Unit 4

Exception handling mechanism. new look try/catch mechanism in Java Enumeration in Java 5 - usage.

exceptions

- An *exception* is an abnormal condition that arises in a code sequence at run time.
- an exception is a runtime error.
- In computer languages that do not support exception handling, errors must be checked and handled manually—typically through the use of error codes, and so on.

Exception-Handling Fundamentals

- A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code.
- When an exceptional condition arises, an object representing that exception is created and *thrown* in the method that caused the error.
- Java exception handling is managed via five keywords: **try**, **catch**, **throw**, **throws**, and **finally**.

- Program statements that you want to monitor for exceptions are contained within a **try** block.
- If an exception occurs within the **try** block, it is thrown. Your code can catch this exception (using **catch**) and handle it in some rational manner.
- System-generated exceptions are automatically thrown by the Java runtime system.
- To manually throw an exception, use the keyword **throw**. Any exception that is thrown out of a method must be specified as such by a **throws** clause.
- Any code that absolutely must be executed after a **try** completes is put in a **finally** block.

syntax

This is the general form of an exception-handling block:

```
try {
// block of code to monitor for errors
}
catch (ExceptionType1 exOb) {
// exception handler for ExceptionType1
}
catch (ExceptionType2 exOb) {
// exception handler for ExceptionType2
}
// ...
finally {
// block of code to be executed after try block ends
}
```

Here, *ExceptionType* is the type of exception that has occurred.

Exception Types

- All exception types are subclasses of the built-in class **Throwable**. Thus, **Throwable** is at the top of the exception class hierarchy.
- Immediately below **Throwable** are two subclasses that

partition exceptions into two distinct branches. One branch is headed by **Exception**.

There is an important subclass of **Exception**, called **RuntimeException**.

Exceptions of this type are automatically defined for the programs that you write and include things such as division by zero and invalid array indexing.

Exception types

$$\begin{tikzcd}
 & \text{throwable} \arrow[dl] \arrow[dr] & \\
\text{exception} \arrow[d] & & \text{error} \\
\text{Runtime exception} & &
\end{tikzcd}$$

Uncaught Exceptions

This small program includes an expression that intentionally causes a divide-by-zero error:

```
class Exc0 {
public static void main(String args[]) {
int d = 0;
int a = 42 / d;
}
}
```

- When the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and then *throws* this exception.

- Here is the exception generated when this example is executed:
- java.lang.ArithmeticException: / by zero
- at Exc0.main(Exc0.java:4)

The stack trace will always show the sequence of method invocations that led up to the error. For example, here is another version of the preceding program that introduces the same error but in a method separate from **main( )**:

```
class Exc1 {
static void subroutine() {
int d = 0;
int a = 10 / d;
}
public static void main(String args[]) {
Exc1.subroutine();
}
}
```

The resulting stack trace from the default exception handler shows how the entire call stack is displayed:

```
java.lang.ArithmeticException: / by zero
at Exc1.subroutine(Exc1.java:4)
at Exc1.main(Exc1.java:7)
```

As you can see, the bottom of the stack is **main**'s line 7, which is the call to **subroutine( )**, which caused the exception at line 4. The call stack is quite useful for debugging, because it pinpoints the precise sequence of steps that led to the error

Using try and catch

- To guard against and handle a run-time error, simply enclose the code that you want to monitor inside a **try** block.
- Immediately following the **try** block, include a **catch** clause that specifies the exception type that you wish to catch. To illustrate how easily this can be done, the following program includes a **try** block and a **catch** clause that processes the

ArithmeticException generated by the division-by-zero error.

```
class Exc2 {
public static void main(String args[]) {
int d, a;
try { // monitor a block of code.
d = 0;
a = 42 / d;
System.out.println("This will not be printed.");
} catch (ArithmeticException e) { // catch divide-by-zero error
System.out.println("Division by zero.");
}
System.out.println("After catch statement.");
}
}
```

This program generates the following output:

```
Division by zero.
After catch statement.
```

- A **try** and its **catch** statement form a unit.
- The scope of the **catch** clause is restricted to those statements specified by the immediately preceding **try** statement.
- A **catch** statement cannot catch an exception thrown by another **try** statement (except in the case of nested **try** statements, described shortly).
- The statements that are protected by **try** must be surrounded by curly braces

