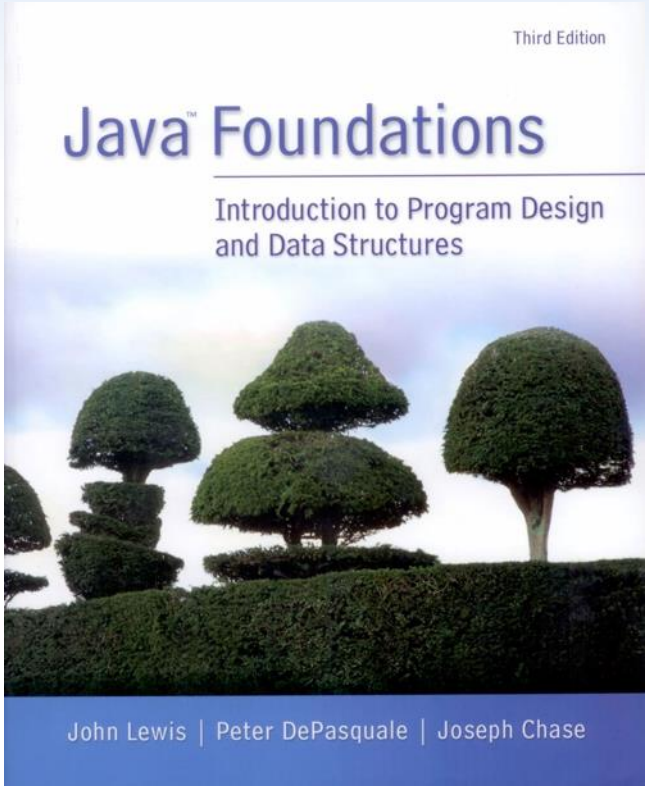


Chapter 3

Using Classes and Objects

Chapter Scope

- Creating objects
- Services of the `String` class
- The Java API class library
- The `Random` and `Math` classes
- Formatting output
- Introducing enumerated types
- Wrapper classes

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("James Gosling");
```

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

Creating Strings

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

