EXCEPTION HANDLING

What is an Exception?

- Exceptional event - **typically an error that occurs during runtime**
- Cause normal program flow to be disrupted
- Examples
  - Divide by zero errors
  - Accessing the elements of an array beyond its range
  - Invalid input
  - Hard disk crash
  - Opening a non-existent file
  - 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 zero at DivByZero.main (DivByZero.java:3)***

→Default exception handler provided by Java runtime and prints out exception description

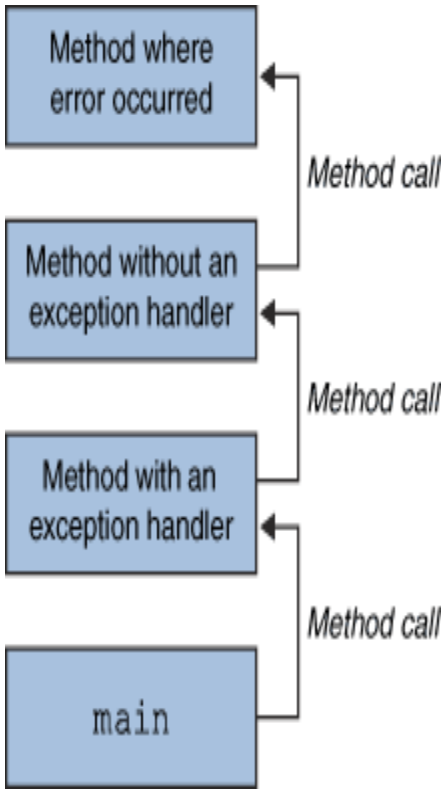
- Prints the stack trace
  - Hierarchy of methods where the exception occurred
- Causes the program to terminate

What Happens When an Exception Occurs?

- When an exception occurs within a method, the method typically creates an exception object and hands it off to the runtime system
- 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 type and
  - the state of the program when the error occurred

What Happens When an Exception Occurs?

- The runtime system searches the call stack for a method that contains an exception handler
- 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



Benefits of Java Exception Handling Framework

- Separating Error-Handling code from “regular” business logic code
- Propagating errors up the call stack
- Grouping and differentiating error types

Separating Error Handling Code from Regular Code

- In traditional programming, error detection, reporting, and handling often lead to confusing spaghetti code
- Consider pseudocode method here that reads an entire file into memory

```
readFile {
    open the file;
    determine its size;
    allocate that much memory;
    read the file into memory;
    close the file;
}
```

Traditional Programming: No separation of error handling code

- 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;
    }
```

Traditional Programming: No separation of error handling code

```
        } else {
                errorCode = -3;
        }
        close the file;
        if (theFileDidntClose && errorCode == 0) {
                errorCode = -4;
        } else {
                errorCode = errorCode and -4;
        }
    } else {
    errorCode = -5;
}
return errorCode;
```

Separating Error Handling Code from Regular Code (in Java)

```
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;
        }
}
```

2-CATCHING EXCEPTIONS

Catching Exceptions

- A method catches an exception using a combination of the **try** and **catch** keywords.
- A try/catch block is placed around the code that might generate an exception.
- Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following:

```
try {
    //Protected code
}
catch(ExceptionName e1) {
    //Catch block
}
```

Catching Exceptions

- A catch statement involves declaring the type of exception you are trying to catch.
- If an exception occurs in protected code, the catch block (or blocks) that follows the try is checked.
- If the type of exception that occurred is listed in a catch block, the exception is passed to the catch block
- much as an argument is passed into a method parameter.

Example

```
import java.io.*;
public class ExcepTest{
      public static void main(String args[]){
        try{
              int a[] = new int[2];
              System.out.println("Access element three :" + a[3]);
      }
      catch(ArrayIndexOutOfBoundsException e){
              System.out.println("Exception thrown :" + e);
         }
        System.out.println("Out of the block");
      }
}
```

//Exception thrown :java.lang.ArrayIndexOutOfBoundsException: 3
Out of the block

Multiple Catch Blocks

- A try block can be followed by multiple catch blocks.
- The syntax for multiple catch blocks looks like the following:

```
try {
    //Protected code
 }catch(ExceptionType1 e1) {
    //Catch block
 }catch(ExceptionType2 e2) {
    //Catch block
 }catch(ExceptionType3 e3) {
    //Catch block
 }
```

Multiple Catch Blocks

- The previous statements demonstrate three catch blocks, but you can have any number of them after a single try.
- If an exception occurs in the protected code, the exception is thrown to the first catch block in the list.
- If the data type of the exception thrown matches ExceptionType1, it gets caught there.
- If not, the exception passes down to the second catch statement.
- This continues until the exception either is caught or falls through all catches, in which case the current method stops execution and the exception is thrown down to the previous method on the call stack.

Multiple Catch Blocks

- When working with multiple catch blocks, you should be extremely careful about how you order them in your code.
- All catch immediately follow try without a code in between.
- A catch block for a more specified type of exception in the inheritance hierarcy should be placed prior to a blcok for a more general type→***otherwise, compiler error, Unreachable catch block***

```
public static void main(String[] args) throws IOException {
   String firstArg;
   String secondArg;
   if (args.length > 0) {
      try {
         firstArg = args[0];
         secondArg = args[1];
      } catch (Exception e) {
         System.out.println("Usage Error: Not enough Arguments");
         System.out.println(e);
         return;
      } catch (IndexOutOfBoundsException e) {
         System.out.println("Usage Error: Not enough Arguments");
         System.out.println(e);
         return;
      }
   }
// Unreachable catch block for IndexOutOfBoundsException. It is already handled by
     the catch block for Exception
```

Example1-Output?

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

Example

```
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 e)    {
                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>
}
...
<other catch blocks>
} finally {

}
```

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

finally

- The finally keyword is used to create a block of code that follows a try block.
- A finally block of code always executes, whether or not an exception has occurred.→*only not executes if System.exit() issued*
- Using a finally block allows you to run any cleanup-type statements that you want to execute, no matter what happens in the protected code.
- A finally block appears at the end of the catch blocks and has the following syntax:

finally

- The finally block is used for code that must always run, whether an error condition (exception) occurred or not.
- The code in the finally block is run after either
  - the try block completes
  - if a caught exception occurred, after the corresponding catch block completes.
- It is always run, even if an uncaught exception occurred in the try or catch block.

finally

- The finally block is typically used for closing files, network connections, etc. that were opened in the try block.
- The reason is that the file or network connection must be closed, whether the operation using that file or network connection succeeded or whether it failed.
- Care should be taken in the finally block to ensure that it does not itself throw an exception.
  - For example, be doubly sure to check all variables for null, etc.
  - Otherwise full code in the block may not be executed!!

```
public class ExcepTest{
    public static void main(String args[]){
        int a[] = new int[2];
        try{
            System.out.println("Access element three :" + a[3]);
        }catch(ArrayIndexOutOfBoundsException e){
            System.out.println("Exception thrown :" + e);
        } finally{
            a[0] = 6;
            System.out.println("First element value: " +a[0]);
            System.out.println("The finally statement is executed");
        }
    }
}
```

// Exception thrown :java.lang.ArrayIndexOutOfBoundsException: 3

First element value: 6

The finally statement is executed

Example2-Output?

```
public static void main(String args[]){
    try {
        System.out.println("I am in try");
        int x = 1/0;
    } catch (ArithmeticException e) {
        System.out.println("I am in catch");
        return;
    } finally{
        System.out.println("I am in finally");
    }
}
```

3-THROWING EXCEPTIONS

Exception Categories

To understand how exception handling works in Java, you need to understand the three categories of exceptions:

1. **Checked exceptions:**
   - A checked exception is an exception that is typically a user error or a problem that cannot be foreseen by the programmer.
   - For example, if a file is to be opened, but the file cannot be found, an exception occurs.
   - **These exceptions cannot simply be ignored at the time of compilation.**

Exception Categories

- A checked exception must be either caught or declared in a method where it can be thrown.
- For example, the java.io.IOException is a checked exception.
- Checked exception classes must be handled, otherwise the compiler gives an error.

Exception Categories

2. Runtime exceptions:

- Unchecked exceptions that are not required to be caught or declared, even if it is allowed to do so.
- So a method can throw a runtime exception, even if this method is not supposed to throw exceptions.
- For example, NullPointerException is an unchecked exception.
- **As opposed to checked exceptions, runtime exceptions are ignored at the time of compilation.**

Following is some examples of Java Unchecked RuntimeException.

- ArithmeticExceptionArithmetic error
  - such as divide-by-zero.
- ArrayIndexOutOfBoundsException
  - Array index is out-of-bounds.
- ArrayStoreException
  - Assignment to an array element of an incompatible type.
- ClassCastException
  - Invalid cast.
- IllegalArgumentException
  - Illegal argument used to invoke a method
- .....

Exception Categories

3. Errors

- These are not exceptions at all
- normally happen in case of severe failures, which are not handled by the java programs.
- Errors are typically ignored in your code because you can rarely do anything about an error.
- For example, if a stack overflow occurs, an error will arise.
- They are also ignored at the time of compilation.
- Example : JVM is out of Memory.
- Normally programs cannot recover from errors.

Throwing Exceptions: The *throw Keyword*

- Java allows you to throw exceptions (generate exceptions)

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);
        }
    }
}
```

```
//Exception caught:java.lang.RuntimeException: throw demo
```