If either division operation causes a divide-by-zero error, it is caught, the value of **a** is set to zero, and the program continues.

```
// Handle an exception and move on.
import java.util.Random;
class HandleError {
public static void main(String args[]) {
int a=0, b=0, c=0;
Random r = new Random();
for(int i=0; i<32000; i++) {
try {
b = r.nextInt();
c = r.nextInt();
a = 12345 / (b/c);
} catch (ArithmeticException e) {
System.out.println("Division by zero.");
a = 0; // set a to zero and continue
}
System.out.println("a: " + a);
}
}
}
```

Displaying a Description of an Exception

- **Throwable** overrides the **toString( )** method (defined by **Object**) so that it returns a string containing a description of the exception. You can display this description in a **println( )** statement by simply passing the exception as an argument.
- For example, the **catch** block in the preceding program can be rewritten like this:

```
catch (ArithmeticException e) {
System.out.println("Exception: " + e);
a = 0; // set a to zero and continue
}
```

- When this version is substituted in the program, and the program is run, each divide-by zero error displays the following message:
- Exception: java.lang.ArithmeticException: / by zero

Multiple catch Clauses

- When an exception is thrown, each **catch** statement is inspected in order, and the first one whose type matches that of the exception is executed.
- After one **catch** statement executes, the others are bypassed, and execution continues after the **try / catch** block

```
// Demonstrate multiple catch statements.
class MultipleCatches {
public static void main(String args[]) {
try {
int a = args.length;
System.out.println("a = " + a);
int b = 42 / a;
int c[] = { 1 };
c[42] = 99;
} catch(ArithmeticException e) {
System.out.println("Divide by 0: " + e);
} catch(ArrayIndexOutOfBoundsException e) {
System.out.println("Array index oob: " + e);
}
System.out.println("After try/catch blocks.");
}
}
```

```
C:\>java MultipleCatches
a = 0
Divide by 0: java.lang.ArithmeticException: / by zero
After try/catch blocks.
C:\>java MultipleCatches TestArg
a = 1
Array index oob:
java.lang.ArrayIndexOutOfBoundsException:42
After try/catch blocks.
```

Further, in Java, unreachable code is an error. For example, consider the following program:

```
/* This program contains an error. A subclass must come before its superclass in
a series of catch statements. If not, unreachable code will be created and a
compile-time error will result.     */
class SuperSubCatch {
public static void main(String args[]) {
try {
int a = 0;
int b = 42 / a;
} catch(Exception e) {
System.out.println("Generic Exception catch.");
}
/* This catch is never reached because
ArithmeticException is a subclass of Exception. */
catch(ArithmeticException e) { // ERROR – unreachable
System.out.println("This is never reached.");
}  }  }
```

Nested try Statements

- The **try** statement can be nested. That is, a **try** statement can be inside the block of another **try**. Each time a **try** statement is entered, the context of that exception is pushed on the stack.
- If an inner **try** statement does not have a **catch** handler for a particular exception, the stack is unwound and the next **try** statement's **catch** handlers are inspected for a match.

```
// An example of nested try statements.
class NestTry {
public static void main(String args[]) {
try {
int a = args.length;
/* If no command-line args are present,
the following statement will generate
a divide-by-zero exception. */
int b = 42 / a;
System.out.println("a = " + a);
try { // nested try block
/* If one command-line arg is used,
then a divide-by-zero exception
will be generated by the following
code.*/
if(a==1) a = a/(a-a); // division by zero
/* If two command-line args are used,
then generate an out-of-bounds
exception. */
if(a==2) {
int c[] = { 1 };
c[42] = 99; // generate an out-of-
boundsexception
}
}
catch(ArrayIndexOutOfBoundsException
e) {
System.out.println("Array index out-of-
bounds: " + e);
}
} catch(ArithmeticException e) {
System.out.println("Divide by 0: " + e);
}
}
}
```

```
C:\>java NestTry
Divide by 0: java.lang.ArithmeticException: / by zero
C:\>java NestTry One
a = 1
Divide by 0: java.lang.ArithmeticException: / by zero
C:\>java NestTry One Two
a = 2
Array index out-of-bounds:
java.lang.ArrayIndexOutOfBoundsException:42
```

