Exception-Handling

What are Exceptions?

Exception is an abnormal condition that arises when executing a program.

In the languages that do not support exception handling, errors must be checked and handled manually, usually through the use of error codes.

In contrast, Java:

1) provides syntactic mechanisms to signal, detect and handle errors
2) ensures a clean separation between the code executed in the absence of errors and the code to handle various kinds of errors
3) brings run-time error management into object-oriented programming

Exceptions

An exception is an object that describes an exceptional condition (error) that has occurred when executing a program.

Exception handling involves the following:

1) when an error occurs, an object (exception) representing this error is created and thrown in the method that caused it
2) that method may choose to handle the exception itself or pass it on
3) either way, at some point, the exception is caught and processed

Exception Constructs

Five constructs are used in exception handling:

1) `try` – a block surrounding program statements to monitor for exceptions
2) `catch` – together with try, catches specific kinds of exceptions and handles them in some way
3) `finally` – specifies any code that absolutely must be executed whether or not an exception occurs
4) `throw` – used to throw a specific exception from the program
5) `throws` – specifies which exceptions a given method can throw

Exception Handling Block

General form:

```
try { … }
catch(Exception1 ex1) { … }
catch(Exception2 ex2) { … }
…
finally { … }
```

where:

1) `try { … }` is the block of code to monitor for exceptions
2) `catch(Exception ex) { … }` is exception handler for the exception `Exception`
3) `finally { … }` is the block of code to execute before the `try` block ends

Exception Hierarchy

All exceptions are sub-classes of the build-in class `Throwable`.

`Throwable` contains two immediate sub-classes:

1) `Exception` – exceptional conditions that programs should catch

   The class includes:

   a) `RuntimeException` – defined automatically for user programs to include: division by zero, invalid array indexing, etc.
   b) use-defined exception classes

2) `Error` – exceptions used by Java to indicate errors with the run-time environment; user programs are not supposed to catch them

Uncaught Exception

What happens when exceptions are not handled?

```
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 throws this object.

This will cause the execution of `Exc0` to stop – once an exception has been thrown it must be caught by an exception handler and dealt with.

Default Exception Handler

As we have not provided any exception handler, the exception is caught by the default handler provided by the Java run-time system.

This default handler:

1) displays a string describing the exception,
2) prints the stack trace from the point where the exception occurred
3) terminates the program

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

Any exception not caught by the user program is ultimately processed by the default handler.

Stack Trace Display

The stack trace displayed by the default error handler shows the sequence of method invocations that led up to the error.

Here the exception is raised in `subroutine()` which is called by `main()`:

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

Own Exception Handling

Default exception handling is basically useful for debugging.

Normally, we want to handle exceptions ourselves because:

1) if we detected the error, we can try to fix it
2) we prevent the program from automatically terminating

Exception handling is done through the try and catch block.

Try and Catch

1) **try** surrounds any code we want to monitor for exceptions

2) **catch** specifies which exception we want to handle and how.

Try and Catch

```
try
{
  ……..
  ……..
  }
  catch(XYZException e)
  {
  ……..
  }
```

Try and Catch

```
class TestStackTrace{
   TestStackTrace()
   {
       divideByZero();
   }

    int divideByZero()
    {
        return 25/0;
    }

    public static void main(String[]args)
    {
            new TestStackTrace();
    }
}
```

Multiple Catch Clauses

When more than one exception can be raised by a single piece of code, several `catch` clauses can be used with one `try` block:

1) each `catch` catches a different kind of exception
2) when an exception is thrown, the first one whose type matches that of the exception is executed
3) after one `catch` executes, the other are bypassed and the execution continues after the try/catch block

Multiple Catch Clauses

```
public void getCustomers() {
    try{
          fileCustomers. read() ;
    }catch(FileNotFoundException e) {
          System.out. println("Can not find file
  Customers" ) ;
    }catch(EOFException e1) {
             System. out. println( "Done with file
  read") ;
    }catch(IOException e2) {
       System. out. println(" Problem reading  file
  " +  e2. getMessage() ) ;
    }
}
```

Order of Multiple Catch Clauses

Order is important:

1) catch clauses are inspected top-down

2) a clause using a super-class will catch all sub-class exceptions

Therefore, specific exceptions *should appear before* more general ones.

In particular, *exception sub-classes must appear before super-classes*.

Throwing Exceptions

So far, we were only catching the exceptions thrown by the Java system.

In fact, a user program may throw an exception explicitly:

```
throw ThrowableInstance;
```

ThrowableInstance must be an object of type Throwable or its subclass.

throw Follow-up

Once an exception is thrown by:

```
throw ThrowableInstance;
```

1) the flow of control stops immediately
2) the nearest enclosing `try` statement is inspected if it has a `catch` statement that matches the type of exception:
   1) if one exists, control is transferred to that statement
   2) otherwise, the next enclosing `try` statement is examined
3) if no enclosing `try` statement has a corresponding `catch` clause, the default exception handler halts the program and prints the stack

Creating Exceptions

Two ways to obtain a Throwable instance:

1) creating one with the new operator

All Java built-in exceptions have at least two constructors: one without parameters and another with one String parameter:

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

2) using a parameter of the catch clause

```
try { … } catch(Throwable e) { … e … }
```

Example: throw

```
class ThrowDemo {
// The method demoproc throws a NullPointerException exception
// which is immediately caught in the try block and re-thrown:
static void demoproc() {
        try {
                throw new NullPointerException("demo");
        } catch(NullPointerException e) {
                System.out.println("Caught inside demoproc.");
                throw e;
        }
}

// The main method calls demoproc within the try block which catches and handles
    the NullPointerException exception:

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

}
```

throws Declaration

If a method is capable of causing an exception that it does not handle, it must specify this behavior by the `throws` clause in its declaration:

```
type name(parameter-list) throws exception-list {
   ...
}
```

where `exception-list` is a comma-separated list of all types of exceptions that a method might throw.

All exceptions must be listed except `Error` and `RuntimeException` or any of their subclasses, otherwise a compile-time error occurs.

Example 1: throws

The throwOne method throws an exception that it does not catch, nor declares it within the throws clause.

