Error Handling

TEKNOTURF

Applications will encounter errors while executing

Reliable applications should handle errors as gracefully as possible

Errors

- Should be the “exception” and not the expected behavior
- Must be handled to create reliable applications
- Can occur as the result of application bugs
- Can occur because of factors beyond the control of the application
  - Databases become unreachable
  - Hard drive fails

Exceptions

TEKNOTURF

What is an Exception ?

- Exceptional event - typically an abnormality or error that occurs during runtime
- Cause the normal flow of a program to be disrupted

Examples

- Divide by zero errors
- Accessing the elements of an array beyond its range
- Invalid input
- Opening a non-existent file

Advantages of Exceptions

TEKNOTURF

Separating Error-Handling Code from "Regular" Code

Exceptions provide the means to separate the details of what to do when something out of the ordinary happens from the main logic of a program

Propagating Errors Up the Call Stack

If the readFile method is the fourth method in a series of nested method calls made by the main program: method1 calls method2, which calls method3, which finally calls readFile

Grouping and Differentiating Error Types

An example of a group of related exception classes in the Java platform are those defined in java.io — IOException and its descendants

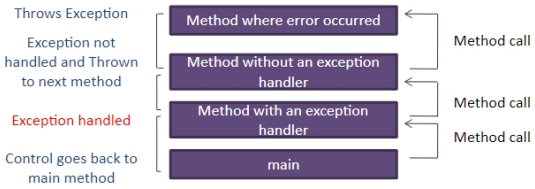
Exception

TEKNOTURF

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 has information about the error, its type and state of the program when the error occurred

The runtime system searches the call stack for the method that has the block of code that handles the exception



Exception

TEKNOTURF

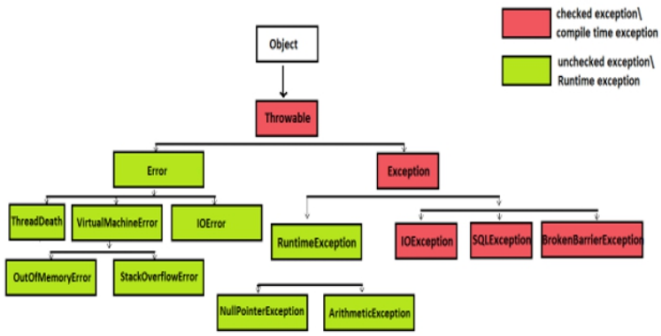
When an appropriate handler is found, the runtime system passes the exception to the handler

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.

Exception Hierarchy



The Error class

Errors

- Used by the Java run-time system to handle errors occurring in the run-time environment
- Generally beyond the control of user programs
- Examples
  - Out of memory errors
  - Hard disk crash

Checked and Unchecked Exception

Checked exception

- Java compiler checks if the program either catches or lists the occurring checked exception
- Checked exceptions should be explicitly handled or properly propagated
- If not, compiler error will occur

Unchecked exceptions

- Not subject to compile-time checking for exception handling
- Built-in unchecked exception classes
  - Error
  - RuntimeException and their subclasses
- Handling all these exceptions may make the program cluttered and may become difficult to manage

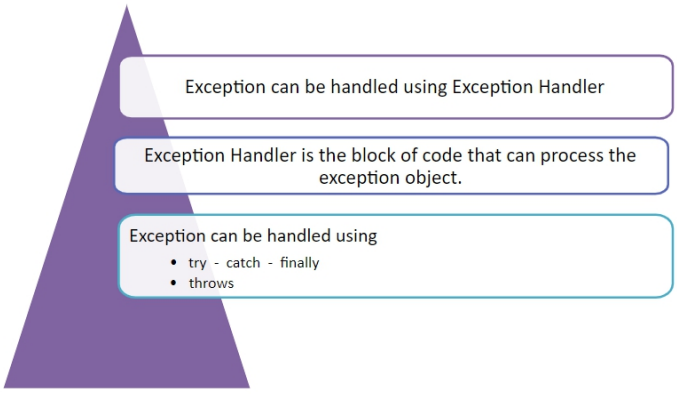
Exception Example

```
public class NumberFormatExceptionDemo{
   public static void main(String[] args) {
      int sum=0;
      for(String a : args) {
           sum +=Integer.parseInt(a);    }
      System.out.println("Sum is  "+ sum);
   }
}
```

