



# EXCEPTION HANDLING

©2015 Manipal Global Education



## OBJECTIVES

### *Exception Handling*



## LEARNING OBJECTIVES

At the end of this lesson, you will be able to:

- Understand Exception
- Differentiate between Checked and Unchecked exception
- Handle Exceptions using try catch.
- Declare Exceptions using throws clause
- Create Custom Exceptions





Refer package **com.mgait.exceptions** in the provided code base for demo programs on the topics covered in this presentation



## CONCEPT *Exception*



## WHAT IS AN EXCEPTION

- An exception is an error condition that occurs during the execution of program
- Disrupts the normal flow of the program's instructions
- Halt's the program execution, if not handled
  
- Examples
  - Running out of memory
  - Accessing an invalid array index
  - Trying to open a file that does not exist
  - Division by Zero

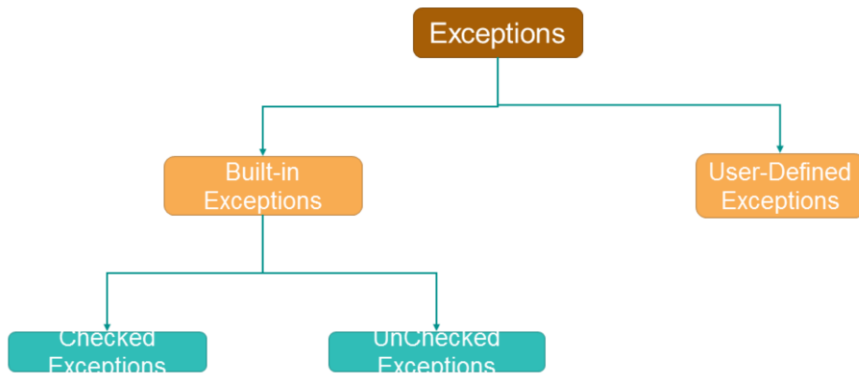


## THROWING AN EXCEPTION

- When an exception occurs in a method
  - The method creates an exception object based on the type of error and hands it over to runtime system – known as *throwing an exception*
- The exception object created contains
  - information about the error and the type of error
  - where the error occurred
- The runtime system attempts to find a handler which can take appropriate action
- If no handler is found, the runtime system terminates the program



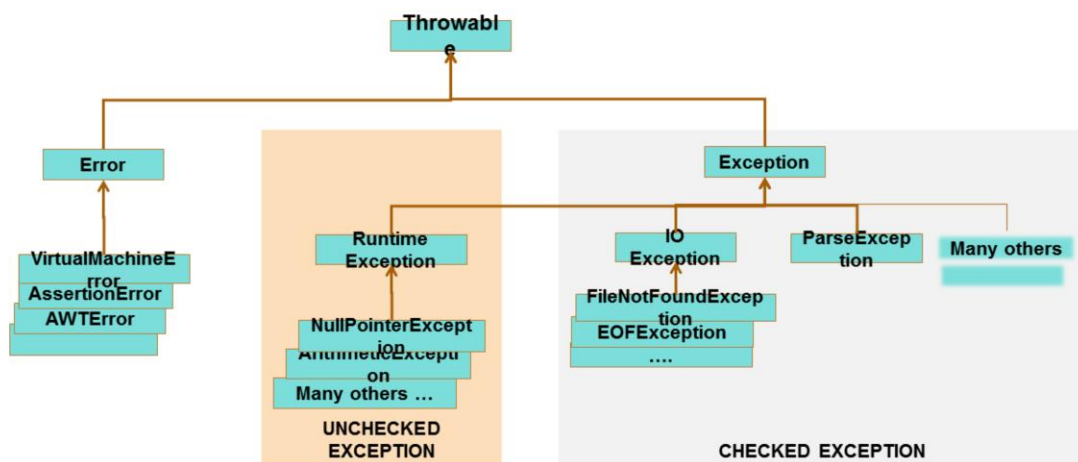
## Categories of Exceptions







## JAVA EXCEPTION HIERARCHY





## UNCHECKED EXCEPTIONS

- Exceptions that are not checked during compile time
- Subclasses of RuntimeException and Error
- Examples
  - ArrayIndexOutOfBoundsException
  - NullPointerException
  - ClassCastException
  - ArithmeticException
  - NumberFormatException
  - IllegalArgumentException
  - StackOverflowError
- Not supposed to be handled by programmer
- Code leading to unchecked exceptions needs to be debugged



