Exception Handling & Assertions

Fresher Learning Program
January, 2012

People matter, results count.

Objectives of Exception And Assertions

- **Purpose**:
  - To understand what is exception and errors in java, how to handle the exception. understanding of assertion.

- **Product**:
  - To understand exception and its handling mechanism
  - Class hierarchy of exception
  - To know how to use - Try-catch-finally block, throws and throw
  - Understanding of checked and unchecked exceptions
  - To know how to create our own exception (user define exception)
  - To understand Assertions, and how to use it

- **Process**:
  - Theory Sessions along with assignments
  - A recap at the end of the session in the form of Quiz.

Table of Contents

Exception - Basics

- What is an Exception?
- Exception, in general terms, is an error event that alters the normal flow of the program in runtime.
- We can say an exception has occurred when the execution of a program terminates abnormally.
- Causes normal program flow to be disrupted
- Examples
  - Divide by zero errors
  - Invalid input
  - Hard disk crash
  - Heap memory exhausted

Exception Example

```
class DivByZero {
        public static void main(String args[]) {
                System.out.println(3/0);
                System.out.println("Pls. print me.");
        }
}
```

Displays this error message:

```
Exception in thread "main"
java.lang.ArithmeticException: / by zeroat
DivByZero.main(DivByZero.java:3)
```

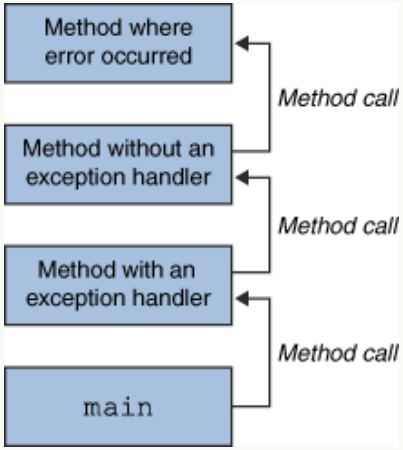
Exception Object

What Happens When an Exception Occurs?

- When an error occurs within a method, the method creates an object and hands it off to the runtime system.
- The object, called an *exception object*, contains information about the error, including its type and the state of the program when the error occurred.
- Creating an exception object and handing it to the runtime system is called “throwing an exception”
- Exception object contains information about the error, including its exception type and the state of the program when the error occurred

Call stack

- After a method throws an exception, the runtime system attempts to find something to handle it.
- The set of possible "something" to handle the exception is the ordered list of methods that had been called to get to the method where the error occurred.
- The list of methods is known as the *call stack* (see the next figure)