```
title = "Java rocks!";
```

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

Invoking Methods

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

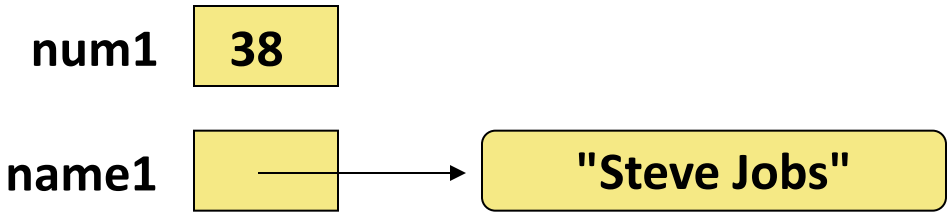
```
count = title.length()
```

- 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

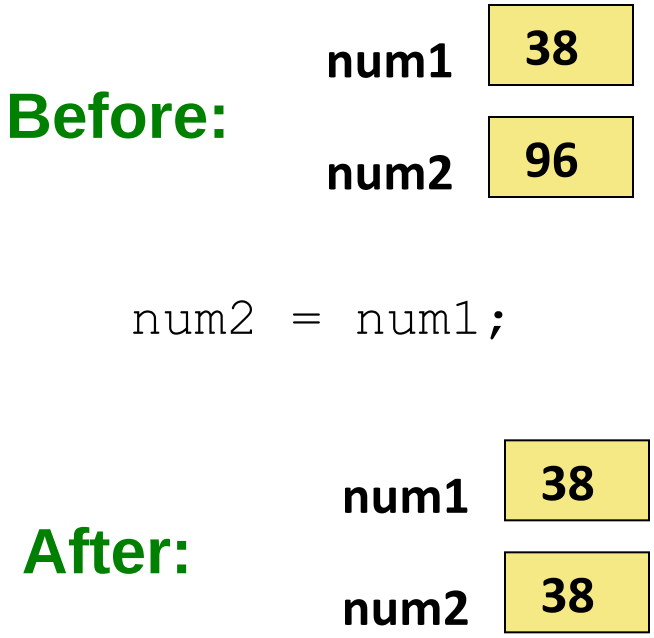
Object References

- 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
- Rather than dealing with arbitrary addresses, we often depict a reference graphically



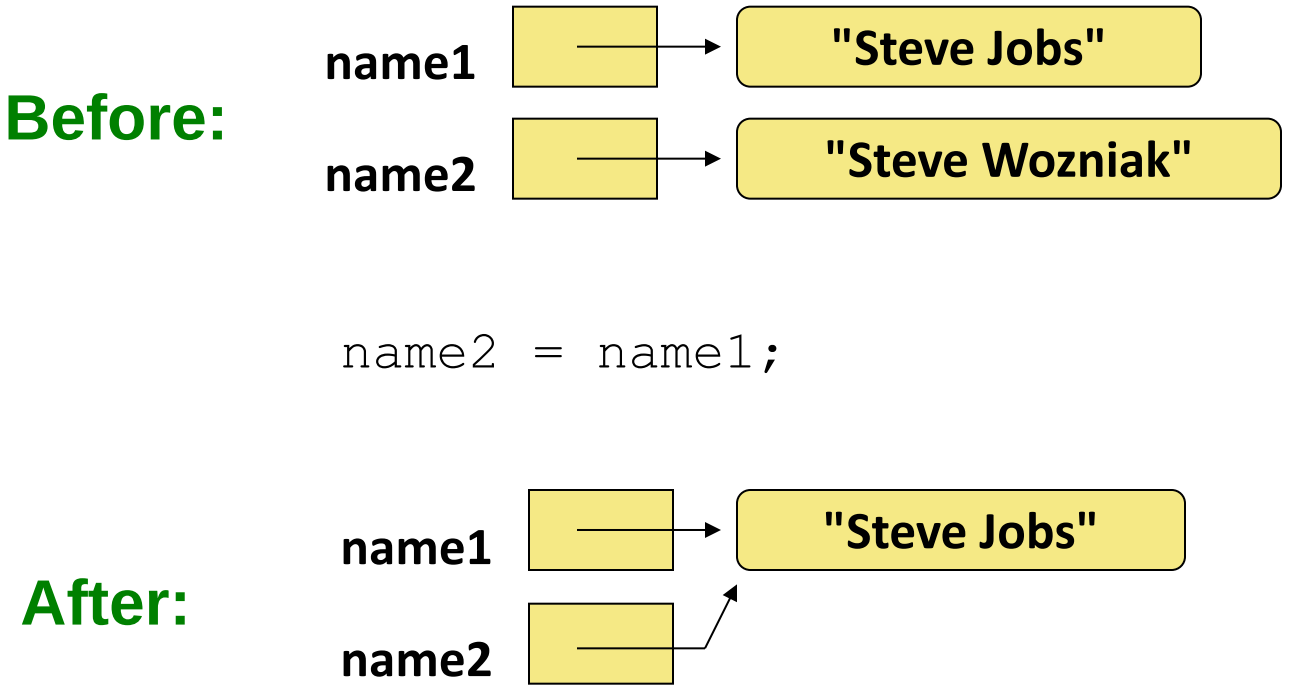
Assignment Revisited

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



Assignment Revisited

- For object references, the address is copied:



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 explicitly

The String Class

- 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

The String Class

- 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

- Some methods of the `String` class:

`String (String str)`
Constructor: creates a new string object with the same characters as `str`.

`char charAt (int index)`
Returns the character at the specified `index`.

`int compareTo (String str)`
Returns an integer indicating if this string is lexically before (a negative return value), equal to (a zero return value), or lexically after (a positive return value), the string `str`.

`String concat (String str)`
Returns a new string consisting of this string concatenated with `str`.

`boolean equals (String str)`
Returns true if this string contains the same characters as `str` (including case) and false otherwise.

`boolean equalsIgnoreCase (String str)`
Returns true if this string contains the same characters as `str` (without regard to case) and false otherwise.

`int length ()`
Returns the number of characters in this string.

`String replace (char oldChar, char newChar)`
Returns a new string that is identical with this string except that every occurrence of `oldChar` is replaced by `newChar`.

`String substring (int offset, int endIndex)`
Returns a new string that is a subset of this string starting at index `offset` and extending through `endIndex-1`.

`String toLowerCase ()`
Returns a new string identical to this string except all uppercase letters are converted to their lowercase equivalent.

`String toUpperCase ()`
Returns a new string identical to this string except all lowercase letters are converted to their uppercase equivalent.