Output :

```
java  NumberFormatExceptionDemo  8  12  1  4  2
Sum is 27

java  NumberFormatExceptionDemo  8  12  four 3.0  2
      Exception in thread "main" java.lang.NumberFormatException: For input string: "four"
      at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
      at java.lang.Integer.parseInt(Integer.java:580)
      at java.lang.Integer.parseInt(Integer.java:615)
      at model.NumberFormatExceptionDemo.main(NumberFormatExceptionDemo.java:7)
```

Catching and Handling Exceptions



try Block

try Block

The first step in constructing an exception handler is to enclose the code that might throw an exception within a try block

```
try {
    code
}
catch and finally blocks . . .
```

catch Block

Associate exception handlers with a try block by providing one or more catch blocks directly after the try block

Each catch block is an exception handler and handles the type of exception indicated by its argument

The argument type, declares the type of exception that the handler can handle and must be a class that inherits from the Throwable class

This catch block

- can contain the code to print the exception and also code to recover from the exception
- gets executed only if an exception object is thrown from its corresponding try block

No code can be between the try and the catch blocks

```
try {
} catch (ExceptionType name) {
} catch (ExceptionType name) {
}
```

The try-catch Example


```
public class NumberFormatExceptionDemo{
   public static void main(String[] args) {
      int sum=0;
      for(String a : args) {
           try{
               sum +=Integer.parseInt(a);
           }
           catch(NumberFormatException   e) {
               //a not an int
               System.err.println(a+" is not an integer ");
           }
      }
      System.out.println("Sum is  "+ sum);
   }
}
```

```
java  NumberFormatExceptionDemo  8  12  four 3.0  2

Output :
four is not an integer
3.0 is not an integer
Sum is  22
```

finally Block

The finally block always executes whether or not an exception occurs

It allows the programmer to avoid having cleanup code accidentally bypassed by a return, continue, or break.

Putting cleanup code in finally block is always a good practice

This block is optional

A try block can have multiple catch blocks , but only one finally block

```
finally {
    if (out != null) {
        System.out.println("Closing PrintWriter");
        out.close();
    } else {
        System.out.println("PrintWriter not open");
    }
}
```

finally Block

```
public class DivideByZeroException {

   public static void main(String[] args) {
      int result  = divide(100,0);       // Line 2
      System.out.println("result : "+result);
   }

   public static int divide(int totalSum, int totalNumber)
   {
      int quotient = -1;
      System.out.println("Computing Division.");
      try{
               quotient  = totalSum/totalNumber;
      }
      catch(ArithmeticException e)  {
          System.out.println("Exception : "+
                                          e.getMessage());
      }
```

```
finally  {
      if(quotient != -1)  {
          System.out.println("Finally Block Executes");
          System.out.println("Result : "+ quotient);
      }
      else {
          System.out.println("Finally Block Executes.
                                  Exception Occurred");
      }
   }
      return quotient;
   }
}
```

Output :
Computing Division.
Exception : / by zero
Finally Block Executes. Exception Occurred
result : -1

try with Multiple catch block

If a try block can throw multiple exceptions it can be handled using multiple catch blocks

If a try has multiple catch blocks, it should be ordered from subclass to super class