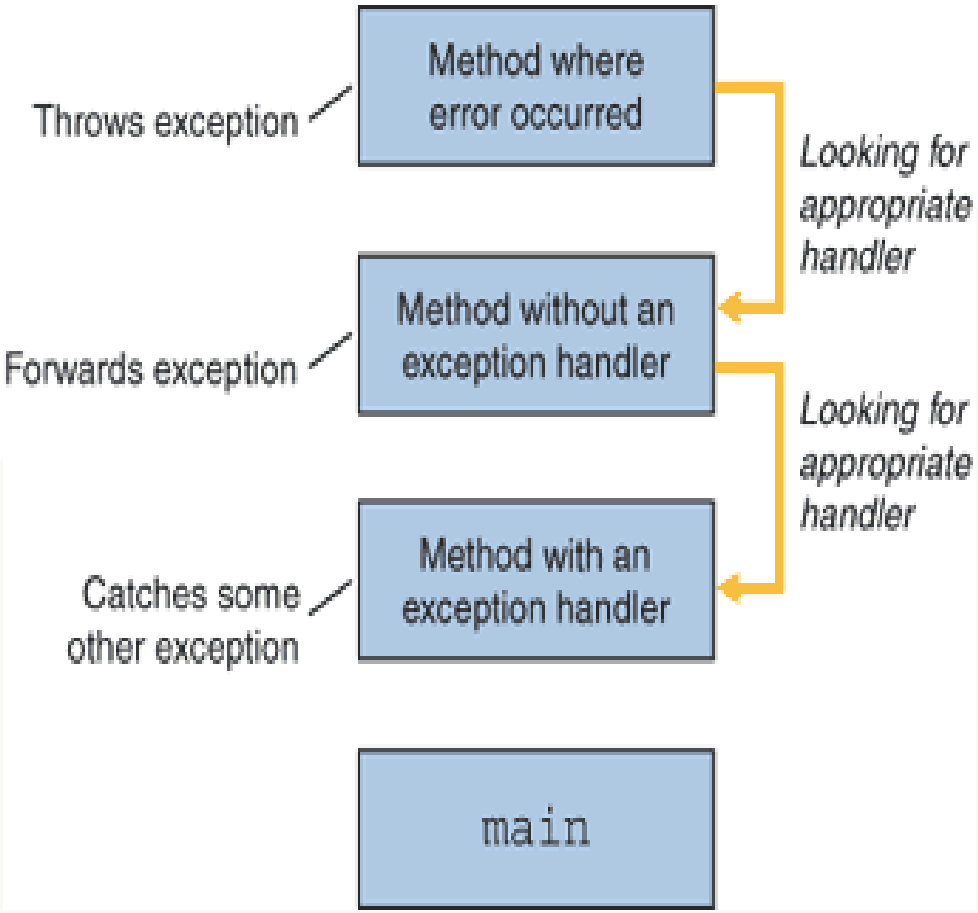
Catch the Exception…

- When an appropriate handler is found, the runtime system passes the exception to the handler
  - An exception handler is considered appropriate if the type of the exception object thrown matches the type that can be handled by the handler
  - The exception handler chosen is said to catch the exception.

- If the runtime system exhaustively searches all the methods on the call stack without finding an appropriate exception handler, the runtime system (and, consequently, the program) terminates and uses the default exception handler

Catch the Exception

- **Searching the Call Stack for an Exception Handler**



Benefits of Exception Handling

- Separating Error-Handling code from “regular” business logic code
- Cleaner code since error handling code and business logic do not have to be mixed up
- Propagating errors up the call stack
  – Whoever best suited to handle the error will handle theerror
- Grouping and differentiating errors based on their types
  – Exception classes are genuine Java classes
  – Exception classes have inheritance hierarchy among themselves

Traditional Programming:

‘In traditional programming, To handle such cases, the *readFile function must have more code to do* error detection, reporting, and handling.

```
errorCodeType readFile {
        initialize errorCode = 0;
        open the file;
        if (theFileIsOpen) {
                determine the length of the file;
                if (gotTheFileLength)
                        allocate that much memory;
                if (gotEnoughMemory)
                        read the file into memory;
                if (readFailed) {
                        errorCode = -1;
                } else {
                        errorCode = -2;
                }
        }
```

Separating Error Handling Code

Exceptions enable you to write the main flow of your code and to deal with the exceptional cases elsewhere

```
readFile {
        try {
                open the file;
                determine its size;
                allocate that much memory;
                read the file into memory;
                close the file;
        } catch (fileOpenFailed) {
                doSomething;
        } catch (sizeDeterminationFailed) {
                doSomething;
        } catch (memoryAllocationFailed) {
                doSomething;
        } catch (readFailed) {
                doSomething;
        } catch (fileCloseFailed) {
                doSomething;
        }
}
```

Catching Exceptions with try-catch

The *try-catch Statements*

Syntax:

```
try {
        <code to be monitored for exceptions>
} catch (<ExceptionType1> <ObjName>) {
        <handler if ExceptionType1 occurs>
}
...
} catch (<ExceptionTypeN> <ObjName>) {
        <handler if ExceptionTypeN occurs>
}
```

The *try-catch Statements*

```
class DivByZero {
   public static void main(String args[]) {
            try {
                        System.out.println(3/0);
                        System.out.println("Please print me.");
             } catch (ArithmeticException exc) {
                        //Division by zero is an ArithmeticException
                        System.out.println(exc);
            }
            System.out.println("After exception.");
    }
}
```

The *try-catch Statements*

- When an exception occurs execution in the try block is terminated, and the program execution control goes to the catch block.
- The control does not come back to try block again.
- It is possible to have multiple catch blocks (for handling different types of exceptions) for a single try block.
- In case of multiple catch blocks, the first one which matches the exception will be invoked and others will be ignored.