- In this case, the **try** within the method is still nested inside the outer **try** block, which calls the method. Here is the previous program recoded so that the nested **try** block is moved inside the method **nesttry( )**:

```
/* Try statements can be implicitly nested via
calls to methods. */
class MethNestTry {
static void nesttry(int a) {
try { // nested try block
/* If one command-line arg is used,
then a divide-by-zero exception
will be generated by the following code. */
if(a==1) a = a/(a-a); // division by zero
/* If two command-line args are used,
then generate an out-of-bounds exception. */
if(a==2) {
int c[] = { 1 };
c[42] = 99; // generate an out-of-bounds exception
}
} catch(ArrayIndexOutOfBoundsException e) {
System.out.println("Array index out-of-bounds: " +
e);
}
}
public static void main(String args[]) {
try {
int a = args.length;
/* If no command-line args are present,
the following statement will generate
a divide-by-zero exception. */
int b = 42 / a;
System.out.println("a = " + a);
nesttry(a);
} catch(ArithmeticException e) {
System.out.println("Divide by 0: " + e);
}
}
}
```

throw

- However, it is possible for your program to throw an exception explicitly, using the **throw** statement. The general form of **throw** is shown here:
- throw *ThrowableInstance*;
- Here, *ThrowableInstance* must be an object of type **Throwable** or a subclass of **Throwable**.
- Primitive types, such as **int** or **char**, as well as non-**Throwable** classes, such as **String** and **Object**, cannot be used as exceptions.
- There are two ways you can obtain a **Throwable** object: using a parameter in a **catch** clause or creating one with the **new** operator.

Here is a sample program that creates and throws an exception. The handler that catches the exception rethrows it to the outer handler.

```
// Demonstrate throw.
class ThrowDemo {
static void demoproc() {
try {
throw new NullPointerException("demo");
} catch(NullPointerException e) {
System.out.println("Caught inside demoproc.");
throw e; // rethrow the exception
}
}

public static void main(String args[]) {
try {
demoproc();
} catch(NullPointerException e) {
System.out.println("Recaught: " + e);
}
}
}
```

This program gets two chances to deal with the same error. First, **main( )** sets up an exception context and then calls **demoproc( )**. The **demoproc( )** method then sets up another exception-handling context and immediately throws a new instance of

NullPointerException, which is caught on the next line. The exception is then rethrown.

Here is the resulting output:

Caught inside demoproc.

Recaught: java.lang.NullPointerException: demo

The program also illustrates how to create one of Java's standard exception objects.

Pay close attention to this line: throw new NullPointerException("demo");

Here, **new** is used to construct an instance of **NullPointerException**

throws

- A **throws** clause lists the types of exceptions that a method might throw. This is necessary for all exceptions, except those of type **Error** or **RuntimeException**, or any of their subclasses. All other exceptions that a method can throw must be declared in the **throws** clause. If they are not, a compile-time error will result.
- This is the general form of a method declaration that includes a **throws** clause:

```
type method-name(parameter-list) throws exception-list
{
// body of method
}
```

Here, *exception-list* is a comma-separated list of the exceptions that a method can throw.

Because the program does not specify a **throws** clause to declare this fact,
the program will not compile.

```
// This program contains an error and will not compile.
class ThrowsDemo {
static void throwOne() {
System.out.println("Inside throwOne.");
throw new IllegalAccessException("demo");
}
public static void main(String args[]) {
throwOne();
}
}
```

make this example compile, you need to make two changes. First, you need to declare that **throwOne( )** throws **IllegalAccessException**. Second, **main( )** must define a **try / catch** statement that catches this exception.

