Exception handling

- Introduction of Exception
- Define exceptions,
- Use of try, catch, and finally statements.
- Exception categories.
- Common exceptions,
- Defining own exceptions

- A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code.(at run time)
- 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.
- 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,**
- **finally.**

- Program statements that we 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, we 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.

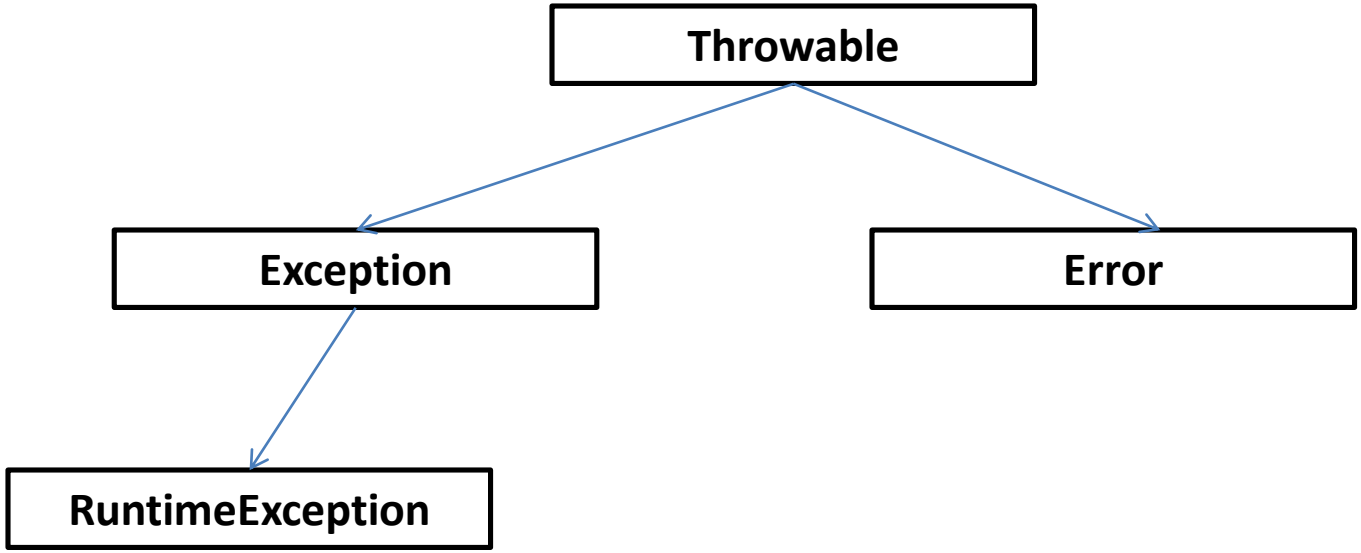
```
try {
// block of code to monitor for errors
}
catch (ExceptionType1 exOb) {
// exception handler for ExceptionType1
}
catch (ExceptionType2 exOb) {
// exception handler for ExceptionType2
}
// ...
finally {
// block of code to be executed before try block ends
}
```

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

Exception Types

- All exception types are subclasses of the built-in class **Throwable**. **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**.
  - This class is used for exceptional conditions that user programs should catch. This is also the class that you will subclass to create your own custom exception types.
  - 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.

- The other branch is topped by **Error,** which defines exceptions that are not expected to be caught under normal circumstances by our program.
  - Exceptions of type Error are used by the Java run-time system to indicate errors having to do with the run-time environment, itself. Stack overflow is an example of such an error.


Throwable
Exception
Error
RuntimeException

Uncaught Exceptions

```
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.
- In this case, we haven't supplied any exception handlers of our own, so the exception is caught by the default handler provided by the Java run-time system.

- Any exception that is not caught by our program will ultimately be processed by the default handler. 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)
- 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. The stack trace will always show the sequence of method invocations that led up to the error.

Using try and catch

- If we handle error manually, it gives two benefits. First, it allows us 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.");  }  }
```

- This program generates the following output:

  Division by zero.

  After catch statement.

- In some cases, more than one exception could be raised by a single piece of code. To handle this type of situation, we 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.
- The try statement can be nested. 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. If no catch statement matches, then the Java run-time system will handle the exception.

throw

- The throw is used for throwing our own exception.
- 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 we 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.

```
// 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 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( ). It can also be obtained by a call to

  getMessage( ), which is defined by Throwable.

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 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
}
```

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

Java's Built-in Exceptions

- Inside the standard package java.lang, Java defines several exception classes.
- The most general of these exceptions are subclasses of the standard type RuntimeException. Since java.lang is implicitly imported into all Java programs, most exceptions derived from RuntimeException are automatically available. Furthermore, 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.
- 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 are called ***checked exceptions****. Java defines several other types of exceptions that relate to its* various class libraries.

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.

```
import java.lang.Exception;
class MyException extends Exception {
  MyException(String message)
   { super(message); }
}
class Test  {
 public static void main(String args[]) {
  int x=5,y=1000;
  try    {
    float z=(float) x/ (float) y;
    if(z<0.01) { throw new MyException("Number too small");  }  }
 catch(MyException e) {
    System.out.println("Caught my exception");
    System.out.println(e.getMessage());   }
 finally  {
  System.out.println("Will be printed always");
 }  }  }
```

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)
- causeExc is the exception that causes the current exception.

- The chained exception methods supported by Throwable are ***getCause( )***and ***initCause( ).***
  - 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.

- the cause exception can be set only once. Thus, you can call initCause( )only once for each exception object.

```
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

In this example, the top-level exception is NullPointerException. To it is added a cause exception, ArithmeticException. When the exception is thrown out of demoproc( ), it is caught by main( ). There, the top-level exception is displayed, followed by the underlying exception, which is obtained by calling getCause( ).