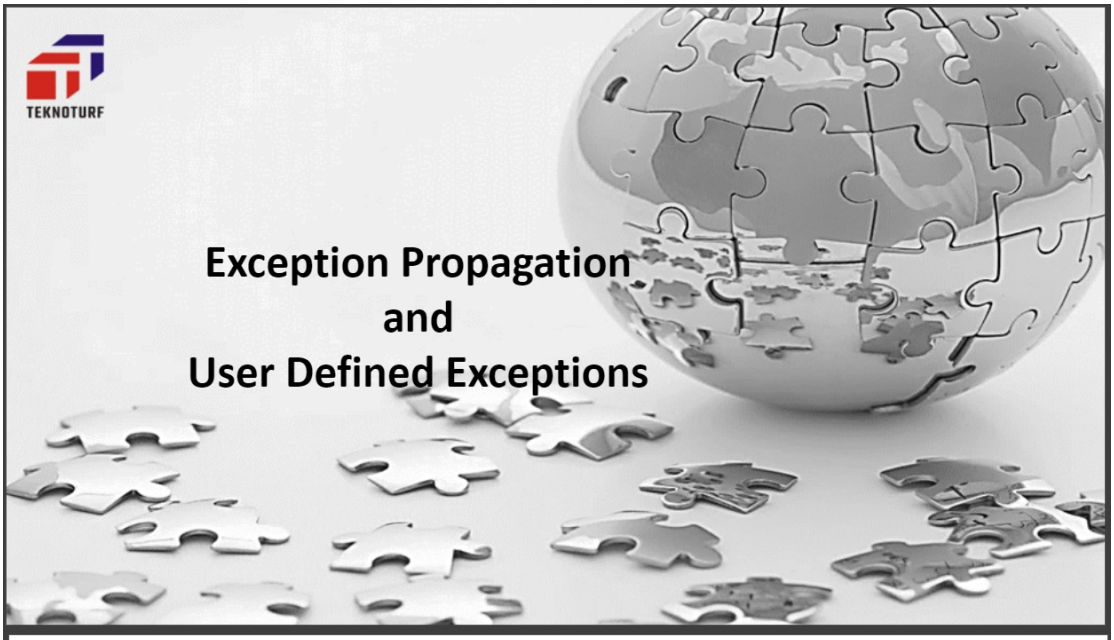
```
try {
  // code that might throw one or more exceptions

} catch (MyException e1) {
  // code to execute if a MyException exception is thrown

} catch (MyOtherException e2) {
  // code to execute if a MyOtherException exception is thrown

} catch (Exception e3) {
  // code to execute if any other exception is thrown
}
```

try with multiple catch Example

```
public class MultipleCatchExample  {
   public static void main (String args[]) {
      int array[]={20,10,30};
      int num1=15,num2=0;
      int result=0;
      try  {
         result = num1/num2;
         System.out.println("The result is" +result);
         for(int index=0;index<3;index++)  {
             System.out.println("The value of array are" +array[index]);
         }
      }
      catch (ArrayIndexOutOfBoundsException e)  {
          System.out.println("Error. Array is out of Bounds");
      }
      catch (ArithmeticException e)  {
          System.out.println ("Can't be divided by Zero");
      }
   }
}
```

Output :
Can't be divided by Zero

In the example, the first catch block is skipped and the second catch block handles the error



Nested try

TEKNOTURF

We can have nested try and catch blocks(a try block inside another try block)

If an inner try statement does not have a matching catch statement for a particular exception, the control is transferred to the next try statement's catch handlers

This continues until one of the catch statements succeeds, or until all of the nested try statements are done in

If none of the catch statements match, then the Java runtime system will handle the exception.

Nested try

TEKNOTURF

```
import java.io.*;
public class NestedTry{
 public static void main (String args[])throws IOException  {
   int num1=2,num2=0,res=0;
   try{
       FileInputStream fis=null;
       fis = new FileInputStream (new File (args[0]));
       try{
                res=num1/num2;
                System.out.println("The result is"+res);
       }
       catch(ArithmeticException e){
                System.out.println("divided by Zero");
       }
     }
```

```
catch (FileNotFoundException e){
       System.out.println("File not found!");
 }
catch(ArrayIndexOutOfBoundsException   e){
       System.out.println("Array index is Out
                        of bound! Argument required");
 }
 catch(Exception e){
   System.out.println("Error.."+e);
  }
 }
}
```

Nested try

If the above program is executed without giving any file name, the output will be "Array index is Out of bound! Argument required " (available in outer try)

If the above program is executed by giving a file name that does not exist, the output will be "File not found! " (available in outer try)

If the above program is executed by giving a file name that exists, the output will be "divided by Zero " (available in inner try).

