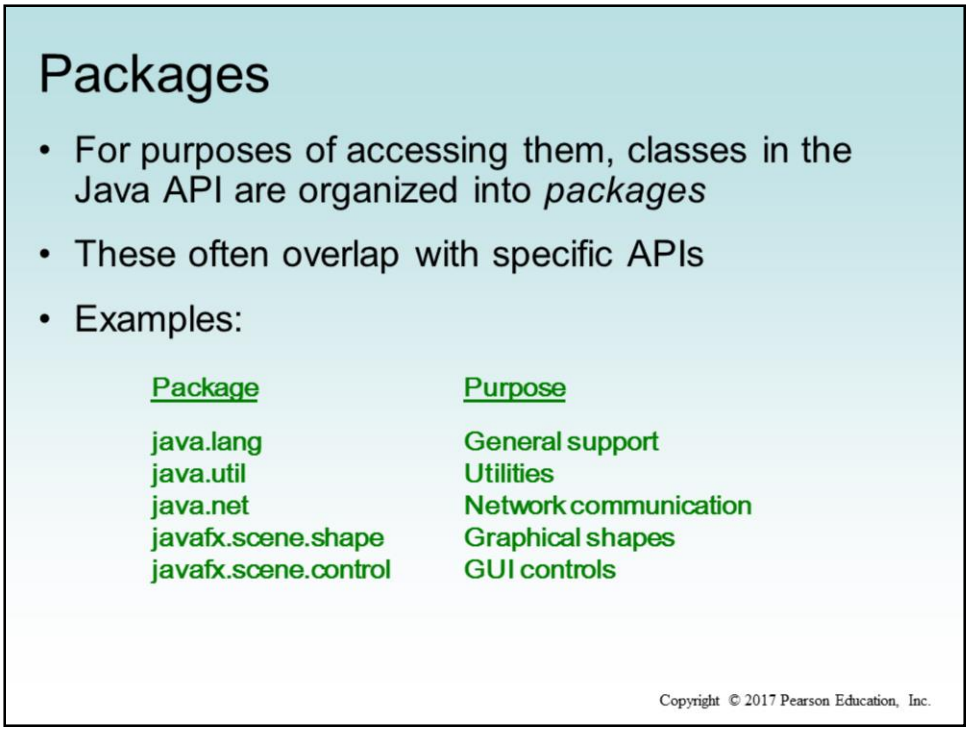
-Classes in the Java standard library were written by someone else for us to use!

-They are not part of the language, they are classes built **using** the Java language



The Java API

- The Java class library is sometimes referred to as the Java API
- API stands for Application Programming Interface
- Clusters of related classes are sometimes referred to as specific APIs:
  - The JavaFX API
  - The Database API