## RUNTIME EXCEPTIONS

- `ArrayIndexOutOfBoundsException` : An out-of-bounds array access

```
int[] array = new int[]{1,2,3,4};  
System.out.print(array[4]);
```

- `NumberFormatException`

```
int i = Integer.parseInt("12.25")
```

- `ArithmeticException` : Divide by Zero

```
int a = 5; int b = 3; int c = 2;  
int result = a / (a - (b +c)) ;
```



## RUNTIME EXCEPTIONS

- ClassCastException : illegal Casting of Objects

```
Number num = new Integer(10);  
Float f = (Float) num;
```

- NullPointerException : Null Pointer access

```
String str = null;  
System.out.print(str.length());
```



## CHECKED EXCEPTIONS

- Exception that are checked during compile time
- If some code within a method might throws a checked exception, then the method
  - Must handle the exception
  - Or declare the exception
- ★  
Java enforces handling of checked exceptions
- Classes other than RuntimeException and Error



## CHECKED EXCEPTIONS Examples

- ParseException
  - Parsing a String containing invalid Date
- SQLException
  - Opening a connection to database with invalid IP address
- ClassNotFoundException
  - Trying to load a class which is not present in the class path
- IOException
  - Signal an exception during I/O operation
  - FileNotFoundException
    - Sub class of IOException
    - Trying to open a file that does not exist

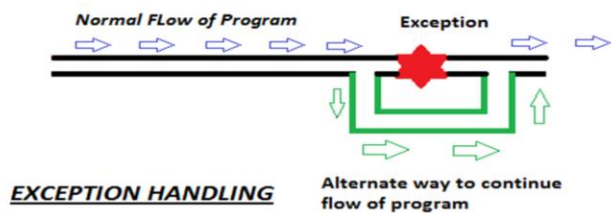


### Commonly used methods of Throwable

- Following are some methods of Throwable class to retrieve information about exception
  - getMessage()
    - returns the detail message string
  - printStackTrace()
    - prints the most recently entered method first and continues down the call stack



## WHY HANDLE EXCEPTIONS







## HANDLING EXCEPTION

- Exceptions can be handled using a try-catch block

```
try
{
    // Code that might throw some exception
}
catch(Exception e1)
{
    // Code to handle the exception of type e1
}
```

- try block should contain the code which might throw an exception
- catch block contains the code for handling and exception, if exception occurs in try block



## HANDLING EXCEPTION

DEMO  
Class : TryCatch\

```
try
{
    statement1;
    statement2;
}
catch (Exception e1)
{
    statement3;
}
```

- If statement1 in try block throws an exception
  - Execution of statement2 is skipped
  - Control is transferred to catch block
- If try block executes successfully without any exception
  - catch clause is skipped

★ if exception thrown in try block does not match with the exception in the catch block, then runtime system halts the program execution



## USING FINALLY BLOCK

DEMO  
Class : Finally

- finally block is used to execute code that must run regardless of an exception occurrence
  - Always executes when the try or catch block completes execution
  - Used as the clean up block to free up resources used in try block, like closing connections

```
try {  
    // Code that must be executed  
}  
catch(Exception e1) {  
    // code that handles exception e1  
}  
finally {  
    // code to release any resource  
    allocated in the try clause.  
}
```

```
try {  
    // Code that must be executed  
}  
finally {  
    // code to release any resource  
    allocated in the try clause.  
}
```



## CATCHING MULTIPLE EXCEPTIONS

- A try block can be followed by multiple catch blocks

```
try {  
    // Code that might throw an exception  
}  
catch(NullPointerException e) {  
    // Handle the NullPointerException  
}  
catch(ParseException e) {  
    // Handle the ParseException  
}
```



## CATCHING MULTIPLE EXCEPTIONS - Rules

- More than one type of Exception can be handled in a single catch block, if super class exception is caught.
- Catch block for IOException will catch IOException and all of its subclasses like FileNotFoundException, EOFException

```
try {  
    // Code that might throw an exception  
}  
catch(IOException e) {  
    // Handle the IOException  
}
```