Multiple catch

```
class MultipleCatch {
  public static void main(String args[]) {
        try {
              int den = Integer.parseInt(args[0]);
              System.out.println(3/den);
         } catch (ArithmeticException exc) {
              System.out.println("Divisor was 0.");
         } catch (ArrayIndexOutOfBoundsException exc2) {
              System.out.println("Missing argument.");
         }
         System.out.println("After exception.");
  }
}
```

Nested try's

```
class NestedTryDemo {
  public static void main(String args[]){
        try {
              int a = Integer.parseInt(args[0]);
              try {
                    int b = Integer.parseInt(args[1]);
                    System.out.println(a/b);
              } catch (ArithmeticException e) {
                    System.out.println("Div by zero error!");
              }
        } catch (ArrayIndexOutOfBoundsException) {
              System.out.println("Need 2 parameters!");
        }
  }
}
```

Catching Exceptions: The *finally Keyword*

Syntax:

```
try {
        <code to be monitored for exceptions>
} catch (<ExceptionType1> <ObjName>) {
        <handler if ExceptionType1 occurs>
} ...

} finally {
        <code to be executed before the try block ends>
}
```

Contains the code for cleaning up after a try or a catch

Catching Exceptions: The *finally Keyword*

- Purpose of the finally block is mainly to free/release the resources which are used in try/catch block.

- This is a special type of clause which is always executed irrespective of exception conditions.

- The finally block is always executed just before the control goes out of either try block (in a normal condition) OR catch block (in exception condition).

Catching Exceptions: The *finally Keyword*

Finally block is coded after try and all the catch blocks.

```
try {
   // code
} catch( Exception e) {
   // code
} finally {
   // code
}
```

Throwing Exceptions: The *throw Keyword*

- Java allows you to throw exceptions (generate exceptions) using *throw keyword*

  throw <exception object>;

- An exception you throw is an object

- You have to create an exception object in the same way you create any other object

  Example:

  throw new ArithmeticException("testing...");

Example: Throwing Exceptions

```
class ThrowDemo {
  public static void main(String args[]){
        String input = “invalid input”;
        try {
              if (input.equals(“invalid input”)) {
                    throw new RuntimeException("throw demo");
              } else {
                     System.out.println(input);
              }
               System.out.println("After throwing");
         } catch (RuntimeException e) {
               System.out.println("Exception caught:" + e);
        }
   }
}
```

Exceptions – Throws clause

- If the checked exception is not caught in the method where it is occurring (called method), it needs to be passed to the caller of the method (calling method), where it should be handled.

- To indicate that the caller method should handle the exception, the called method must declare the exception in 'throws' clause.

E.g. –

```
public int calledMethod() throws ClassNotFoundException {
        // code throwing the ClassNotFoundException
}
```

The caller method must handle ClassNotFoundException.

Example: Throwing Exceptions

E.g.

```
public int callerMethod() {
    try {
        calledMethod();
    }
    catch (ClassNotFoundException cnfe) {
        // do something (handling)
    }
}
```

What if none of the caller methods handle the thrown exception?

The exception is finally handled by the JVM and JVM prints out the stack trace.

Exception Classes and Hierarchy

- Exception Hierarchy

Exception Class Hierarchy			
Throwable	Error	LinkageError, ...	
		VirtualMachineError, ...	
	Exception	ClassNotFoundException,	
		CloneNotSupportedException,	
		IllegalAccessException,	
		InstantiationException,	
		InterruptedException,	
		IOException,	EOFException,
			FileNotFoundException,
			...
		RuntimeException,	ArithmeticException,
			ArrayStoreException,
			ClassCastException,
			IllegalArgumentException, (IllegalThreadStateException and NumberFormatException as subclasses)
			IllegalMonitorStateException,
			IndexOutOfBoundsException,
			NegativeArraySizeException,
			NullPointerException,
			SecurityException
		...	