The corrected example is shown here:

```
// This is now correct.
class ThrowsDemo {
static void throwOne() throws IllegalAccessException {
System.out.println("Inside throwOne.");
throw new IllegalAccessException("demo");
}
public static void main(String args[]) {
try {
throwOne();
} catch (IllegalAccessException e) {
System.out.println("Caught " + e);
}
}
}
```

Here is the output generated by running this example program:

```
inside throwOne
caught java.lang.IllegalAccessException: demo
```

finally

- **finally** creates a block of code that will be executed after a **try /catch** block has completed and before the code following the **try/catch** block.
- The **finally** block will execute whether or not an exception is thrown.
- If an exception is thrown, the **finally** block will execute even if no **catch** statement matches the exception. Any time a method is about to return to the caller from inside a **try/catch** block, via an uncaught exception or an explicit return statement, the **finally** clause is also executed just before the method returns.
- The **finally** clause is optional. However, each **try** statement requires at least one **catch** or a **finally** clause.

```
// Demonstrate finally.
class FinallyDemo {
// Throw an exception out of the method.
static void procA() {
try {
System.out.println("inside procA");
throw new RuntimeException("demo");
} finally {
System.out.println("procA's finally");
}
}
// Return from within a try block.
static void procB() {
try {
System.out.println("inside procB");
return;
} finally {
System.out.println("procB's finally");
}
}
// Execute a try block normally.
static void procC() {
try {
System.out.println("inside procC");
} finally {
System.out.println("procC's finally");
}
}
public static void main(String args[]) {
try {
procA();
} catch (Exception e) {
System.out.println("Exception caught");
}
procB();
procC();
}
}
```

output

- Here is the output generated by the preceding program:
- inside procA
- procA's finally
- Exception caught
- inside procB
- procB's finally
- inside procC
- procC's finally

Java’s Built-in Exceptions

- Checked exceptions
- lists those exceptions defined by **java.lang** that must be included in a method’s **throws** list if that method can generate one of these exceptions and does not handle it itself.
- These are called *checked exceptions*.

Unchecked exceptions

- these exceptions need not be included in any method’s **throws** list.
- In the language of Java, these are called *unchecked exceptions* because the compiler does not check to see if a method handles or throws these exceptions.

Java's Unchecked **RuntimeException** Subclasses Defined in **java.lang**

- **Exception** **Meaning**

ArithmeticException ----------Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException -------------------Array index is out-of-bounds.
ArrayStoreException --------------------------------------Assignment to an array element of an incompatible type.
ClassCastException ----------------------Invalid cast.
EnumConstantNotPresentException ---------------An attempt is made to use an undefined enumeration value.
IllegalArgumentException ---------------------------Illegal argument used to invoke a method.
IllegalMonitorStateException -------------------Illegal monitor operation, such as waiting on an unlocked thread.
IllegalStateException -----------------------------Environment or application is in incorrect state.
IllegalThreadStateException---------------- Requested operation not compatible with current thread state.
IndexOutOfBoundsException ----------------------Some type of index is out-of-bounds.
NegativeArraySizeException -------------Array created with a negative size.
NullPointerException---------------------------- Invalid use of a null reference.
NumberFormatException I--------------------nvalid conversion of a string to a numeric format.
SecurityException -------------------------------Attempt to violate security.
StringIndexOutOfBounds -------------------------------Attempt to index outside the bounds of a string.
TypeNotPresentException --------------------------------type not found.
UnsupportedOperationException ------------------------An unsupported operation was encountered.

Java’s Checked Exceptions Defined in **java.lang**

- **Exception** **Meaning**

ClassNotFoundException -------------Class not found.

CloneNotSupportedException------------ Attempt to clone an object that does not implement the **Cloneable** interface.

IllegalAccessException ----------Access to a class is denied.

InstantiationException----------- Attempt to create an object of an abstract class or interface.

InterruptedException -----------One thread has been interrupted by another thread.

NoSuchFieldException ---------A requested field does not exist.

NoSuchMethodException---------- A requested method does not exist.

ReflectiveOperationException ----------Superclass of reflection-related exceptions.

Creating Your Own Exception Subclasses

- Although Java’s built-in exceptions handle most common errors, you will probably want to create your own exception types to handle situations specific to your applications.
- This is quite easy to do: just define a subclass of **Exception** (which is, of course, a subclass of **Throwable**). Your subclasses don’t need to actually implement anything—it is their existence in the type system that allows you to use them as exceptions.
- The **Exception** class does not define any methods of its own. It does, of course, inherit those methods provided by **Throwable**. Thus, all exceptions, including those that you create, have the methods defined by **Throwable** available to them.

