Web Technology 11
Exception Handling

Chittaranjan Pradhan
School of Computer Engineering,
KIIT University

Exceptions

- Exception is an abnormal condition that arises when executing a program. *Exception is an error which can be handled. But, an error is an error which can't be handled*
- 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:
  - provides syntactic mechanisms to signal, detect and handle errors
  - ensures a clean separation between the code executed in the absence of errors and the code to handle various kinds of errors
  - brings run-time error management into object-oriented programming

Exception Handling

Exception Handling

- An exception is an object that describes an exceptional condition (error) that has occurred when executing a program
- Exception handling involves the following:
  - when an error occurs, an object (exception) representing this error is created and thrown in the method that caused it
  - that method may choose to handle the exception itself or pass it on
  - either way, at some point, the exception is caught and processed
- The advantage of exception handling is to maintain the normal flow of the application

Exception Sources

Exception Sources

Exceptions can be:

- **generated by the Java run-time system**
  - Fundamental errors that violate the rules of the Java language or the constraints of the Java execution environment
- **manually generated by programmer's code**
  - Such exceptions are typically used to report some error conditions to the caller of a method

Java Built-In Exceptions

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:
  - **unchecked exceptions** - the compiler does not check if a method handles or throws there exceptions
  - **checked exceptions** - must be included in the method's throws clause if the method generates but does not handle them

Unchecked Built-In Exceptions

Unchecked Built-In Exceptions

These are the exceptions which inherit RuntimeException. They are not checked at compile time, but are checked at runtime

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
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

Checked Built-In Exceptions

These are the exceptions which directly inherit Throwable class except RuntimeException and Error. They are checked at compile time

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

Exception Constructs

Exception Constructs

Five constructs are used in exception handling:

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

Exception-Handling Block

Exception-Handling Block

try {...}
catch(Exception1 ex1) {...}
catch(Exception2 ex2) {...}
...
finally {...}

where:

- try {...} is the block of code to monitor for exceptions
- catch(Exception ex) {...} is exception handler for the exception Exception
- finally {...} is the block of code to execute before the try block ends