Exception Classes and Hierarchy

Multiple catches should be ordered from subclass to super class.

```
class MultipleCatchError {
  public static void main(String args[]){
        try {
              int a = Integer.parseInt(args [0]);
              int b = Integer.parseInt(args [1]);
              System.out.println(a/b);
         } catch (ArrayIndexOutOfBoundsException e) {
         } catch (Exception ex) {
        }
  }
}
```

Checked Exceptions &Unchecked Exceptions

- Checked exception

  – Java compiler checks if the program either catches or lists the occurring checked exception

  – If not, compiler error will occur

- Unchecked exceptions

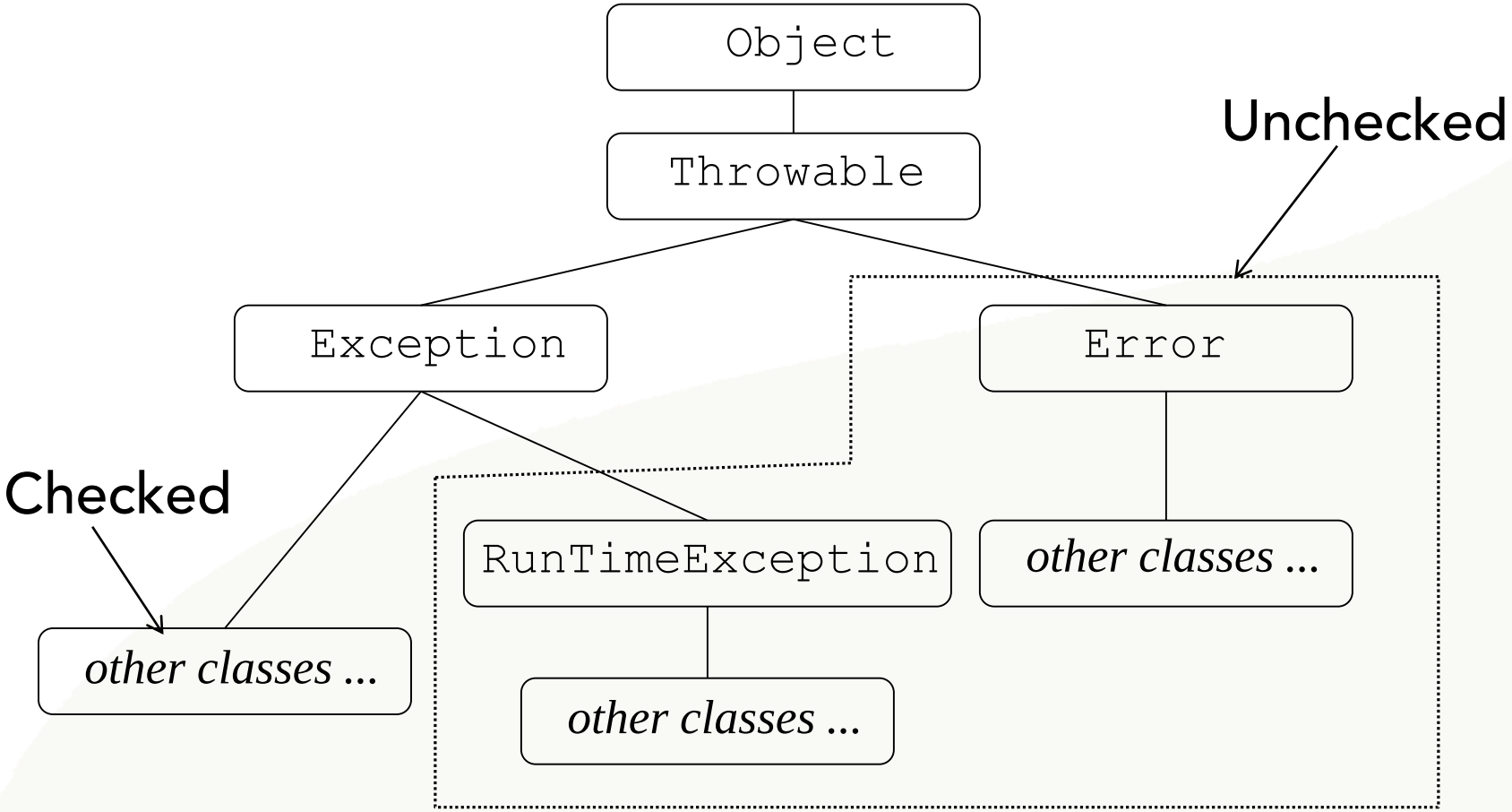
  – Not subject to compile-time checking for exception handling