- **Exception** defines four public constructors. Two support chained exceptions, described in the next section. The other two are shown here:
- Exception( )
- Exception(String *msg*)
- The first form creates an exception that has no description.
- The second form lets you specify a description of the exception.
- The version of **toString( )**
- defined by **Throwable** (and inherited by **Exception**) first displays the name of the exception followed by a colon, which is then followed by your description. By overriding **toString( )**

The Methods Defined by **Throwable**

- final void addSuppressed(Throwable *exc*)---------------------------Adds *exc* to the list of suppressed exceptions associated with the invoking exception. Primarily for use by the **try**-with-resources statement.
- Throwable fillInStackTrace( ) --------------Returns a **Throwable** object that contains an completed stack trace. This object can be rethrown.
- Throwable getCause( ) -----Returns the exception that underlies the current exception. If there is no underlying exception, **Null** is returned.
- String getLocalizedMessage( )------- Returns a localized description of the exception.
- String getMessage( ) -----------Returns a description of the exception.
- String toString( ) -------------Returns a **String** object containing a description of the exception. This method is called by **println( )** when outputting a **Throwable** object.

```
// This program creates a custom exception type.
class MyException extends Exception {
private int detail;
MyException(int a) {
detail = a;
}
public String toString() {
return "MyException[" + detail + "]";
}
}
```

```
class ExceptionDemo {
static void compute(int a) throws MyException {
System.out.println("Called compute(" + a + ")");
if(a > 10)
{    throw new MyException(a);          }
System.out.println("Normal exit");
}
public static void main(String args[]) {
try {
compute(1);
compute(20);
} catch (MyException e) {
System.out.println("Caught " + e);
}
}
}
```

```
class ExceptionDemo {
static void compute(int a) throws MyException {
System.out.println("Called compute(" + a + ")");
if(a > 10)
{   throw new MyException(a);          }
System.out.println("Normal exit");
}
public static void main(String args[]) {
try {
compute(1);
compute(20);
} catch (MyException e) {
System.out.println("Caught " + e);
}
}
}
```

```
Called compute(1)
Normal exit
Called compute(20)
Caught MyException[20]
```

- The **main( )** method sets up an exception handler for **MyException**, then calls **compute( )** with a legal value (less than 10) and an illegal one to show both paths through the code. Here is the result:

```
Called compute(1)
Normal exit
Called compute(20)
Caught MyException[20]
```

Chained Exceptions

- Beginning with JDK 1.4, a feature was incorporated into the exception subsystem: *chained exceptions*.
- The chained exception feature allows you to associate another exception with an exception. This second exception describes the cause of the first exception.

.

- To allow chained exceptions, two constructors and two methods were added to **Throwable**. The constructors are shown here:

Throwable(Throwable *causeExc*)

Throwable(String *msg*, Throwable *causeExc*)

- In the first form, *causeExc* is the exception that causes the current exception. That is *causeExc* is the underlying reason that an exception occurred.
- The second form allows you to specify a description at the same time that you specify a cause exception. These two constructors have also been added to the **Error**, **Exception**, and **RuntimeException** classes.

.

- The chained exception methods supported by **Throwable** are **getCause( )** and **initCause( )**. These methods are shown in Table 10-3 and are repeated here for the sake of discussion.
- Throwable getCause( )
- Throwable initCause(Throwable *causeExc*)
- The **getCause( )** method returns the exception that underlies the current exception.
- If there is no underlying exception, **null** is returned. The **initCause( )** method associates *causeExc* with the invoking exception and returns a reference to the exception

```
// Demonstrate exception chaining.
class ChainExcDemo {
static void demoproc() {
// create an exception
NullPointerException e =
new NullPointerException("top layer");
// add a cause
e.initCause(new ArithmeticException("cause"));
throw e;
}
public static void main(String args[]) {
try {
demoproc();
} catch(NullPointerException e) {
// display top level exception
System.out.println("Caught: " + e);
// display cause exception
System.out.println("Original cause: " +
e.getCause());
}
}
}
```

The output from the program is shown here:

- Caught: java.lang.NullPointerException: top layer
- Original cause: java.lang.ArithmeticException: cause

Three Recently Added Exception Features

- The first automates the process of releasing a resource, such as a file, when it is no longer needed. It is based on an expanded form of the **try** statement called ***try**-with-resources*
- The second feature is called *multi-catch*
- the third is sometimes referred to as *final rethrow* or *more precise rethrow*.

