Enumerations, Autoboxing and Static Import

Enumerations

- Enumerations was added since JDK 5.
- An enumeration is a list of named constants.
- Java enumeration defines a class type.
- Java enumeration can have constructors, methods and instance variables.
- An enumeration is created using the enum keyword.

```
//An enumeration of apple varieties.
enum Apple {
	Jonathan, GoldenDel, RedDel, Winesap, Cortland
}
```

The identifiers Jonathan, GoldenDel, and so on, are called enumeration constants. Each is implicitly declared as a public, static final member of Apple. Furthermore, their type is the type of the enumeration in which they are declared, which is Apple in this case.

Below code shows an example of enum usage.

```
class EnumDemo {
	public static void main(String args[]) {
		Apple ap;
		ap = Apple.RedDel;
		// Output an enum value.
		System.out.println("Value of ap: " + ap);
		System.out.println();
		ap = Apple.GoldenDel;
		// Compare two enum values.
		if (ap == Apple.GoldenDel)
			System.out.println("ap contains GoldenDel.\n");
		// Use an enum to control a switch statement.
		switch (ap) {
		case Jonathan:
			System.out.println("Jonathan is red.");
			break;
		case GoldenDel:
			System.out.println("Golden Delicious is yellow.");
			break;
		case RedDel:
			System.out.println("Red Delicious is red.");
			break;
		case Winesap:
			System.out.println("Winesap is red.");
			break;
		case Cortland:
			System.out.println("Cortland is red.");
			break;
		}
	}
}
```

The values() and valueOf() Methods

All enumerations automatically contain two predefined methods: values( ) and valueOf( ).

public static *enum-type* [ ] values( )
public static *enum-type* valueOf(String *str*)

- The values( ) method returns an array that contains a list of the enumeration constants.
- The valueOf( ) method returns the enumeration constant whose value corresponds to the string passed in str.
- In both cases, enum-type is the type of the enumeration.

```
class EnumDemo2 {
	public static void main(String args[]) {
		Apple ap;
		System.out.println("Here are all Apple constants:");
		// use values()
		Apple allapples[] = Apple.values();
		for (Apple a : allapples)
			System.out.println(a);
		System.out.println();
		// use valueOf()
		ap = Apple.valueOf("Winesap");
		System.out.println("ap contains " + ap);
	}
}
```

The output from the program is shown here:

```
Here are all Apple constants:
Jonathan
GoldenDel
RedDel
Winesap
Cortland

ap contains Winesap
```

Java Enumerations Are Class Types

- Java enumeration is a class type.
- Java enum can have constructors, add instance variables and methods, and even implement interfaces.
- However, no object is created as an instance of an enum.
- Instead, the constructor of an enum is called when each enumeration constant is created.

```
//Use an enum constructor, instance variable, and method.
enum Apple {
	Jonathan(10), GoldenDel(9), RedDel(12), Winesap(15), Cortland(8);

	private int price; // price of each apple

	// Constructor
	Apple(int p) {
		price = p;
	}

	int getPrice() {
		return price;
	}
}

public class Test {
	public static void main(String args[]) {
		Apple ap;
		// Display price of Winesap.
		System.out.println("Winesap costs " + Apple.Winesap.getPrice() + "
cents.\n");
		// Display all apples and prices.
		System.out.println("All apple prices:");
		for (Apple a : Apple.values())
			System.out.println(a + " costs " + a.getPrice() + " cents.");
	}
}
```

The output is shown here:

```
Winesap costs 15 cents.
All apple prices:
Jonathan costs 10 cents.
GoldenDel costs 9 cents.
RedDel costs 12 cents.
Winesap costs 15 cents.
Cortland costs 8 cents.
```

Enumerations Inherit Enum

- All enumerations automatically inherit one: java.lang.Enum
- This class defines several methods that are available for use by all enumerations.
- Some methods that might be useful:
    - ordinal( ) : returns a value that indicates an enumeration constant's position in the list of constants
    - compareTo( ) : compare the ordinal value of two constants of the same enumeration
    - equals( ) : compare for equality an enumeration constant with any other object

```
//Demonstrate ordinal(), compareTo(), and equals().
//An enumeration of apple varieties.
enum Apple {
	Jonathan, GoldenDel, RedDel, Winesap, Cortland
}

public class Test {
	public static void main(String args[]) {
		Apple ap, ap2, ap3;
		// Obtain all ordinal values using ordinal().
		System.out.println("Here are all apple constants" + " and their
ordinal values: ");
		for (Apple a : Apple.values())
			System.out.println(a + " " + a.ordinal());
		ap = Apple.RedDel;
		ap2 = Apple.GoldenDel;
		ap3 = Apple.RedDel;
		System.out.println();
		// Demonstrate compareTo() and equals()
		if (ap.compareTo(ap2) < 0)
			System.out.println(ap + " comes before " + ap2);
		if (ap.compareTo(ap2) > 0)
			System.out.println(ap2 + " comes before " + ap);
		if (ap.compareTo(ap3) == 0)
			System.out.println(ap + " equals " + ap3);
		System.out.println();
		if (ap.equals(ap2))
			System.out.println("Error!");
		if (ap.equals(ap3))
			System.out.println(ap + " equals " + ap3);
		if (ap == ap3)
			System.out.println(ap + " == " + ap3);
	}
}
```

Type Wrappers

- The type wrappers are Double, Float, Long, Integer, Short, Byte, Character, and Boolean.
- These classes offer a wide array of methods that allow you to fully integrate the primitive types into Java's object hierarchy.

