Exception Handling

- **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.

- That method may choose to handle the exception itself, or pass it on.

- Either way, at some point, the exception is *caught* and processed.

- Exceptions can be generated by the Java run-time system, or they can be manually generated by your code.

- Exceptions thrown by Java relate to fundamental errors that violate the rules of the Java language or the constraints of the Java execution environment.

- Manually generated exceptions are typically used to report some error condition to the caller of a method.

- 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 run-time 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 before a method returns is put in a **finally** block.

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 before try block ends
}
```

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

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.
- This causes the execution of **Exc0** to stop, because once an exception has been thrown
- it must be *caught* by an exception handler and dealt with immediately

the exception is caught by the default handler provided by the Java run-time system.

➢The default handler displays a string describing the exception, prints a stack trace from the point at which the exception occurred, and terminates the program.

➢Here is the output generated when this example is executed.

java.lang.ArithmeticException: / by zero at Exc0.main(Exc0.java:4)

➢Notice how the class name, **Exc0**; the method name, **main**; the filename, **Exc0.java**; and the line number, **4**, are all included in the simple stack trace.

➢Also, notice that the type of the exception thrown is a subclass of **Exception** called **ArithmeticException**, which more specifically describes what type of error happened.

➢Java supplies several built-in exception types that match the various sorts of run-time errors that can be generated

```
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)
```

Using try and catch

Benefits:
First, it allows you to fix the error.
Second, it prevents the program from automatically terminating.

```
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.");
}
}
.
```

Output:
Division by zero.
After catch statement

Displaying a Description of an Exception

We can display this description of an object 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

In some cases, more than one exception could be raised by a single piece of code.

To handle this type of situation, you can specify two or more **catch** clauses, each catching a different type of exception.

•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.

```
class MultiCatch {
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.");
}
}
```

Here is the output generated by running it both ways:

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

```
/* 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.

- This continues until one of the **catch** statements succeeds, or until all of the nested **try** statements are exhausted.

- If no **catch** statement matches, then the Java run-time system will handle the exception.

```
// 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-bounds exception
}
}
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
```

throw

➢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**.

➢Simple 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 into a **catch** clause, or
- creating one with the **new** operator.

- The flow of execution stops immediately after the **throw** statement; any subsequent statements are not executed.
- The nearest enclosing **try** block is inspected to see if it has a **catch** statement that matches the type of the exception.
- If it does find a match, control is transferred to that statement.
- If not, then the next enclosing **try** statement is inspected, and so on.
- If no matching **catch** is found, then the default exception handler halts the program and prints the stack trace.

```
throw new NullPointerException("demo");
```

- **new** is used to construct an instance of **NullPointerException**.
- All of Java's built-in run-time exceptions have at least two constructors: one with no parameter and one that takes a string parameter.
- When the second form is used, the argument specifies a string that describes the exception.
- This string is displayed when the object is used as an argument to **print( )** or **println( )**.

```
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);
}
}
}
```

throws

➢**If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception**.
➢We can do this by including a **throws** clause in the method's declaration. 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.

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

Output:

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

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

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.

- The **finally** clause is optional. However, each **try** statement requires at least one **catch** or a **finally** clause

- 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.

- This can be useful for closing file handles and freeing up any other resources that might have been allocated at the beginning of a method with the intent of disposing of them before returning.

```
class FinallyDemo {
// Through 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:

```
inside procA
procA’s finally
Exception caught
inside procB
procB’s finally
inside procC
procC’s finally
```

Java's Built-in Exceptions

Java's Unchecked RuntimeException Subclasses

they 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 Checked Exceptions Defined in java.lang

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.*

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.
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	Invalid conversion of a string to a numeric format.
SecurityException	Attempt to violate security.
StringIndexOutOfBounds	Attempt to index outside the bounds of a string.
UnsupportedOperationException	An unsupported operation was encountered.

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.

Creating Your Own Exception Subclasses

- define a subclass of **Exception** (which is, of course, a subclass of **Throwable**)

- 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.

- They are shown below:

Throwable fillInStackTrace( )
Returns a **Throwable** object that contains a 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 which contains the description of the exception

Throwable initCause(Throwable *causeExc*)
Associates *causeExc* with the invoking exception as a cause of the invoking exception. Returns a reference to the exception.

```
class MyException extends Exception {
private int detail;
MyException(int a) {
detail = a;
}
public String toString() {
return "MyException[" + detail + "]";
}
}
```

```
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");
}
```

Output:

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