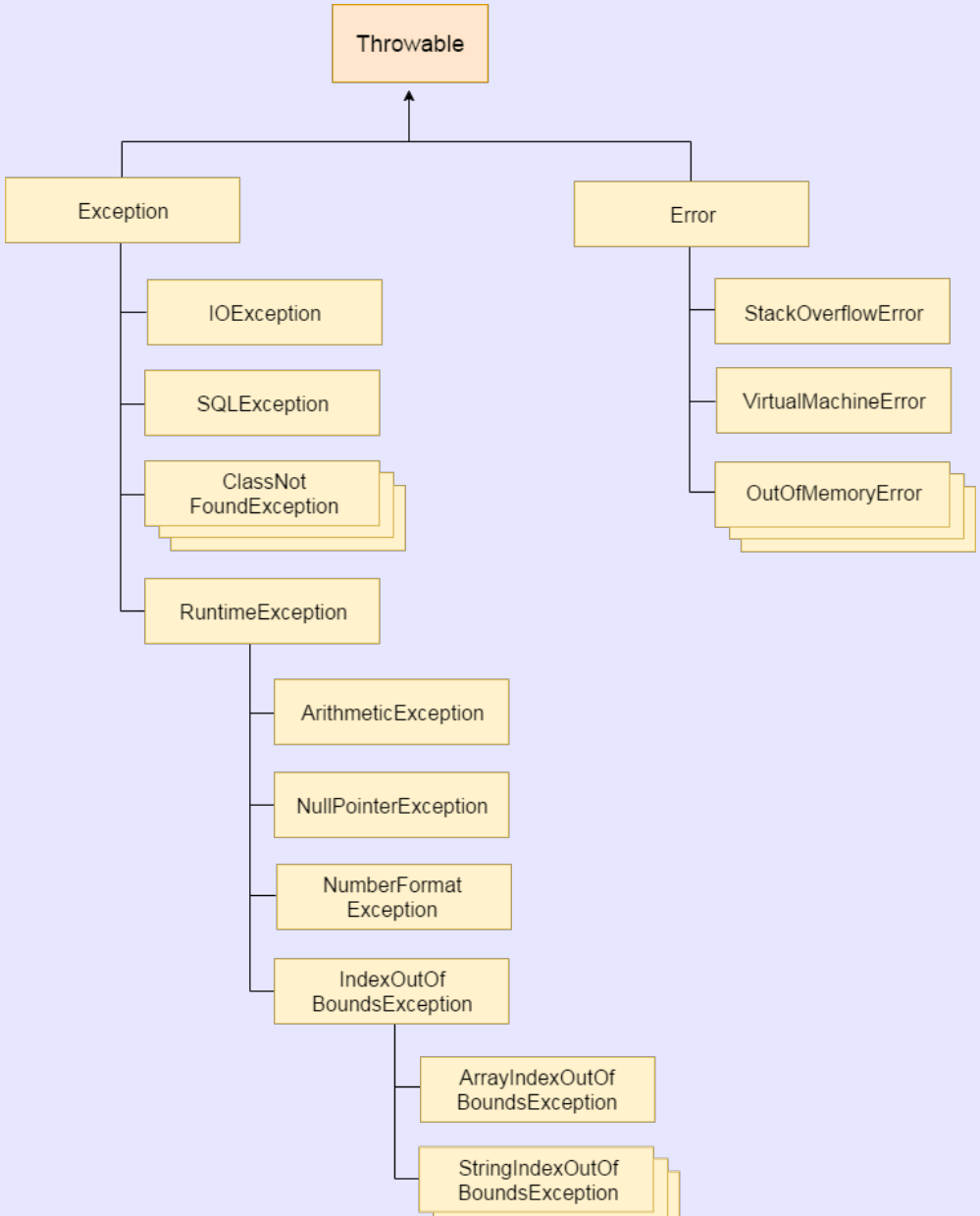
Exception Hierarchy

Exception Hierarchy

- All exceptions are sub-classes of the build-in class Throwable
- Throwable contains two immediate sub-classes:
  - Exception: exceptional conditions that programs should catch. The class includes:
    - RuntimeException: defined automatically for user programs to include: division by zero, invalid array indexing, etc
    - User-defined exception classes
  - Error: exceptions used by Java to indicate errors with the run-time environment; user programs are not supposed to catch them

Exception Hierarchy...

Exception Hierarchy...



Uncaught Exception

Uncaught Exception

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

- 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

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:
  - displays a string describing the exception
  - prints the stack trace from the point where the exception occurred
  - 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

Stack Trace Display

- Stack Trace is a list of method calls from the point when the application was started to the point where the exception was thrown. The most recent method calls are at the top
- It's typically printed to the console when an unexpected error occurs
- It is a kind of call stack, which maintains the call sequence of methods at any given point in time
- *Throwable* class provides a method *public void printStackTrace()* which can be used to print out the entire stack trace

Own Exception Handling

- Default exception handling is basically useful for debugging
- Normally, we want to handle exceptions ourselves because:
  - if we detected the error, we can try to fix it
  - we prevent the program from automatically terminating
- Exception handling is done through the try and catch block

Try and Catch

Try and Catch

- try surrounds any code we want to monitor for exceptions
- catch specifies which exception we want to handle and how

```
try {
    d = 0;
    a = 42 / d;
    System.out.println("This will not be printed.");
    }
```

- control moves immediately to the catch block:

```
catch (ArithmeticException e) {
                    System.out.println("Division by zero.");
                    }
```

- *The exception is handled and the execution resumes*

Try and Catch...

Try and Catch...

- The scope of catch is restricted to the immediately preceding try statement -it cannot catch exceptions thrown by another try statements
- Resumption occurs with the next statement after the try/catch block:

```
try { ... }
catch (ArithmeticException e) { ... }
System.out.println("After catch statement.");
```

- *The purpose of catch should be to resolve the exception and then continue as if the error had never happened*

Try and Catch...

```
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);
                        }
        }       }
```

Exception Display

Exception Display

- All exception classes inherit from the Throwable class
- Throwable overrides toString() to describe the exception textually:

```
try { ... }
catch (ArithmeticException e) {
        System.out.println("Exception: " + e);
        }
```

- The following text will be displayed:
  Exception: java.lang.ArithmeticException: / by zero

Multiple Catch Clauses

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:

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

Multiple Catch Clauses...

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

Multiple Catch Clauses...

Multiple Catch Clauses...