```
//********************************************************************
//  StringMutation.java       Java Foundations
//
//  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);
```

```
        // 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());
    }
}
```

The Java API

- A *class library* is a collection of classes that we can use when developing programs
- The *Java API* is the standard class library that is part of any Java development environment
- API stands for Application Programming Interface
- Various classes we've already used (`System`, `Scanner`, `String`) are part of the Java API
- Other class libraries can be obtained through third party vendors, or you can create them yourself

Packages

- The classes of the Java API are organized into *packages*

Package	Provides support to
`java.applet`	Create programs (applets) that are easily transported across the Web.
`java.awt`	Draw graphics and create graphical user interfaces; AWT stands for Abstract Windowing Toolkit.
`java.beans`	Define software components that can be easily combined into applications.
`java.io`	Perform a wide variety of input and output functions.
`java.lang`	General support; it is automatically imported into all Java programs.
`java.math`	Perform calculations with arbitrarily high precision.
`java.net`	Communicate across a network.
`java.rmi`	Create programs that can be distributed across multiple computers; RMI stands for Remote Method Invocation.
`java.security`	Enforce security restrictions.
`java.sql`	Interact with databases; SQL stands for Structured Query Language.
`java.text`	Format text for output.
`java.util`	General utilities.
`javax.swing`	Create graphical user interfaces with components that extend the AWT capabilities.
`javax.xml.parsers`	Process XML documents; XML stands for eXtensible Markup Language.

Import Declarations

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

```
java.util.Scanner
```

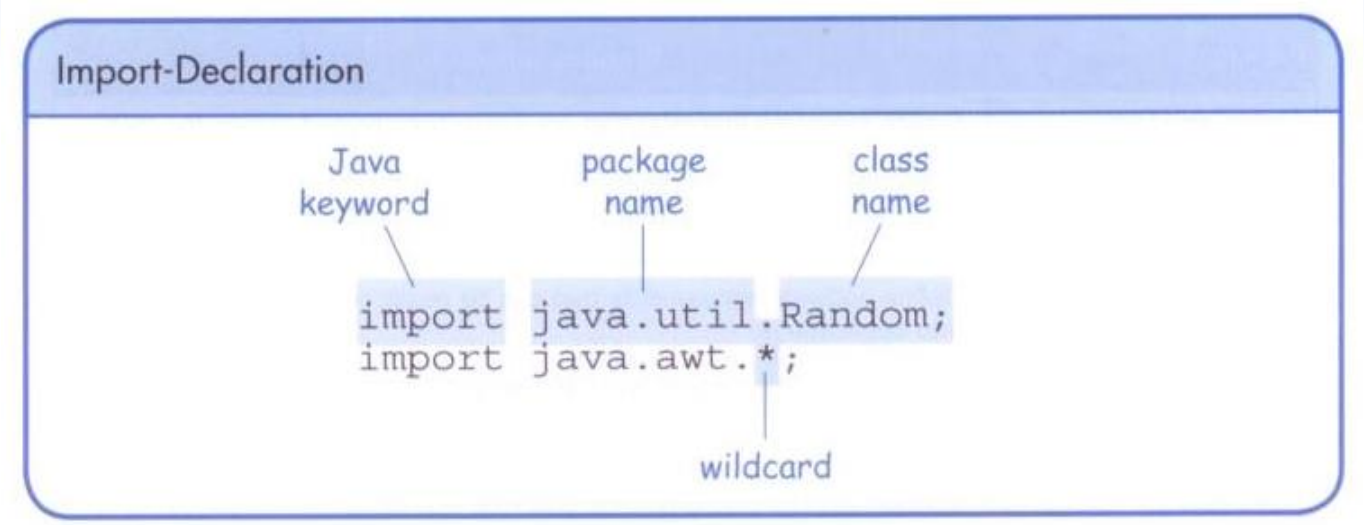
- Or you can *import* the class, and then use just the class name:

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

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

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

Import Declarations



The java.lang Package

- 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

- 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

- Some methods of the `Random` class:

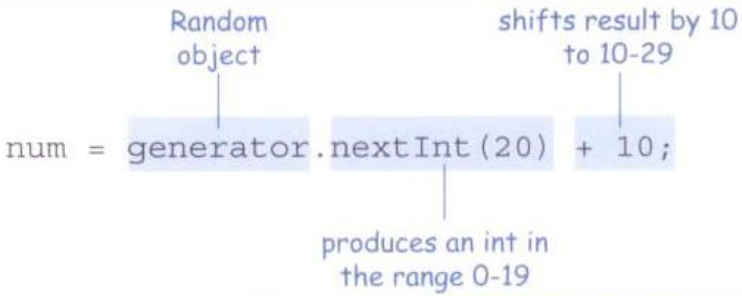
`Random ()`
Constructor: creates a new pseudorandom number generator.

`float nextFloat ()`
Returns a random number between 0.0 (inclusive) and 1.0 (exclusive).

`int nextInt ()`
Returns a random number that ranges over all possible `int` values (positive and negative).

`int nextInt (int num)`
Returns a random number in the range 0 to `num-1`.

Generating-Random-Numbers