Call Stack Mechanism

If an exception is not handled in the current try-catch block, it is thrown to the caller of that method

If the exception gets back to the main method and is not handled there, the program is terminated abnormally

A stack trace provides information on the execution history of the current thread and lists the names of the classes and methods that were called at the point when the exception occurred

The following code shows how to call the getStackTrace method on the exception object

```
catch (Exception cause) {
    StackTraceElement elements[] = cause.getStackTrace();
    for (int i = 0, n = elements.length; i < n; i++) {
        System.err.println(elements[i].getFileName()
            + ":" + elements[i].getLineNumber()
            + ">> "
            + elements[i].getMethodName() + "()");
    }
}
```

Rules for try-catch-finally

For each try block there can be zero or more catch blocks but only one finally block

The catch blocks and finally block must always appear in conjunction with a try block

A try block must be followed by either at least one catch block or one finally block

The order of the exception handlers in the catch block must be from the most specific exception

throw and throws usage

throw <exception reference>;

throw new ArithmeticException("Division attempt by 0");

```
Method() throws <ExceptionType_1>,..., <ExceptionType_n> {
        //statements
}
```

throws and throw - Example

```
public class DivideByZeroException {

  public static void main(String[] args) {
    try{
          int result  = divide(100,10);
          result  = divide(100,0);
    System.out.println("result : "+result);
    }
    catch(ArithmeticException e){
          System.out.println("Exception : "+
    e.getMessage());
    }
  }

  public static int divide(int totalSum, int
    totalNumber) throws   ArithmeticException
  {
    int quotient = -1;
    System.out.println("Computing Division.");
```

```
try{
      if(totalNumber == 0){
          throw new ArithmeticException("Division attempt
                                          by 0");
      }
      quotient  = totalSum/totalNumber;
}
 finally{
      if(quotient != -1){
      System.out.println("Finally Block Executes");
      System.out.println("Result : "+ quotient);
      }else{
      System.out.println("Finally Block Executes. Exception
                                  Occurred");
      }
}
 return quotient;
}
}
```

Handle or Declare Rule

Handle the exception by using the try-catch-finally block

Declare that the code causes an exception by using the throws clause

- void trouble() throws IOException { ... }
- void trouble() throws IOException, MyException { ... }

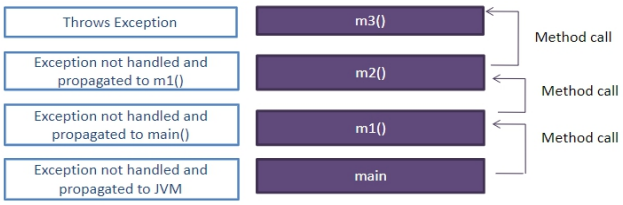
You do not need to declare runtime exceptions or errors

You can choose to handle runtime exceptions

Exception Propogation

Exception propagation is a way of propagating exception from a method to the previous method in the call stack until it is caught.

If uncaught thrown to JVM and program gets terminated



Exception Propogation

```
public class ExceptionPropogationDemo
{
   public static void main(String a[])  {
     m1(); //Line 4
   }
   public static void m1() {
     m2();  //Line 7
   }
   public static void m2() {
     m3();  //Line 10
   }
   public static void m3() {
     throw new ArithmeticException(); //Line13
   }
}
```