The throws/throw Keywords

- A checked exception must be either catched or declared!!
- If a method does not handle a checked exception, the method must declare that it may throw the exception using **throws** keyword.
- The throws keyword appears at the end of a method's signature.
- You can throw an exception by either creating a newly instantiated exception or an exception that you just caught
- Try to understand the difference in throws and throw keywords.
  - throw *someThrowableObject within code*
  - declare that a method throws *someThrowableObject*
- The following method declares that it throws a RemoteException:

The throws/throw Keywords

```
import java.io.*;
public class className {
    public void deposit(double amount) throws RemoteException {
        // Method implementation
        throw new RemoteException();
    }
    //Remainder of class definition
}
```

Throwing Multiple Exceptions

- A method can declare that it throws more than one exception, in which case the exceptions are declared in a list separated by commas.

```
public class className {
    public void f() throws RemoteException,EmptyStackException{
        // Method implementation
    }
    //Remainder of class definition
}
```

Example

```
public void ioOperation(boolean isResourceAvailable) {
    if (!isResourceAvailable) {
        throw new IOException();
    }
}
```

Problem??

//You should add throws declaration

Solution to the Example

```
public void ioOperation(boolean isResourceAvailable) throws IOException {
    if (!isResourceAvailable) {
        throw new IOException();
    }
}
```

```
public void ioOperation(boolean isResourceAvailable) {
    try {
        if (!isResourceAvailable) {
            throw new IOException();
        }
    } catch(IOException e) {
    // Handle caught exceptions.
    }
}
```

Note the following

- A catch clause cannot exist without a try statement.
- It is not compulsory to have finally clauses when ever a try/catch block is present.
- The try block cannot be present without either catch clause or finally clause.
- From the above statement, you cannot have finally block alone on its own. below are the combinations allowed.
  - try catch finally
  - try catch
  - try finally
- No code can be written in between the try, catch, finally blocks.

Rules on Exc Handling

- A method is required to either catch or list all exceptions it might throw
  - Except for *Error or RuntimeException, or their* subclasses
- If a method may cause an exception to occur but does not catch it, then it must say so using the *throws keyword*
  - Applies to checked exceptions only
  - Syntax:

```
<type> <methodName> (<parameterList>) throws <exceptionList> {
    <methodBody>
}
```

Example: Method throwing an Exception

```
1  class ThrowingClass {
2     static void meth() throws ClassNotFoundException {
3        throw new ClassNotFoundException ("demo");
4     }
5  }
6  class ThrowsDemo {
7     public static void main(String args[]) {
8        try {
9           ThrowingClass.meth();
10       } catch (ClassNotFoundException e) {
11          System.out.println(e);
12       }
13    }
14 }
```

Exception Classes and Hierarchy

- Multiple catches should be ordered from subclass to superclass.

```
1  class MultipleCatchError {
2     public static void main(String args[]){
3        try {
4           int a = Integer.parseInt(args [0]);
5           int b = Integer.parseInt(args [1]);
6           System.out.println(a/b);
        - } catch (ArrayIndexOutOfBoundsException e) {...
      -     } catch (Exception ex) {...
1           }
2     }
3  }
```

Declaring your own Exception

- We can define our own Exception class as below:
- class MyException extends Exception{ }
- You just need to extend the Exception class to create your own Exception class.
- These are considered to be checked exceptions.
- The following InsufficientFundsException class is a user-defined exception that extends the Exception class, making it a checked exception.
- An exception class is like any other class, containing useful fields and methods.

- // File Name InsufficientFundsException.java import java.io.*;

```
public class InsufficientFundsException extends Exception {
        private double amount;
        public InsufficientFundsException(double amount) {
                this.amount = amount;
        }
        public double getAmount() {
                return amount;
        }
    }

public void withdraw(double balance) throws InsufficientFundsException {
    if(amount <= balance) {
        balance -= amount;
    } else {
        double needs = amount - balance;
        throw new InsufficientFundsException(needs);
    }
}
```