```
class ThrowsDemo {

static void throwOne() {

System.out.println("Inside throwOne.");

throw new IllegalAccessException("demo");


}


public static void main(String args[]) {

throwOne();

}
}
```

Therefore this program does not compile.

Example 2 : throws

Corrected program: throwOne lists exception, main catches it:

```
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

When an exception is thrown:

1) the execution of a method is changed
2) the method may even return prematurely.

This may be a problem is many situations.

For instance, if a method opens a file on entry and closes on exit; exception handling should not bypass the proper closure of the file.

The `finally` block is used to address this problem.

The *finally* clause

The `try/catch` statement requires at least one `catch` or `finally` clause, although both are optional:

```
try { … }
catch(Exception1 ex1) { … } …
finally { … }
```

Executed after try/catch whether of not the exception is thrown.

Any time a method is to return to a caller from inside the try/catch block via:

1) uncaught exception or
2) explicit return

the finally clause is executed just before the method returns.

Example: finally

Three methods to exit in various ways.

```
class FinallyDemo {
// procAprematurely breaks out of the tryby tghrowing an exception, the finally clause is executed on the
    way out:
static void procA() {
          try {
                    System.out.println("inside procA");
                    throw new RuntimeException("demo");
          } finally {
                    System.out.println("procA's finally");
          }
}
// procB's s try statement is exited via a return statement, the finally clause is executed before procB returns:
static void procB() {
          try {
                    System.out.println("inside procB");
                    return;
          } finally {
                    System.out.println("procB's finally");
          }
}
```

Example: finally

In procC, the try statement executes normally without error, however the finally clause is still executed:

```
static void procC() {
        try {
                System.out.println("inside procC");
        } finally {
        System.out.println("procC's finally");
        }
}
```

```
//Demonstration of the three methods:

public static void main(String args[]) {
        try {
                procA();
        } catch (Exception e) {
                System.out.println("Exception caught");
        }
        procB();
        procC();
        }
}
```

Java Built-In Exceptions

The default `java.lang` package provides several exception classes, all sub-classing the `RuntimeException` class.

Two sets of build-in exception classes:

1) unchecked exceptions – the compiler does not check if a method handles or throws there exceptions
2) checked exceptions – must be included in the method's `throws` clause if the method generates but does not handle them

Unchecked Built-In Exceptions 1

Methods that generate but do not handle those exceptions need not declare them in the `throws` clause:

`ArithmeticException`	arithmetic error such as divide-by-zero
`ArrayIndexOutOfBoundsException`	array index out of bounds
`ArrayStoreException`	assignment to an array element of the wrong type
`ClassCastException`	invalid cast
`IllegalArgumentException`	illegal argument used to invoke a method
`IllegalMonitorStateException`	illegal monitor behavior, e.g. waiting on an unlocked thread
`IllegalStateException`	environment of application is in incorrect state

Unchecked Built-In Exceptions 2

`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 null reference
`NumberFormatException`	invalid conversion of a string to a numeric format
`SecurityException`	attempt to violate security
`StringIndexOutOfBounds`	attempt to index outside the the bounds of a string
`UnsupportedOperationException`	an unsupported operation was encountered

Checked Built-In Exceptions

Methods that generate but do not handle those exceptions must declare them in the `throws` clause:

`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 Own Exception Classes

Build-in exception classes handle some generic errors.

For application-specific errors define your own exception classes.

How? Define a subclass of `Exception`:

```
class MyException extends Exception { … }
```

`MyException` need not implement anything – its mere existence in the type system allows to use its objects as exceptions.

Throwable Class

Exception itself is a sub-class of Throwable.

All user exceptions have the methods defined by the Throwable class:

1) Throwable **fillInStackTrace**() – returns a Throwable object that contains a completed stack trace; the object can be rethrown.

2) Throwable **getCause**() – returns the exception that underlies the current exception. If no underlying exception exists, null is returned.

3) String **getLocalizedMessage**() – returns a localized description of the exception.

4) String **getMessage**() – returns a description of the exception

5) StackTraceElement[] **getStackTrace**() – returns an array that contains the stack trace; the method at the top is the last one called before exception.

Example: Own Exceptions

A new exception class is defined, with a private detail variable, a one-parameter constructor and an overridden toString method:

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

Example: Own Exceptions

ctd..

```
class ExceptionDemo {
//The static compute method throws the MyException exception
   whenever its a argument is greater than 10:
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");
}
}

//
```

The main method calls compute with two arguments within a try block that catches the MyException exception:

```
public static void main(String args[]) {
        try {
                compute(1);
                compute(20);
        } catch (MyException e) {
                System.out.println("Caught " + e); }
        }
}
```

Exceptions Usage

New Java programmers should break the habit of returning error codes to signal abnormal exit from a method.

Java provides a clean and powerful way to handle errors and unusual boundary conditions through its:

1) `try`
2) `catch`
3) `finally`
4) `throw` and
5) `throws` statements.

Q &A