  – Built-in unchecked exception classes

    - Error
    - RuntimeException
    - Their subclasses

  – Handling all these exceptions may make the program cluttered and may become a nuisance

Exception Class Hierarchy



Difference between Error & Exception Classes

The Error Class

'Error' is a condition which cannot be handled in java code. This is supposed to be handled by the JVM.

The Exception Class

'Exception' is a condition which is supposed to be handled in java code.

Checked Exception Characteristics

- A checked exception is any subclass of Exception (or Exception itself), excluding class RuntimeException and its subclasses.
- You should compulsorily handle the checked exceptions in your code, otherwise your code will not be compiled. i.e you should put the code which may cause checked exception in try block. "checked" means they will be checked at compile time itself.
- There are two ways to handle checked exceptions. You may declare the exception using a throws clause or you may use the try..catch block.

Unchecked Exception Characteristics

- Unchecked exceptions are RuntimeException and any of its subclasses. Class Error and its subclasses also are unchecked.

- Unchecked runtime exceptions represent conditions that, generally speaking, reflect errors in your program's logic and cannot be reasonably recovered from at run time.

- With an unchecked exception, however, compiler doesn't force client programmers either to catch the exception or declare it in a throws clause.

- The most Common examples are:

  ArrayIndexOutOfBoundException, NUllPointerException, ClassCastException

User Defined Exceptions

To handle the specific conditions in the code, it is possible to create user defined exceptions.

Steps to follow

– Create a class that *extends the RuntimeException or the Exception class*

– Customize the class

- Members and constructors may be added to the class
- Example:

```
class HateStringExp extends RuntimeException {
        /* some code */
 }
```

User Defined Exceptions

```
class TestHateString {
        public static void main(String args[]) {
            String input = "invalid input";
            try {
                        if (input.equals("invalid input")) {
                                throw new HateStringExp();
                        }
                        System.out.println("Accept string.");
            } catch (HateStringExp e) {
                        System.out.println("Hate string!”);
            }
        }
}
```

Exceptions - summary

- Exceptions indicate interruption of the normal flow.
- In multiple catch blocks, more specific exceptions should be caught first.
- It is a good practice to at least log the exception in the catch block instead of keeping it empty.
- finally block is the ideal place to release resources
- Exception handling is costly and should be used whenever necessary.

Assertions In Java

What are Assertions?

- Allow the programmer to find out if the program behaves as expected
- Informs the person reading the code that a particular condition should always be satisfied
  - Running the program informs you if assertions made are true or not
  - If an assertion is not true, an *AssertionError is thrown*
- User has the option to turn it off or on when running the application

Enabling or Disabling Assertions

- Program with assertions may not work properly if used by clients not aware that assertions were used in the code

- Compiling

  With assertion feature:

  ```
  javac –source 1.4 MyProgram.java
  ```

  Without the assertion feature:

  ```
  javac MyProgram.java
  ```

- Enabling assertions:

  Use the –enableassertions or –ea switch.

  ```
  java –enableassertions MyProgram
  ```

Assert Syntax

- Two forms:
  - – Simpler form:

    assert <expression1>;

  where

    - <expression1> is the condition asserted to be true

  - – Other form:

    assert <expression1> : <expression2>;

  where

    - <expression1> is the condition asserted to be true
    - <expression2> is some information helpful in diagnosing why the statement failed

Assert Syntax

```
class AgeAssert {
        public static void main(String args[]) {
                int age = Integer.parseInt(args[0]);
                assert(age>0);
                 /* if age is valid (i.e., age>0) */
                if (age >= 18) {
                        System.out.println("You're an adult! =)");
                }
        }
}
```

Recap (Hot keywords)

finally

User define exception

throws

try catch

assertions

unchecked exception

checked exception

runtime exception

throw

Thank You For Your Time

People matter, results count.