Output :

```
Exception in thread "main" java.lang.ArithmeticException
at
ExceptionPropogationDemo.m3(ExceptionPropogationDemo.java:13)
at
ExceptionPropogationDemo.m2(ExceptionPropogationDemo.java:10)
at ExceptionPropogationDemo.m1(ExceptionPropogationDemo.java:7)
at
ExceptionPropogationDemo.main(ExceptionPropogationDemo.java:4)
```

Exception Propogation

```
public class ExceptionPropogationDemo
{
    public static void main(String a[]) throws IOException  {
          m1();
    }
    public static void m1() throws IOException {
          m2();
    }
    public static void m2() throws IOException {
          m3();
    }
    public static void m3() throws IOException {
          throw new IOException();
    }
}
```

```
Exception in thread "main" java.io.IOException
at ExceptionPropogationDemo.m3(ExceptionPropogationDemo.java:15)
at ExceptionPropogationDemo.m2(ExceptionPropogationDemo.java:12)
at ExceptionPropogationDemo.m1(ExceptionPropogationDemo.java:9)
at ExceptionPropogationDemo.main(ExceptionPropogationDemo.java:6)
```

Method Overriding and Exceptions

TEKNOTURF

The overriding method can throw

- No exceptions
- One or more of the exceptions thrown by the overridden method
- One or more subclasses of the exceptions thrown by the overridden method

The overriding method cannot throw

- Additional exceptions not thrown by the overridden method
- Super classes of the exceptions thrown by the overridden method

Method Overriding and Exceptions

TEKNOTURF

Example 1 :

```
class Parent  {
  public void method1() throws Exception
  {
      System.out.println("Parent");
  }
}
public class Child extends Parent  {
  public void method1() throws Exception
  {
      System.out.println("Child");

  }
  public static void main(String a[]) throws Exception
  {
       Parent p=new Child();
      p.method1();
   }
}
```

Example 2 :

```
class Parent {
  public void method1() throws Exception
  {
      System.out.println("Parent");
  }
}
public class Child extends Parent  {
 public void method1() throws ArithmeticException
  {
      System.out.println("Child");
  }
  public static void main(String a[]) throws Exception
  {
       Parent p=new Child();
      p.method1();
   }
}
```

Method Overriding and Exceptions

TEKNOTURF

Example 1 :

```
class Parent  {
  public void method1() throws Exception
  {
      System.out.println("Parent");
  }
}
public class Child extends Parent  {
  public void method1()
  {
      System.out.println("Child");

  }
  public static void main(String a[]) throws Exception
  {
       Parent p=new Child();
      p.method1();
   }
}
```

Example 2 :

```
class Parent {
  public void method1() throws ArithmeticException
  {
      System.out.println("Parent");
  }
}
public class Child extends Parent  {
 public void method1()  throws Exception
  {
      System.out.println("Child");
  }
  public static void main(String a[]) throws Exception
  {
       Parent p=new Child();
      p.method1();
   }
}
```

Compile Time Error

User Defined Exception

Though Java provides an extensive set of in-built exceptions, there are cases in which we may need to define our own exceptions in order to handle the various application specific errors.

While defining a user defined exception, we need to extend the Exception class.

User Defined Exceptions - Example


```
class InvalidAgeException extends Exception  {
   public InvalidAgeException(String message)
   {
      super(message);
   }
}

public class CustomException {
   public static void validateAge(int age) throws
                                    InvalidAgeException  {
       if(age < 18)
       {
            throw new InvalidAgeException("Not a
                                        valid Age to
     vote");
       }
      else {
        System.out.println("Eligible to vote");
      }
   }
   public static void main(String arg[])  {
      try  {
        validateAge(15);
      }
      catch(InvalidAgeException e)  {
        System.out.println(e.getMessage());
      }
   }
}
```

Try with multiple catch

Java 7, introduced the ability to catch multiple exceptions in the same catch block, also known as *multi-catch*.

```
public static void main(String[] args) {
   try {
       Path path = Paths.get("dolphinsBorn.txt");
       String text = new String(Files.readAllBytes(path));
       LocalDate date = LocalDate.parse(text);
       System.out.println(date);
   } catch (DateTimeParseException | IOException e) {
       e.printStackTrace();
       throw new RuntimeException(e);
   } }
```

```
//Compile Time Error
catch(Exception1 e | Exception2 e | Exception3 e)

//Works correctly

catch(Exception1 | Exception2 | Exception3 e)
```

