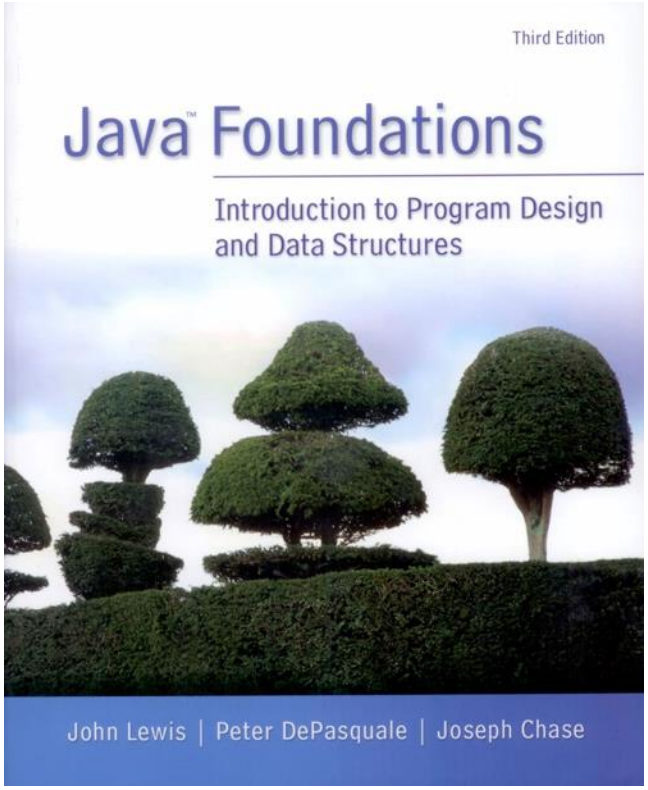


Chapter 3

Using Classes and Objects

Objectives

- How to utilize existing Java classes from different packages to accomplish various tasks
- Specifically:
  - How to deal with strings using the `String` class
  - How to generate pseudo-random numbers using the `Random` class
  - How to make complex mathematical expressions using the `Math` class
  - How to format output using `NumberFormat` and `DecimalFormat` classes
  - Use wrapper classes

Creating and Using Objects

Creating Objects

A variable holds either a primitive type 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

Creating Objects

Generally, we use the `new` operator to create an object

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

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

Creating an object is called ***instantiation***

An object is an ***instance*** of a particular class

Invoking Methods

Once an object has been instantiated, we can use the ***dot operator*** to invoke its methods

```
count = title.length()
```

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

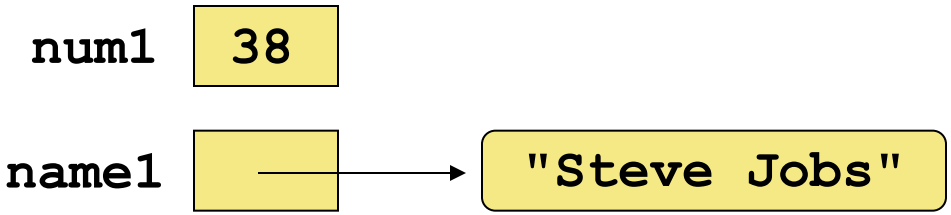
A method invocation can be thought of as asking an object to perform a service

References

A primitive variable contains the value itself

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



Creating Objects - Memory Viewpoint

```
int a = 5;
```

Memory Location	Content	Variable Name
0000		
0001		
0002		
0003		
0004		
0005		
…		
65535		

Creating Objects - Memory Viewpoint

```
int a = 5;
```

Memory Location	Content	Variable Name
0000	5	**a**
0001		
0002		
0003		
0004		
0005		
…		
65535		

Creating Objects - Memory Viewpoint

```
int a = 5;

double x = 1.7;
```

Memory Location	Content	Variable Name
0000	5	**a**
0001		
0002		
0003		
0004		
0005		
…		
65535		

Creating Objects - Memory Viewpoint

```
int a = 5;

double x = 1.7;
```

Memory Location	Content	Variable Name
0000	5	**a**
0001	1.7	**x**
0002		
0003		
0004		
0005		
…		
65535		

Creating Objects - Memory Viewpoint

```
int a = 5;

double x = 1.7;

String name = "hi";
```