- Order is important:
  - catch clauses are inspected top-down
  - 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

```
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.");
            }
        catch(ArithmeticException e) {
            System.out.println("This is never reached.");
            }
    }
```

Nested try Statements

Nested try Statements

The try statements can be nested:

- If an inner try does not catch a particular exception the exception is inspected by the outer try block
- This continues until:
  - one of the catch statements succeeds or
  - all the nested try statements are exhausted
- In the latter case, the Java run-time system will handle the exception

Nested try Statements...

```
class NestTry {
        public static void main(String args[]) {
                try {
                        int a = args.length;
                        int b = 42 / a;
                        System.out.println("a = " + a);
                        try {
                                if (a==1)
                                        a = a/(a-a);
                                if (a==2) {
                                        int c[] = { 1 };
                                        c[42] = 99;
                                        }
                                }
                        catch(ArrayIndexOutOfBoundsException e) {
                                System.out.println(e);
                                }
                        }
                catch(ArithmeticException e) {
                        System.out.println("Divide by 0: " + e);
                        }
                }
        }
```

finally Block

- When an exception is thrown:
  - the execution of a method is changed
  - 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 will execute whether or not an exception is thrown
- 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*

finally Block...

finally Block...

- Any time a method is to return to a caller from inside the try/catch block via:
  - uncaught exception or
  - explicit return
- The finally clause is executed just before the method returns

```
class FinallyDemo {
    static void procA() {
        try {
                System.out.println("inside procA");
                throw new RuntimeException("demo");
                }
        finally {
                System.out.println("procA's finally");
        }          }
```

- procA prematurely breaks out of the try by throwing an exception, the finally clause is executed

finally Block...

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

- procB's try statement is exited via a return statement, the finally clause is executed before procB returns

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

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

finally Block...

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

Throwing Exceptions

Throwing Exceptions

- The throw statement can be used either to throw our own exceptions or to throw an instance of Java's exception
  throw ThrowableInstance;
- *ThrowableInstance* must be an object of type *Throwable* or its subclass
- Once an exception is thrown by:
  throw ThrowableInstance;
  - the flow of control stops immediately
  - the nearest enclosing try statement is inspected if it has a catch statement that matches the type of exception:
    - if one exists, control is transferred to that statement
    - otherwise, the next enclosing try statement is examined
  - 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:

- The 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");

- using a parameter of the catch clause:

  try {...}
  catch(Throwable e) {... e ...}

Creating Exceptions...

```
class ThrowDemo {
        static void demoproc() {
                try {
                        throw new NullPointerException("demo");
                        }
                catch(NullPointerException e) {
                        System.out.println("Caught inside demoproc.");
                        throw e;
                        }
                }
        public static void main(String args[]) {
                try {
                        demoproc();
                } catch(NullPointerException e) {
                        System.out.println("Recaught: " + e);
                        }
                }
        }
```

throws Statement

throws Statement

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

```
class ThrowsDemo {
        static void throwOne() {
                System.out.println("Inside throwOne.");
                throw new IllegalAccessException("demo");
                }
        public static void main(String args[]) {
                throwOne();
                }                       // this program does not compile
        }
```

throws Statement...

throws Statement...

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

```
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);
                        }
}       }
```

User-Defined Exceptions

- Build-in exception classes handle some generic errors. For application-specific errors define your own exception classes
- Define a subclass of Exception:
  class MyException extends Exception { ... }
  Exception itself is a sub-class of *Throwable*
- All user exceptions have the methods defined by the *Throwable* class
  - *Throwable fillInStackTrace()*: returns a Throwable object that contains a completed stack trace; the object can be rethrown
  - *Throwable getCause()*: returns the exception that underlies the current exception. If no underlying exception exists, null is returned
  - *String getLocalizedMessage()*: returns a localized description of the exception

User-Defined Exceptions...

User-Defined Exceptions...

- ...
  - *String getMessage()*: returns a description of the exception
  - *StackTraceElement[] getStackTrace()*: returns an array that contains the stack trace; the method at the top is the last one called before exception
  - *Throwable initCause(Throwable causeExc)*: associates causeExc with the invoking exception as its cause, returns the exception reference
  - *void printStackTrace()*: displays the stack trace
  - *void printStackTrace(PringStream stream)*: sends the stack trace to the specified stream
  - *void setStackTrace(StackTraceElement elements[])*: sets the stack trace to the elements passed in elements; for specialized applications only
  - *String toString()*: returns a String object containing a description of the exception; called by print() when displaying a Throwable object

User-Defined Exceptions...

```
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);
    }     }         }
```