Intro to Exceptions

Types of Programming Errors

- Three types of error:
  - **Syntax Errors** – arise because the rules of the language are not followed.
  - **Runtime Errors** – arise because the program tries to perform an operation that is impossible to carry out.
  - **Logic Errors** – arise because the program does perform the way it was intended to.
- **Syntax errors** are caught by the compiler, and fixed before the program is run.
- **Logic Errors** are detected by testing, and are fixed through debugging.
- **Runtime Errors** cause *Exceptions* and may be handled at runtime.

Exceptions

- An *exception* is an event that describes an unusual or erroneous situation at runtime.
- Exceptions are wrapped up as objects
- A program can deal with an exception in one of three ways:
  - ignore it
  - handle it where it occurs
  - handle it an another place in the program
- An *error* is also represented as an object in Java, but usually represents an unrecoverable situation and should not be caught

Why Use Exceptions?

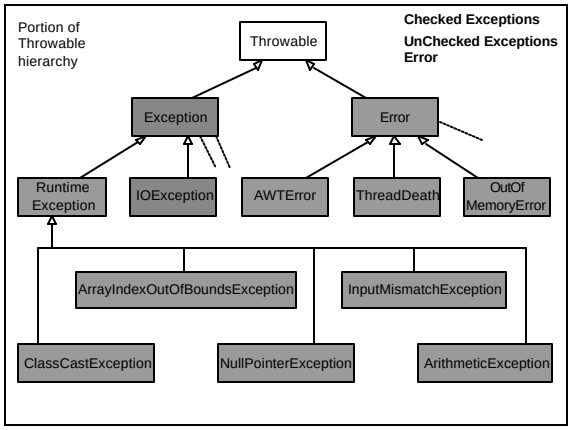
- Uses of exception handling
  - Process exceptions from program components
  - Handle exceptions in a uniform manner in large projects
  - Remove error-handling code from "main line" of execution
- What if Exception is not handled?
  - Might terminate program execution

Exceptions Types

- Two Types
  - Unchecked
    - Subclasses of RuntimeException and Error.
    - Does not require explicit handling
    - Run-time errors are internal to your program, so you can get rid of them by debugging your code
    - For example, null pointer exception; index out of bounds exception; division by zero exception; ...

Exceptions Types

- Two Types
  - Checked
    - Must be caught or declared in a throws clause
    - Compile will issue an error if not handled appropriately
    - Subclasses of Exception other than subclasses of RuntimeException.
    - Other arrive from external factors, and cannot be solved by debugging
    - Communication from an external resource – e.g. a file server or database



How are Java Exceptions Handled

How are Java exceptions handled

- Basic Java exception handling is managed via keywords: **try**, **catch**, **finally, throw, throws**.
- try block
  - Code that could generate errors put in `try` blocks
- catch block
  - Code for error handling enclosed in a `catch` clause
- finally block
  - The `finally` clause always executes
  - Resources that are opened may need to be closed during exception handling
  - Appears after the last catch block
  - It wil not execute if System.exit(0) occurs first

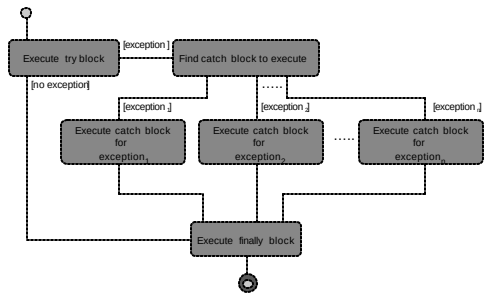
How are Java exceptions handled

- throw
  - To manually throw an exception, use the keyword **throw**.
- throws
  - throws exception out of the method, requiring it to be caught and handled by an appropriate exception handler
  - Any exception that is thrown out of a method must be specified as such by a **throws** clause.