Multi catch

ArithmeticException and **ArrayIndexOutOfBoundsException**:

```
catch(ArithmeticException | ArrayIndexOutOfBoundsException e)
 {
```

The following program shows the multi-catch feature in action:

```
// Demonstrate the multi-catch feature.
class MultiCatch {
public static void main(String args[]) {
int a=10, b=0;
int vals[] = { 1, 2, 3 };
try {
int result = a / b; // generate an ArithmeticException
// vals[10] = 19; // generate an ArrayIndexOutOfBoundsException
// This catch clause catches both exceptions.
}
catch(ArithmeticException | ArrayIndexOutOfBoundsException e)
 {
System.out.println("Exception caught: " + e);
}
System.out.println("After multi-catch.");
}
}
```

- Automatic resource management is based on an expanded form of the **try** statement.
- Here is its general form:
- try (*resource-specification*) {
- // use the resource
- }
- Here, *resource-specification* is a statement that declares and initializes a resource, such as a file stream.
- It consists of a variable declaration in which the variable is initialized with a reference to the object being managed.
- When the **try** block ends, the resource is automatically released.

```
/* This version of the ShowFile program uses a try-with resources
statement to automatically close a file after it is no loner needed.
Note: This code requires JDK 7 or later.
*/
import java.io.*;
class ShowFile {
public static void main(String args[])
{
int i;
// First, confirm that a filename has been specified.
if(args.length != 1) {
System.out.println("Usage: ShowFile filename");
Return;
}
// The following code uses a try-with-resources statement to open
// a file and then automatically close it when the try block is left.
try(FileInputStream fin = new FileInputStream(args[0])) {
do {
i = fin.read();
if(i != -1) System.out.print((char) i);
} while(i != -1);
} catch(FileNotFoundException e) {
System.out.println("File Not Found.");
} catch(IOException e) {
System.out.println("An I/O Error Occurred");
}
}
}
```

In the program, pay special attention to how the file is opened within the **try** statement:

```
try(FileInputStream fin = new FileInputStream(args[0])) {
```

single **try-**with resources

statement to manage both **fin** and **fout**.

```
/* A version of CopyFile that uses try-with-resources.
It demonstrates two resources (in this case files) being
managed by a single try statement.
*/
import java.io.*;
class CopyFile {
public static void main(String args[]) throws IOException
{
int i;
// First, confirm that both files have been specified.
if(args.length != 2) {
System.out.println("Usage: CopyFile from to");
return;
}
// Open and manage two files via the try statement.
try (FileInputStream fin = new FileInputStream(args[0]);
FileOutputStream fout = new FileOutputStream(args[1]))
{
do {
i = fin.read();
if(i != -1) fout.write(i);
} while(i != -1);
} catch(IOException e) {
System.out.println("I/O Error: " + e);
}
}
}
```

In this program, notice how the input and output files are opened within the **try** block:

```
try (FileInputStream fin = new FileInputStream(args[0]);
FileOutputStream fout = new FileOutputStream(args[1]))
{
// ...
```

After this **try** block ends, both **fin** and **fout** will have been closed.

Enumerations

- *enumeration* is a list of named constants.
- Java enumerations appear similar to enumerations in other languages.
- However, this similarity may be only skin deep because, in Java, an enumeration defines a class type. For example, in Java, an enumeration can have constructors, methods, and instance variables.

Enumeration Fundamentals

- An enumeration is created using the **enum** keyword. For example, here is a simple
- enumeration that lists various apple varieties:

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

The identifiers **Jonathan**, **GoldenDel**, and so on, are called *enumeration constants*

For example, this declares **ap** as a variable of enumeration type **Apple**:

- Apple ap;

Because **ap** is of type **Apple**, the only values that it can be assigned (or can contain) are those defined by the enumeration. For example, this assigns **ap** the value **RedDel**:

```
ap = Apple.RedDel;
```

Notice that the symbol **RedDel** is preceded by **Apple**.

