Enumerations:

An enumeration is a list of named constants.

// An enumeration of apple varieties.

enum Apple { Jonathan, GoldenDel, RedDel, Winesap, Cortland }

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 output from the program is shown here:

Value of ap: RedDel ap contains GoldenDel. Golden Delicious is yellow.

All enumerations automatically contain two predefined methods: values( ) and valueOf( ). Their general forms are shown here: 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

*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*

As explained, a Java enumeration is a class type.

you can give them constructors, add instance variables and methods, and even implement interfaces

*// 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; } } class EnumDemo3 { public static void main(String args[]) { Apple ap; 260 Part I: The Java Language // 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:

* Although you can’t inherit a superclass when declaring an enum, all enumerations automatically inherit one: java.lang.Enum.
* You can obtain a value that indicates an enumeration constant’s position in the list of constants. This is called its ordinal value, and it is retrieved by calling the ordinal( ) method, shown here: final int ordinal( ) It returns the ordinal value of the invoking constant. Ordinal values begin at zero
* Two methods are defined
  + final int compareTo(enum-type e)
  + *enum-type is the type of the enumeration, and e is the constant being compared to the invoking constant. Remember, both the invoking constant and e must be of the same enumeration*
  + equality an enumeration constant with any other object by using equals( ), which overrides the equals( ) method defined by Object
* *enum Apple { Jonathan, GoldenDel, RedDel, Winesap, Cortland } class EnumDemo4 { 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); } } The output from the program is shown here: Here are all apple constants and their ordinal values: Jonathan 0 GoldenDel 1 RedDel 2 Winesap 3 Cortland 4 GoldenDel comes before RedDel RedDel equals RedDel RedDel equals RedDel RedDel == RedDel*

Type Wrappers:

The process of encapsulating a value within an object is called boxing. Thus, in the program, this line boxes the value 100 into an Integer:

Integer iOb = new Integer(100);

The process of extracting a value from a type wrapper is called unboxing. For example, the program unboxes the value in iOb with this statement:

int i = iOb.intValue();

Integer iOb = 100; // autobox an int

Notice that no object is explicitly created through the use of new. Java handles this for you, automatically. To unbox an object, simply assign that object reference to a primitive-type variable. For example, to unbox iOb, you can use this line:

int i = iOb; // auto-unbox

**Annotation:**

* Annotation is a super-interface of all annotations. It is declared within the java.lang.annotation package.
* It overrides hashCode( ),equals( ), and toString( ), which are defined by Object.
* It also specifies annotationType( ),which returns a Class object that represents the invoking annotation
* They are SOURCE, CLASS, and RUNTIME.
* An annotation with a retention policy of SOURCE is retained only in the source file and is discarded during compilation.
* An annotation with a retention policy of CLASS is stored in the .class file during compilation. However, it is not available through the JVM during run time.
* An annotation with a retention policy of RUNTIME is stored in the .class file during compilation and is available through the JVM during run time.
* Thus, RUNTIME retention offers the greatest annotation persistence.

**public** **class** Meta {

@MyAnno(str = "Annotation Example", val = 100)

**public** **static** **void** myMeth() {

Meta ob = **new** Meta();

// Obtain the annotation for this method

// and display the values of the members.

**try** {// First, get a Class object that represents

// this class.

Class<Meta> c=(Class<Meta>) ob.getClass();

// Now, get a Method object that represents

// this method.

Method m = c.getMethod("myMeth");

// Next, get the annotation for this class.

MyAnno anno = m.getAnnotation(MyAnno.**class**);

// Finally, display the values.

System.***out***.println(anno.str() + " " + anno.val());

} **catch** (NoSuchMethodException exc) {

System.***out***.println("Method Not Found.");

}

}

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

*myMeth*();

}

**package** practise;

**import** java.lang.annotation.Retention;

**import** java.lang.annotation.RetentionPolicy;

@Retention(RetentionPolicy.***RUNTIME***)

**public** **@interface** MyAnno {

String str();

**int** val();

}

**Marker Annotations**

A*marker* annotation is a special kind of annotation that contains no members. Its sole purpose

is to mark a declaration. Thus, its presence as an annotation is sufficient. The best way to

determine if a marker annotation is present is to use the method **isAnnotationPresent( )**,

which is a defined by the **AnnotatedElement** interface.