Exception-Handling Struct

```
try                                                    //try block
{
    // write code that could generate exceptions
} catch (<exception to be caught>)                     //catch block
{
    //write code for exception handling
}
 ……
 catch (<exception to be caught>)                      //catch block
{
    //code for exception handling
} finally                                              //finally block
{
    //any clean-up code, release the acquired resources
}
```

try-catch-finally block

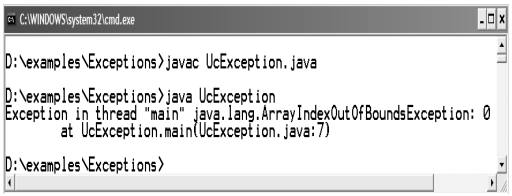


Examples

Example: Unchecked Exceptions

```
public class UcException {
  public static void main(String args[ ]) {
    System.out.println(args[0]);
  }
}
```

Example: Unchecked Exceptions compile & execute



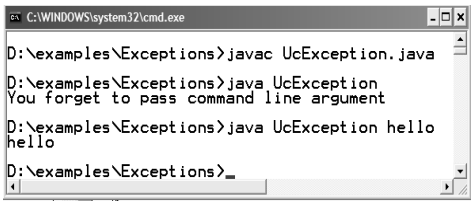
Example: Unchecked Exceptions

```
public class UcException {
  public static void main(String args[ ]) {
    System.out.println(args[0]);
  }
}
```

Example: Unchecked Exceptions Modification

```
public class UcException {
  public static void main(String args[ ]) {
    try {
      System.out.println(args[0]);
    }catch (IndexOutOfBoundsException ex) {
      System.out.println("You forget to pass command line argument");
    }
  }
}
```

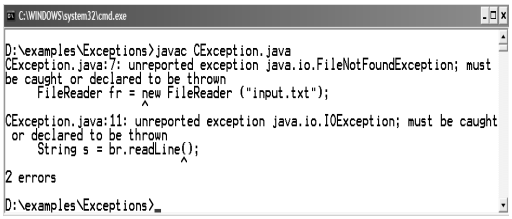
Example: Unchecked Exceptions compile & execute



Example: Checked Exceptions

```
import java.io.*;

public class CException {
  public static void main(String args[ ]) {
     FileReader fr = new FileReader ("input.txt");
     BufferedReader br = new BufferedReader(fr);

     //read the line
     String s = br.readLine();
     System.out.println(s);
  }
}
```

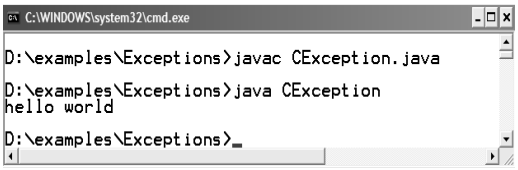
Example: Checked Exceptions compile & execute



Example: Checked Exceptions Modification

```
import java.io.*;
public class CException {
  public static void main(String args[ ]) {
     try {
          FileReader fr = new FileReader ("input.txt");
          BufferedReader br = new BufferedReader(fr);
          //read the line
          String s = br.readLine();
          System.out.println(s);
    } catch (IOException ex ) {
          System.out.println(ex);
    }
  }
}
```

Example: Checked Exceptions compile & execute



Example: finally block

```
import java.io.*;
public class FBlockDemo {
  public static void main(String args[ ]) {
     try {
          FileReader fr = new FileReader ("numbers.txt");
          BufferedReader br = new BufferedReader(fr);

          String s = br.readLine();
          System.out.println(s);

     }catch (IOException ioEx) {
          System.out.println(ioEx);
     } finally {
          System.out.println("finally block always execute");
          //write any clean up code if required
     }

  }//end of main
}//end of class
```

Compile & Execute

If "string.txt" isn't there, it will throw FileNotFoundException
Note that finally block executes

If "string.txt" exist, hopefully no exception would be thrown
Note that finally block still executes

Multiple Catch Blocks

- Possible to have multiple catch clauses for a single try statement
  - Essentially checking for different types of exceptions that may happen
- Evaluated in the order of the code
  - *Bear in mind the Exception hierarchy when writing multiple catch clauses!*
  - If you catch Exception first and then IOException, the IOException will never be caught!