Two enumeration constants can be compared for equality by using the = = relational operator. For example, this statement compares the value in **ap** with the **GoldenDel** constant:

- if(ap == Apple.GoldenDel) // ...

An enumeration value can also be used to control a **switch** statement. Of course, all of the **case** statements must use constants from the same **enum** as that used by the **switch** expression. For example, this **switch** is perfectly valid:

```
// Use an enum to control a switch statement.
switch(ap) {
case Jonathan:
// ...
case Winesap:
```

When an enumeration constant is displayed, such as in a **println( )** statement, its name is output. For example, given this statement:

```
System.out.println(Apple.Winesap);
```

the name **Winesap** is displayed.

```java
// 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.
```

The values( ) and valueOf( ) Methods

- 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.

For example, in the

- case of the **Apple** enumeration shown earlier, the return type of **Apple.valueOf("Winesap")** is **Winesap**.

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

output

- The output from the program is shown here:
- Here are all Apple constants:
- Jonathan
- GoldenDel
- RedDel
- Winesap
- Cortland
- ap contains Winesap

- However, this step is not necessary because the **for** could have been written as shown here, eliminating the need for the **allapples** variable:
- for(Apple a : Apple.values())
- System.out.println(a);
- Now, notice how the value corresponding to the name **Winesap** was obtained by calling **valueOf( )**.
- ap = Apple.valueOf("Winesap");
- As explained, **valueOf( )** returns the enumeration value associated with the name of the constant represented as a string.

Java Enumerations Are Class Types

- The fact that **enum** defines a class gives the Java enumeration extra ordinary power.
- For example, you can give them constructors, add instance variables and methods, and even implement interfaces.
- It is important to understand that each enumeration constant is an object of its enumeration type. Thus, when you define a constructor for an **enum**, the constructor is called when each enumeration constant is created. Also, each enumeration constant has its own copy of any instance variables defined by the enumeration.

```
// 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;
// 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.
```

- This version of **Apple** adds three things. The first is the instance variable **price**, which is used to hold the price of each variety of apple. The second is the **Apple** constructor, which is passed the price of an apple. The third is the method **getPrice( )**, which returns the value of **price**.

// Use an enum constructor.

```
enum Apple {
Jonathan(10), GoldenDel(9), RedDel, Winesap(15), Cortland(8);
private int price; // price of each apple
// Constructor
Apple(int p) { price = p; }
// Overloaded constructor
Apple() { price = -1; }
int getPrice() { return price; }
}
```

Notice that in this version, **RedDel** is not given an argument. This means that the default constructor is called, and **RedDel**’s price variable is given the value –1.

Here are two restrictions that apply to enumerations. First, an enumeration can’t

inherit another class.

Second, an **enum** cannot be a superclass.

This means that an **enum** can’t be extended.

Enumerations Inherit Enum

- Although you can’t inherit a superclass when declaring an **enum**, all enumerations automatically inherit one: **java.lang.Enum**. This class defines several methods that are available for use by all enumerations

ordinal

- 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.
- Thus, in the **Apple** enumeration, **Jonathan** has an ordinal value of zero, **GoldenDel** has an ordinal value of 1, **RedDel** has an ordinal value of 2, and so on.

Compare to

- You can compare the ordinal value of two constants of the same enumeration by using
- the **compareTo( )** method. It has this general form:
- final int compareTo(*enum-type e*)
- Here, *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.
- If the invoking constant has an ordinal value less than *e*'s, then **compareTo( )** returns a negative value.
- If the two ordinal values are the same, then zero is returned.

equals

- You can compare for equality an enumeration constant with any other object by using **equals( )**, which overrides the **equals( )** method defined by **Object**.
- Although **equals( )** can compare an enumeration constant to any other object, those two objects will be equal only
- if they both refer to the same constant, within the same enumeration.
- Simply having ordinal values in common will not cause **equals( )** to return true if the two constants are from different enumerations.
- Remember, you can compare two enumeration references for equality by using = =.

- // Demonstrate ordinal(), compareTo(), and equals().
- // An enumeration of apple varieties.
- 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