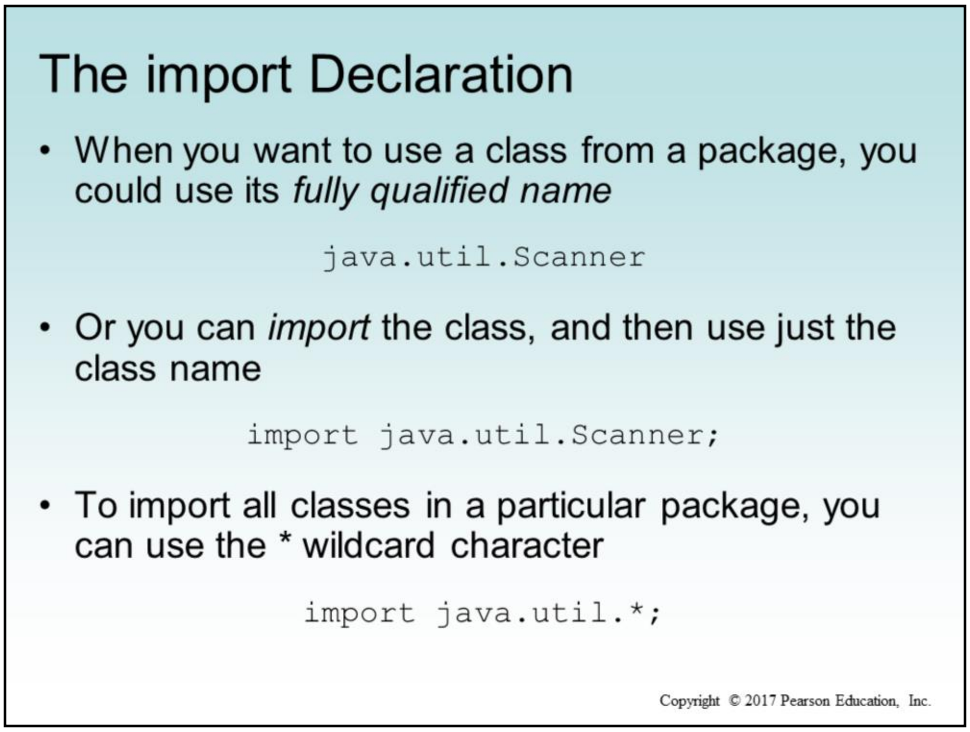
Packages

- For purposes of accessing them, classes in the Java API are organized into *packages*
- These often overlap with specific APIs
- Examples:

Package	Purpose
java.lang	General support
java.util	Utilities
java.net	Network communication
javafx.scene.shape	Graphical shapes
javafx.scene.control	GUI controls

-The String class, for example, is contained in the java.lang package of the API

-The Scanner class, for example, is contained in the java.util package of the API



The import Declaration

- When you want to use a class from a package, you could use its *fully qualified name*

```
java.util.Scanner
```

- Or you can *import* the class, and then use just the class name

```
import java.util.Scanner;
```

- To import all classes in a particular package, you can use the * wildcard character

```
import java.util.*;
```

-Using an import statement prevents us from using the fully-qualified name of a class

-If we didn't use the import statement, `import java.util.Scanner`

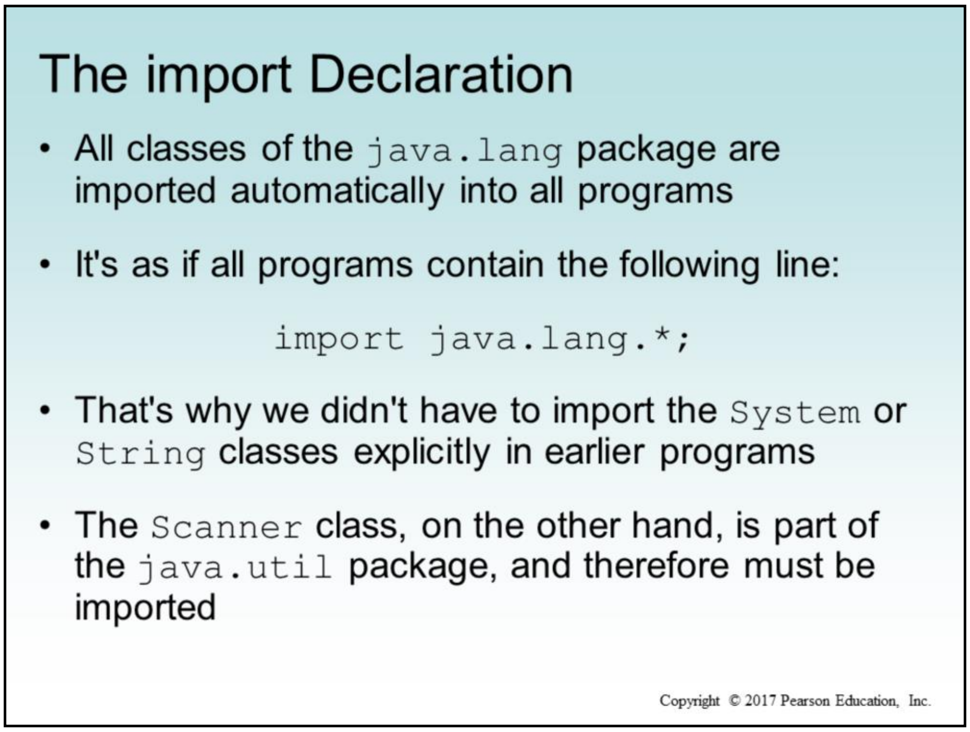
-We would have to refer to the fully-qualified Scanner class as java.util.Scanner

-For example, instead of:

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

-We would have to specify:

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



The import Declaration

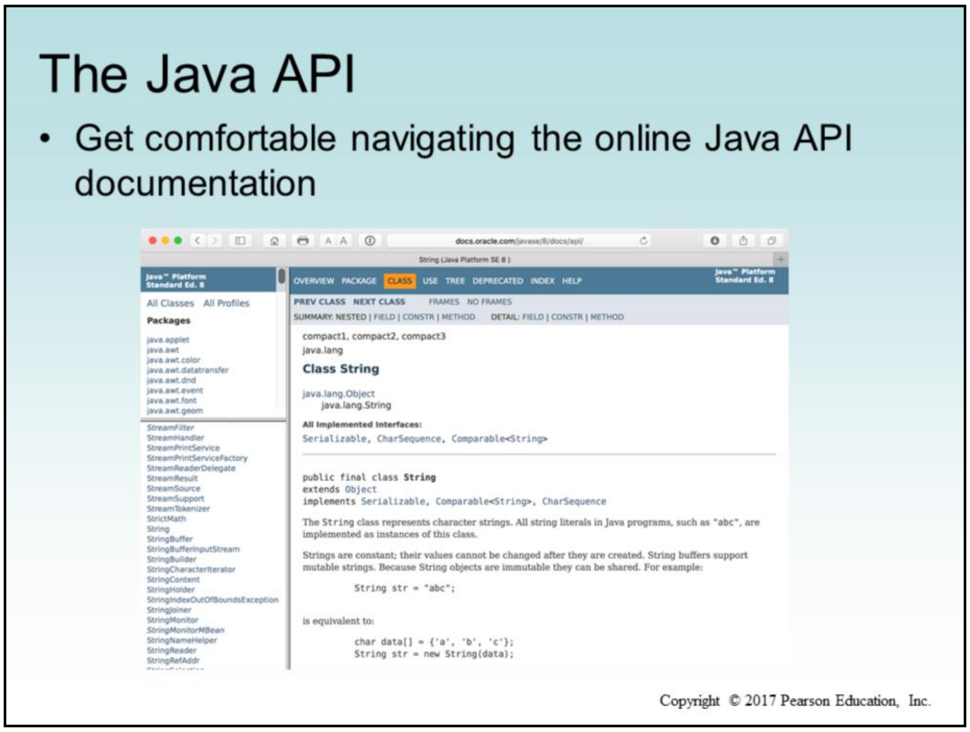
- All classes of the `java.lang` package are imported automatically into all programs
- It's as if all programs contain the following line:

```
import java.lang.*;
```

- That's why we didn't have to import the `System` or `String` classes explicitly in earlier programs
- The `Scanner` class, on the other hand, is part of the `java.util` package, and therefore must be imported

-If only one class from a package is being used, better to specify the single class

-If more than one class, designate the package with the * wildcard



The Java API

- Get comfortable navigating the online Java API documentation

docs.oracle.com/javase/8/docs/api/
String (Java Platform SE 8)
Java™ Platform Standard Ed. 8
OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP
PREV CLASS NEXT CLASS FRAMES NO FRAMES
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD
All Classes All Profiles

Packages

java.applet
java.awt
java.awt.color
java.awt.datatransfer
java.awt.dnd
java.awt.event
java.awt.font
java.awt.geom

StreamFilter
StreamHandler
StreamPrintService
StreamPrintServiceFactory
StreamReaderDelegate
StreamResult
StreamSource
StreamSupport
StreamTokenizer
StrictMath
String
StringBuffer
StringBufferInputStream
StringBuilder
StringCharacterIterator
StringContent
StringHolder
StringIndexOutOfBoundsException
StringJoiner
StringMonitor
StringMonitorMBean
StringNameHelper
StringReader
StringRefAddr

compact1, compact2, compact3
java.lang

Class String

java.lang.Object
java.lang.String

All Implemented Interfaces:
Serializable, CharSequence, Comparable<String>

```
public final class String
extends Object
implements Serializable, Comparable<String>, CharSequence
```

The String class represents character strings. All string literals in Java programs, such as "abc", are implemented as instances of this class.

Strings are constant; their values cannot be changed after they are created. String buffers support mutable strings. Because String objects are immutable they can be shared. For example:

```
        String str = "abc";
```

is equivalent to:

```
        char data[] = {'a', 'b', 'c'};
        String str = new String(data);
```

-Here's the link to the Java version 6 API listing all the classes by packages:

http://docs.oracle.com/javase/6/docs/api/index.html

-Try to find the documentation on the String class and the Scanner class