**package** practise;

**import** java.lang.annotation.\*;

**import** java.lang.reflect.\*;

// A marker annotation.

@Retention(RetentionPolicy.***RUNTIME***)

**@interface** MyMarker { }

**class** Marker {

// Annotate a method using a marker.

// Notice that no ( ) is needed.

@MyMarker

**public** **static** **void** myMeth()

{

Marker ob = **new** Marker();

**try**

{

Method m = ob.getClass().getMethod("myMeth");

// Determine if the annotation is present.

**if**(m.isAnnotationPresent(MyMarker.**class**))

System.***out***.println("MyMarker is present.");

}

**catch** (NoSuchMethodException exc) {

System.***out***.println("Method Not Found.");

}

}

**public** **static** **void** main(String args[]) {

*myMeth*();

}

}

**Some Restrictions**

There are a number of restrictions that apply to annotation declarations. First, no annotation

can inherit another. Second, all methods declared by an annotation must be without parameters.

Furthermore, they must return one of the following:

• Aprimitive type, such as **int** or **double**

• An object of type **String** or **Class**

• An **enum** type

• Another annotation type

• An array of one of the preceding types

Annotations cannot be generic. In other words, they cannot take type parameters

**The transient and volatile Modifiers:**

When an instance variable is declared as **transient**, then its value need not persist when

an object is stored. For example:

class T {

transient int a; // will not persist

int b; // will persist

}

Here, if an object of type **T** is written to a persistent storage area, the contents of **a** would

not be saved, but the contents of **b** would.

The **volatile** modifier tells the compiler that the variable modified by **volatile** can be

changed unexpectedly by other parts of your program

**instance of**

Java provides the run-time operator **instanceof** to answer this question.

The **instanceof** operator has this general form:

*objref* instanceof *type*

Here, *objref* is a reference to an instance of a class, and *type* is a class type

**package** practise;

**class** A {

**int** i, j;

}

**class** B {

**int** i, j;

}

**class** C **extends** A {

**int** k;

}

**class** D **extends** A {

**int** k;

}

**public** **class** InstanceOf {

**public** **static** **void** main(String args[]) {

A a = **new** A();

B b = **new** B();

C c = **new** C();

D d = **new** D();

**if**(a **instanceof** A)

System.***out***.println("a is instance of A");

**if**(b **instanceof** B)

System.***out***.println("b is instance of B");

**if**(c **instanceof** C)

System.***out***.println("c is instance of C");

**if**(c **instanceof** A)

System.***out***.println("c can be cast to A");

**if**(a **instanceof** C)

System.***out***.println("a can be cast to C");

System.***out***.println();

// compare types of derived types

A ob;

ob = d; // A reference to d

System.***out***.println("ob now refers to d");

**if**(ob **instanceof** D)

System.***out***.println("ob is instance of D");

System.***out***.println();

ob = c; // A reference to c

System.***out***.println("ob now refers to c");

**if**(ob **instanceof** D)

System.***out***.println("ob can be cast to D");

**else**

System.***out***.println("ob cannot be cast to D");

**if**(ob **instanceof** A)

System.***out***.println("ob can be cast to A");

System.***out***.println();

// all objects can be cast to Object

**if**(a **instanceof** Object)

System.***out***.println("a may be cast to Object");

**if**(b **instanceof** Object)

System.***out***.println("b may be cast to Object");

**if**(c **instanceof** Object)

System.***out***.println("c may be cast to Object");

**if**(d **instanceof** Object)

System.***out***.println("d may be cast to Object");

}

}

o/p:

a is instance of A

b is instance of B

c is instance of C

c can be cast to A

ob now refers to d

ob is instance of D

ob now refers to c

ob cannot be cast to D

ob can be cast to A

a may be cast to Object

b may be cast to Object

c may be cast to Object

d may be cast to Object

**Static Import:**

**import static** statement. The first, which is used by

the preceding example, brings into view a single name. Its general form is shown here:

import static *pkg*.*type-name*.*static-member-name*;

Here, *type-name* is the name of a class or interface that contains the desired static member. Its full

package name is specified by *pkg.* The name of the member is specified by *static-member-name.*

The second form of static import imports all static members of a given class or interface.

Its general form is shown here:

import static *pkg*.*type-name*.\*;

If you will be using many static methods or fields defined by a class, then this form lets you

bring them into view without having to specify each individually