Memory Location	Content	Variable Name
0000	5	**a**
0001	1.7	**x**
0002		
0003		
0004		
0005		
…		
65535		

Creating Objects - Memory Viewpoint

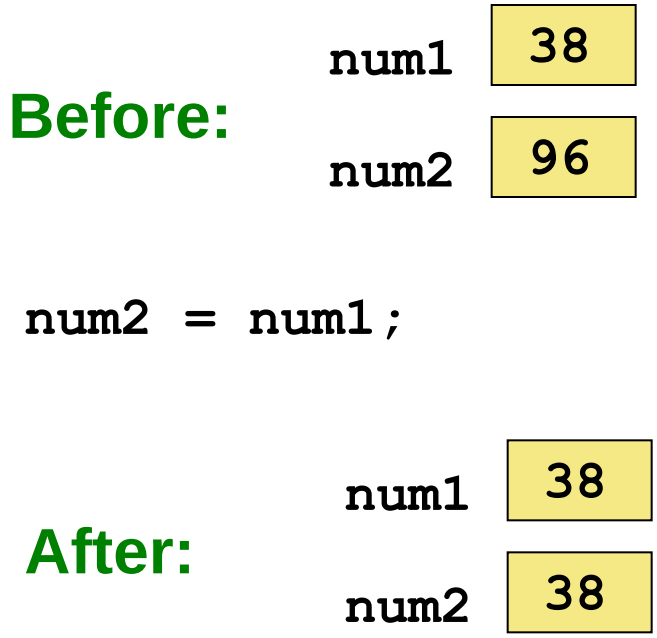
```
int a = 5;

double x = 1.7;

String name = "hi";
```

Memory Location	Content	Variable Name
0000	5	**a**
0001	1.7	**x**
0002	0004	**name**
0003		
0004	‘h’	
0005	‘i’	
…		
65535		

Assignment Revisited

Assignment takes a copy of a value and stores it in a variable

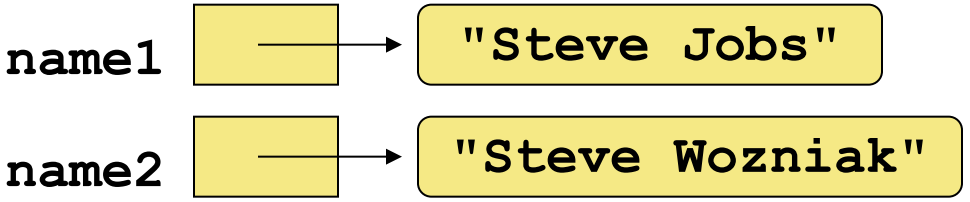
For primitive types



Reference Assignment

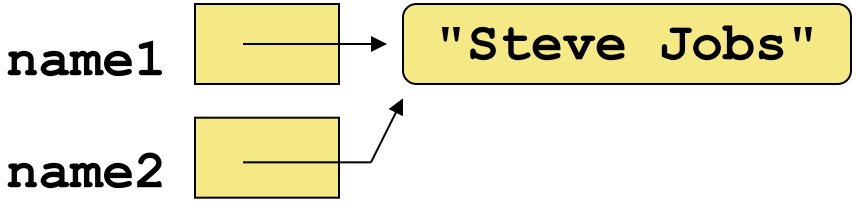
For object references, assignment copies the address

Before:




```
name2 = name1;
```

After:



Aliases

Two or more references that refer to the same object are called ***aliases*** of each other

> Consequence: one object can be accessed using multiple reference variables

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

The String Class

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 only 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

- Thus 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

`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

`String` Methods

Examples of String methods:

- `length()`
- `charAt(int index)`
  - e.g.: charAt(3)
- `toUpperCase()`
- `replace(char oldChar, char newChar)`
  - e.g.: replace('E', 'X')
- `substring(3, 30)`

`EXAMPLE: StringMutation.java`

Using Other 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

`System`, `Scanner`, `String` are part of the Java standard class library

Other class libraries can be obtained through third party vendors, or you can create them yourself

Packages

The classes of the Java standard class library are organized into ***packages***

Some of the packages in the standard class library are

Package	Purpose
java.lang	General support
java.applet	Creating applets for the web
java.awt	Graphics and graphical user interfaces
javax.swing	Additional graphics capabilities
java.net	Network communication
java.util	Utilities
javax.xml.parsers	XML document processing