Character

Character is a wrapper around a char. The constructor for Character is

```
Character(char ch)
```

Here, ch specifies the character that will be wrapped by the Character object being created.

To obtain the char value contained in a Character object, call charValue( ), shown here:

```
char charValue( )
```

It returns the encapsulated character.

Boolean

Boolean is a wrapper around boolean values. It defines these constructors:

```
Boolean(boolean boolValue)
Boolean(String boolString)
```

In the first version, boolValue must be either true or false. In the second version, if boolString contains the string "true" (in uppercase or lowercase), then the new Boolean object will be true. Otherwise, it will be false.

To obtain a boolean value from a Boolean object, use booleanValue( ), shown here:

```
boolean booleanValue( )
```

It returns the boolean equivalent of the invoking object.

Numeric Type Wrappers

These are Byte, Short, Integer, Long, Float, and Double. All of the numeric type wrappers inherit the abstract class Number. Number declares methods that return the value of an object in each of the different number formats. These methods are shown here:

```
byte byteValue( )
double doubleValue( )
float floatValue( )
int intValue( )
long longValue( )
short shortValue( )
```

Autoboxing

Beginning with JDK 5, Java added two important features: autoboxing and auto-unboxing.

Autoboxing Fundamentals

Autoboxing is the process by which a primitive type is automatically encapsulated (boxed) into its equivalent type wrapper whenever an object of that type is needed. There is no need to explicitly construct an object.

Auto-unboxing is the process by which the value of a boxed object is automatically extracted (unboxed) from a type wrapper when its value is needed. There is no need to call a method such as intValue( ) or doubleValue( ).

```
Integer iOb = 100; // autobox an int
int i = iOb; // auto-unbox
```

Autoboxing and Methods

In addition to the simple case of assignments, autoboxing automatically occurs whenever a primitive type must be converted into an object; auto-unboxing takes place whenever an object must be converted into a primitive type. Thus, autoboxing/unboxing might occur when an argument is passed to a method, or when a value is returned by a method.

```
//Autoboxing/unboxing takes place with
//method parameters and return values.
public class Test {

	// Take an Integer parameter and return
	// an int value;
	static int m(Integer v) {
		return v; // auto-unbox to int
	}

	public static void main(String args[]) {
		// Pass an int to m() and assign the return value
		// to an Integer. Here, the argument 100 is autoboxed
		// into an Integer. The return value is also autoboxed
		// into an Integer.
		Integer iOb = m(100);
		System.out.println(iOb);
	}
}

//This program displays the following result:
//100
```

Autoboxing/Unboxing Occurs in Expressions

In general, autoboxing and unboxing take place whenever a conversion into an object or from an object is required. This applies to expressions. Within an expression, a numeric object is automatically unboxed. The outcome of the expression is reboxed, if necessary.

```
//Autoboxing/unboxing occurs inside expressions.
public class Test {

	public static void main(String args[]) {
		Integer iOb, iOb2;
		int i;
		iOb = 100;
		System.out.println("Original value of iOb: " + iOb);
		// The following automatically unboxes iOb,
		// performs the increment, and then reboxes
		// the result back into iOb.
		++iOb;
		System.out.println("After ++iOb: " + iOb);
		// Here, iOb is unboxed, the expression is
		// evaluated, and the result is reboxed and
		// assigned to iOb2.
		iOb2 = iOb + (iOb / 3);
		System.out.println("iOb2 after expression: " + iOb2);
		// The same expression is evaluated, but the
		// result is not reboxed.
		i = iOb + (iOb / 3);
		System.out.println("i after expression: " + i);
	}
}

/*
 * The output is shown here:
	Original value of iOb: 100
	After ++iOb: 101
	iOb2 after expression: 134
	i after expression: 134
 */
```

Static Import

If you have to access the static members of a class, then it is necessary to qualify the references with the class from which they come.

Java 5 provides the static import feature that enables unqualified access to static members without having to qualify them with the class name.

```
package com.example.test;

import com.example.domain.PlayingCard;
import static com.example.domain.Suit.*;

public class TestPlayingCardStatic {

	public static void main(String[] args) {
		PlayingCard card = new PlayingCard(CLUBS, 2); // no need to prefix
the class
		System.out.println("card is the " + card.getRank() + " of the " +
card.getSuitName());
		System.out.println("card is the " + card.getRank() + " of the " +
card.getSuit().getName());
	}

}
```

Use static imports sparingly. If you overuse the static import feature, it can make your program unreadable and unmaintainable, polluting its namespace with all of the static members that you import. Readers of your code (including you, a few months after you wrote it) will not know from which class a static member comes. Importing all of the static members from a class can be very harmful to readability; if you need one or two members only, import them individually. Used appropriately, static import can make your program more readable, by removing the boilerplate of repetition of class names.