```
//********************************************************************
//  RandomNumbers.java       Java Foundations
//
//  Demonstrates the creation of pseudo-random numbers using the
//  Random class.
//********************************************************************

import java.util.Random;

public class RandomNumbers
{
   //-----------------------------------------------------------------
   //  Generates random numbers in various ranges.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      Random generator = new Random();
      int num1;
      float num2;

      num1 = generator.nextInt();
      System.out.println("A random integer: " + num1);

      num1 = generator.nextInt(10);
      System.out.println("From 0 to 9: " + num1);
```

```
        num1 = generator.nextInt(10) + 1;
        System.out.println("From 1 to 10: " + num1);

        num1 = generator.nextInt(15) + 20;
        System.out.println("From 20 to 34: " + num1);

        num1 = generator.nextInt(20) - 10;
        System.out.println("From -10 to 9: " + num1);

        num2 = generator.nextFloat();
        System.out.println("A random float (between 0-1): " + num2);

        num2 = generator.nextFloat() * 6;  // 0.0 to 5.999999
        num1 = (int)num2 + 1;
        System.out.println("From 1 to 6: " + num1);
    }
}
```

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

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);
```

- We'll discuss static methods in more detail later

- Some methods of the `Math` class:

`static int abs (int num)`
Returns the absolute value of `num`.

`static double acos (double num)`

`static double asin (double num)`

`static double atan (double num)`
Returns the arc cosine, arc sine, or arc tangent of `num`.

`static double cos (double angle)`

`static double sin (double angle)`

`static double tan (double angle)`
Returns the angle cosine, sine, or tangent of `angle`, which is measured in radians.

`static double ceil (double num)`
Returns the ceiling of `num`, which is the smallest whole number greater than or equal to num.

`static double exp (double power)`
Returns the value e raised to the specified `power`.

`static double floor (double num)`
Returns the floor of num, which is the largest whole number less than or equal to `num`.

`static double pow (double num, double power)`
Returns the value `num` raised to the specified `power`.

`static double random ()`
Returns a random number between 0.0 (inclusive) and 1.0 (exclusive).

`static double sqrt (double num)`
Returns the square root of `num`, which must be positive.

```
//********************************************************************
//  Quadratic.java       Java Foundations
//
//  Demonstrates the use of the Math class to perform a calculation
//  based on user input.
//********************************************************************

import java.util.Scanner;

public class Quadratic
{
   //-----------------------------------------------------------------
   //  Determines the roots of a quadratic equation.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      int a, b, c;  // ax^2 + bx + c
      double discriminant, root1, root2;

      Scanner scan = new Scanner(System.in);

      System.out.print("Enter the coefficient of x squared: ");
      a = scan.nextInt();

      System.out.print("Enter the coefficient of x: ");
      b = scan.nextInt();
```

```
      System.out.print("Enter the constant: ");
      c = scan.nextInt();

      // Use the quadratic formula to compute the roots.
      // Assumes a positive discriminant.

      discriminant = Math.pow(b, 2) - (4 * a * c);
      root1 = ((-1 * b) + Math.sqrt(discriminant)) / (2 * a);
      root2 = ((-1 * b) - Math.sqrt(discriminant)) / (2 * a);

      System.out.println("Root #1: " + root1);
      System.out.println("Root #2: " + root2);
   }
}
```

Formatting Ouput

- 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.

```
//********************************************************************
//  Purchase.java       Java Foundations
//
//  Demonstrates the use of the NumberFormat class to format output.
//********************************************************************

import java.util.Scanner;
import java.text.NumberFormat;

public class Purchase
{
   //-----------------------------------------------------------------
   //  Calculates the final price of a purchased item using values
   //  entered by the user.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      final double TAX_RATE = 0.06;  // 6% sales tax

      int quantity;
      double subtotal, tax, totalCost, unitPrice;

      Scanner scan = new Scanner(System.in);

      NumberFormat fmt1 = NumberFormat.getCurrencyInstance();
      NumberFormat fmt2 = NumberFormat.getPercentInstance();
```

```
        System.out.print("Enter the quantity: ");
        quantity = scan.nextInt();

        System.out.print("Enter the unit price: ");
        unitPrice = scan.nextDouble();

        subtotal = quantity * unitPrice;
        tax = subtotal * TAX_RATE;
        totalCost = subtotal + tax;

        // Print output with appropriate formatting
        System.out.println("Subtotal: " + fmt1.format(subtotal));
        System.out.println("Tax: " + fmt1.format(tax) + " at "
                           + fmt2.format(TAX_RATE));
        System.out.println("Total: " + fmt1.format(totalCost));
    }
}
```