Example: Multiple catch blocks

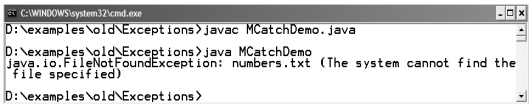
```
/*   numbers.txt contains numbers. After reading number
     from file, prints its square on console  */

import java.io.*;
public class MCatchDemo {
 public static void main(String args[ ]) {
    try {
          //may throw FileNotFound & IOException
          FileReader fr = new FileReader ("numbers.txt");
          BufferedReader br = new BufferedReader(fr);

          //read the line
          String s = br.readLine();

          //may throw NumberFormatException, if s is not no.
          int number = Integer.parseInt(s);
          System.out.println(number * number);
```

Example: Multiple catch blocks

```
    } catch (NumberFormatException nfEx ) {
        System.out.println(nfEx );
    } catch (FileNotFoundException fnfEx) {
          System.out.println(fnfEx);
    } catch (IOException ioEx) {
          System.out.println(ioEx);
    }
  }
}
```

Compile & Execute

If "numbers.txt" isn't there, it will throw FileNotFoundException



If "numbers.txt" exist and contains a number, Hopefully no exception would be occurred

The throws clause

The throws clause

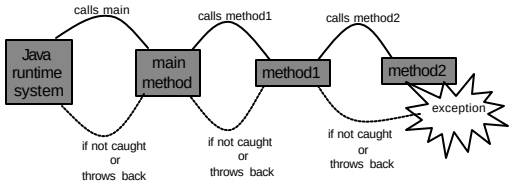
- Method doesn't want to handle exception itself
- it throws the exception, the caller should handle this exception or throws the exception itself
- A method should specify the exceptions it throws by placing a throws clause after the parameter list

printStackTrace() is your friend!

- When dealing with exceptions
- Especially when debugging
- printStackTrace() will:
  - Show you the full calling history
  - With line numbers
- Note:
  - *Bad* idea to eat an exception silently!
  - Either printStackTrace() or pass it along to be handled at a different level

What if

- Method throws back the exception
- No catch blocks matches
- Exception is not handled



Example: throws clause

```
//  this example shows the use of throws clasuse and printStackTrace() method
import java.io.*;
public class ThrowsDemo {
   //method used to read line from file
   public static void method2 () {
     try {
          FileReader fr = new FileReader("string.txt");
          BufferedReader br = new BufferedReader(fr);
          String s = br.readLine();
          System.out.println(s);
    }catch (IOException ioEx) {
       ioEx.printStackTrace();
    }
 } //end of method
```

Example: throws clause

```
   // used to call method1
   public static void method1 () {
      method2();
   }

   public static void  main(String args[]){
      ThrowsDemo.method1();
   }

}//end of class
```

Example: throws clause

- Method2 doesn't want to handle exception itself, so it throws the exception to the caller
- So method2 modified as

```
.....
   public static void method2 ()  throws IOException {

          FileReader fr = new FileReader();
          BufferedReader br = new BufferedReader(fr);
          String s =  br.readLine();
          System.out.println(s);

    } //end of method
.......
```

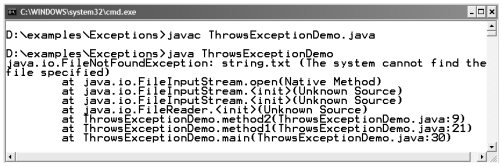
Example: throws clause

- As method2 is throwing the exception and method1 is invoking method 2
- So method1 either can handles the coming exception or rethrows it
- If method1 is handling the exception than method1 would be modified as

```
   // used to call method1
   public static void method1 () {
     try {
        method2();
     catch (IOException ioEx) {
        ioEx.printStackTrace();
     }
   }

   public static void main(String args[]){
      ThrowsDemo.method1();
   }

}//end of class
```

Compile & Execute

If "string.txt" isn't there, it will throw FileNotFoundException



If "string.txt" exist there and contains string