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.*;
```

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

The Random Class and Pseudorandom Number Generation

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

The `Random` Class

The `Random` class methods:

- nextInt()
  - Returns a random integer
- nextInt(int n)
  - Returns a random integer in the range [0, n)
- nextFloat(), nextDouble()
  - Returns a random float or double value in the range [0.0,1.0)

Examples

First we need to create the Random object:

```
Random gen = new Random();
```

Then we can create pseudorandom numbers:

Integer in range of [0, 7)

```
int randnum = gen.nextInt(7);
```

Integer in range of [-3, 4)

```
int randnum = -3 + gen.nextInt(7);
```

Double in range of [-2.5, 7.5)

```
double randnum =
-2.5 + 10 * gen.nextDouble();
```

Pseudorandom Number Generation

In general:

To create a random `int` in range of [a, b)

```
a + gen.nextInt(b-a)
```

To create a random `double` in range of [a, b)

```
a + (b-a) * gen.nextDouble()
```

`EXAMPLE: RandomNumbers.java`

The Math Class

The `Math` Class

The `Math` class is part of the `java.lang` package

The `Math` class contains methods that perform various mathematical functions

These include

- absolute value
- square root
- exponentiation
- trigonometric functions

`EXAMPLE: Quadratic.java`

The `Math` Class

The methods of the `Math` class are ***static methods*** (also called ***class methods***)

Static methods can be invoked through the class name – **no object of the `Math` class is needed**

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

Formatting Output

Formatting Output

It is often necessary to format values in certain ways so that they can be presented properly

The Java API contains classes that provide formatting capabilities

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

Formatting Output

The `NumberFormat` class has static methods that return a formatter object

`getCurrencyInstance()`

`getPercentInstance()`

Each formatter object has a method called `format` that returns a string with the specified information in the appropriate format

Formatting Output

Some methods of the `NumberFormat` class:

`String format (double number)`
Returns a string containing the specified `number` formatted according to this object's pattern.

`static NumberFormat getCurrencyInstance()`
Returns a NumberFormat object that represents a currency format for the current locale.

`static NumberFormat getPercentInstance()`
Returns a NumberFormat object that represents a percentage format for the current locale.

EXAMPLE: Purchase.java

Formatting Output

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

Formatting Output

Some methods of the `DecimalFormat` class:

`DecimalFormat (String pattern)`
Constructor: creates a new DecimalFormat object with the specified `pattern`.

`void applyPattern (String pattern)`
Applies the specified `pattern` to this `DecimalFormat` object.

`String format (double number)`
Returns a string containing the specified `number` formatted according to the current pattern.

EXAMPLE: CircleStats.java

Wrapper Classes for Primitive Data Types

Wrapper Classes

The `java.lang` package contains ***wrapper classes*** that correspond to each primitive type

Primitive Type	Wrapper Class
`byte`	`Byte`
`short`	`Short`
`int`	`Integer`
`long`	`Long`
`float`	`Float`
`double`	`Double`
`char`	`Character`
`boolean`	`Boolean`
`void`	`Void`

Wrapper Classes

The following declaration creates an `Integer` object which represents the integer 40 as an object:

```
Integer age = new Integer(40);
```

An object of a wrapper class can be used in any situation where a primitive value will not suffice

- e.g., primitive values could not be stored in containers, but wrapper objects could be

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:

```
num = Integer.parseInt(str);
```

The wrapper classes often contain useful constants as well

- For example, the `Integer` class contains `MIN_VALUE` and `MAX_VALUE` which hold the smallest and largest `int` values

Autoboxing

Autoboxing is the automatic conversion of a primitive value to a corresponding wrapper object

```
Integer obj;
int num = 42;
obj = num;
```

The assignment creates the appropriate `Integer` object

The reverse conversion (called ***unboxing***) also occurs automatically as needed

Summary

- Classes provide methods that extend the functionality of the program
- Objects of a class are created using the `new` operator
- Object variables contain the address of the object
- The `String` class provides methods to manipulate string objects
- The `Random` class provides methods to generate pseudorandom numbers
- The `Math` class provides methods for performing various mathematical functions
- The `NumberFormat` and `DecimalFormat` classes provide methods for formatting output
- Wrapper classes allow primitive data types to be treated as objects