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.

```
//********************************************************************
//  CircleStats.java       Java Foundations
//
//  Demonstrates the formatting of decimal values using the
//  DecimalFormat class.
//********************************************************************

import java.util.Scanner;
import java.text.DecimalFormat;

public class CircleStats
{
   //-----------------------------------------------------------------
   //  Calculates the area and circumference of a circle given its
   //  radius.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      int radius;
      double area, circumference;

      Scanner scan = new Scanner(System.in);

      System.out.print("Enter the circle's radius: ");
      radius = scan.nextInt();
```

```
      area = Math.PI * Math.pow(radius, 2);
      circumference = 2 * Math.PI * radius;

      // Round the output to three decimal places
      DecimalFormat fmt = new DecimalFormat("0.###");

      System.out.println("The circle's area: " + fmt.format(area));
      System.out.println("The circle's circumference: "
                         + fmt.format(circumference));
   }
}
```

Enumerated Types

- Java allows you to define an enumerated type, which can then be used to declare variables
- An enumerated type establishes all possible values for a variable of that type
- The values are identifiers of your own choosing
- The following declaration creates an enumerated type called `Season`

```
enum Season {winter, spring, summer, fall};
```

- Any number of values can be listed

Enumerated Types

- Once a type is defined, a variable of that type can be declared

```
Season time;
```

and it can be assigned a value

```
time = Season.fall;
```

- The values are specified through the name of the type
- Enumerated types are *type-safe* – you cannot assign any value other than those listed

Enumerated Types

- Internally, each value of an enumerated type is stored as an integer, called its *ordinal value*
- The first value in an enumerated type has an ordinal value of zero, the second one, and so on
- However, you cannot assign a numeric value to an enumerated type, even if it corresponds to a valid ordinal value

Enumerated Types

- The declaration of an enumerated type is a special type of class, and each variable of that type is an object
- The `ordinal` method returns the ordinal value of the object
- The `name` method returns the name of the identifier corresponding to the object's value

```
//********************************************************************
//  IceCream.java       Java Foundations
//
//  Demonstrates the use of enumerated types.
//********************************************************************

public class IceCream
{
   enum Flavor {vanilla, chocolate, strawberry, fudgeRipple, coffee,
                rockyRoad, mintChocolateChip, cookieDough}

   //-----------------------------------------------------------------
   //  Creates and uses variables of the Flavor type.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      Flavor cone1, cone2, cone3;

      cone1 = Flavor.rockyRoad;
      cone2 = Flavor.chocolate;

      System.out.println("cone1 value: " + cone1);
      System.out.println("cone1 ordinal: " + cone1.ordinal());
      System.out.println("cone1 name: " + cone1.name());
```

```
        System.out.println();
        System.out.println("cone2 value: " + cone2);
        System.out.println("cone2 ordinal: " + cone2.ordinal());
        System.out.println("cone2 name: " + cone2.name());

        cone3 = cone1;

        System.out.println();
        System.out.println("cone3 value: " + cone3);
        System.out.println("cone3 ordinal: " + cone3.ordinal());
        System.out.println("cone3 name: " + cone3.name());
    }
}
```

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:

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

- An object of a wrapper class can be used in any situation where a primitive value will not suffice
- For example, some objects serve as collections of other objects
- Primitive values could not be stored in such collections, 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

Wrapper Classes

- Some methods of the `Integer` class:

`Integer (int value)`
Constructor: creates a new Integer object storing the specified value.

`byte byteValue ()`
`double doubleValue ()`
`float floatValue ()`
`int intValue ()`
`long longValue ()`
Return the value of this Integer as the corresponding primitive type.

`static int parseInt (String str)`
Returns the int corresponding to the value stored in the specified string.

`static String toBinaryString (int num)`
`static String tohexString (int num)`
`static String toOctalString (int num)`
Returns a string representation of the specified integer value in the corresponding base.

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