Try with multi-catch

Java intends multi-catch to be used for exceptions that aren't related, and it prevents from specifying redundant types in a multi-catch.

```
try {
    throw new IOException();
}
catch (FileNotFoundException |  IOException e) { }
// DOES NOT COMPILE
```

FileNotFoundException is a subclass of IOException.
Specifying it in the multi-catch is redundant, and the compiler gives a message as :
The exception FileNotFoundException is already caught by the alternative IOException

Try with resources

When we work with other resources, say file or database, its essential to close the resources once their usage is over

Assume, operation of reading data from one file and write to another file

The code will be as follows

```
public void writeData(Path path1, Path path2)
                throws  IOException {
   BufferedReader in = null;
   BufferedWriter out = null;
   try {
      in = Files.newBufferedReader(path1);
      out = Files.newBufferedWriter(path2);
      out.write(in.readLine());
   } finally {
       if (in != null) in.close();   if (out != null)
                                        out.close();
} }
```

Resources used

Resources closed

Try with resources

```
public void writeData(Path path1, Path path2) throws  IOException {
    try (BufferedReader in = Files.newBufferedReader(path1);
         BufferedWriter out = Files.newBufferedWriter(path2);)   {
           out.write(in.readLine());
   }
}
```

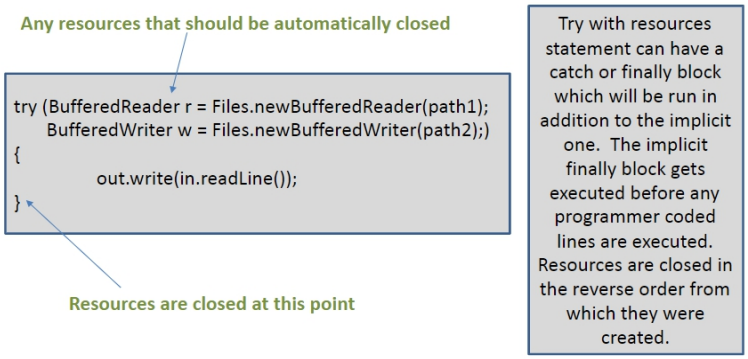
There is no longer code just to close resources.

The new *try-with-resources* statement automatically closes all resources opened in the try clause.

This feature is also known as *automatic resource management*, because Java automatically takes care of the closing.

There is no finally block in the try-with-resources code. It exists implicitly.

Try with resources



Try with resources statement can have a catch or finally block which will be run in addition to the implicit one. The implicit finally block gets executed before any programmer coded lines are executed. Resources are closed in the reverse order from which they were created.

Try with resources

```
try (Scanner s = new
Scanner(System.in)) {
            s.nextLine();
} catch(Exception e) {
      s.nextInt(); // DOES NOT COMPILE
  } finally{
     s.nextInt(); // DOES NOT COMPILE
 }
```

The problem here, is that the Scanner has gone out of scope at the end of the try clause.

```
public class  Sample {
   public static void main(String
                        a[])  {
  try (Sample s=new Sample();) {
      System.out.println(s);
   }
  }
}
```

Java doesn't allow this, because it doesn't know how to close Sample. It throws an error *"The resource type Sample does not implement java.lang.AutoCloseable"*

To use a class in try with resources, , Java requires it to implement an interface, AutoCloseable.

Try with resources

```
public class Sample implements AutoCloseable {
   public void close() {
        System.out.println("Close gate");
   }
   public static void main(String[] args) {
       try (Sample s = new Sample() ) {
            System.out.println("Hello");
       }
   }
}
```

Auto Closeable is an interface with an abstract method
public void close() throws Exception;