## CATCHING MULTIPLE EXCEPTIONS - Rules

DEMO  
Class : MultipleCatch

- If catch block for subclass exceptions are included , they should be coded before the catch block for super class exception

```
try {  
    // Code that might throw an exception  
}  
catch(FileNotFoundException e) {  
    // Handle the FileNotFoundException  
}  
catch(EOFException e) {  
    // Handle the EOF Exception  
}  
catch(IOException e) {  
    // Handle the IOException  
}
```

- FileNotFoundException and EOFException are subclasses of IOException



## CATCHING MULTIPLE EXCEPTIONS – Java 7

- In Java SE 7 and later, a single catch block can handle more than one type of exception separated with a vertical bar (|).

```
try {  
    // Code that might throw an exception  
}  
catch (IOException | ArithmeticException ex) {  
    // Handle IOException or ArithmeticException  
}
```



### NESTED TRY CATCH BLOCK

- Java allows nesting of try and catch blocks
- If an inner try statement does not have a matching catch statement then
  - Control is transferred to the next try statement's catch handlers
- If none of the catch statements match, then the Java run-time system will handle the exception

```
try {  
    try {  
        //code to be executed within the try block  
    }  
    catch (Exception1 e1)  
    {    s1 //statement to handle the exception }  
}  
catch (Exception2 e2){  
    s2 //statements to handle the exception  
}
```

- If exception e1 is thrown in inner try, inner catch handles the exception
- If exception e2 is thrown in inner try, outer catch handles the exception





\_\_\_\_\_ is a superclass of all exception classes

- ☐ Exception
- ☐ Throwable
- ☐ RuntimeException
- ☐ IOException





Which class should X represent in the below code?

```
public class Test{  
    public static void main(String args[]){  
        try{  
            int a = Integer.parseInt("four");  
        } catch(X x){ }  
    }  
}
```

- ☐ IllegalStateException
- ☐ NumberFormatException
- ☐ ClassCastException
- ☐ ArrayIndexOutOfBoundsException
- ☐ None of these





Which of the below statement is/are true about Error?

- A. An Error is a subclass of Throwable.
- B. An Error is a subclass of Exception.
- C. Error indicates serious problems that a reasonable application should not try to catch.
- D. An Error is a subclass of IOException.
- ☐ A and D
- ☒ **A and B**
- ☐ B and C
- ☐ B and D
- ☐ A and C





What is the output of this program?

```
public class Foo {  
    public static void main(String[] args) {  
        try{  
            return;  
        }finally {  
            System.out.println( "Finally" );  
        } } }  
}
```

1. Finally
2. Compilation fails
3. The code runs with no Output
4. An exception is thrown at runtime





Which block gets executed compulsorily whether exception occurs or not?

- ☐ finally
- ☐ throw
- ☐ throws
- ☐ catch





What will be the output of the program?

```
public class MyProgram {  
    public static void main(String args[]) {  
        try {  
            System.out.print("Hello world ");  
        } finally {  
            System.out.println("Finally executing ");  
        }  
    }  
}
```

- ☐ The program will not compile because no exceptions are specified
- ☐ The program will not compile because no catch clauses are specified
- ☐ Hello world
- ☐ Hello world Finally executing





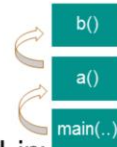
## CONCEPT

### *Declaring Exceptions*



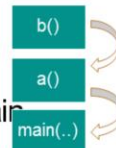
## PROPAGATING EXCEPTIONS

- A method can throw an exception back to its calling method
- Consider the call stack of a program to understand the method invocation



CALL STACK: Method  
Invocation

- In case an exception occurs and it is not handled, the method throws the exception to its calling method
- Exception occurring in method b() is thrown to a(), and a() to main



CALL STACK: Exception  
propagation





## DECLARING EXCEPTIONS

DEMO  
Class : DeclareExcp,  
TestDeclare

- A method can declare that it can propagate an exception to its calling method
- If a method doesn't handle a checked exception, it must declare it using

```
public String formatDate(String dateString) throws ParseException{  
    //code that can throw a ParseException  
}
```

- Exception propagated must be handled by the calling method using try-catch

OR

declared using throws clause

33

➤ Finally the exception will have to be handled by the main() method else

©2015 Manipal Global Education



## OVERRIDING RULES

```
class Super{
    public void method1() throws IOException{
        //code that can throw a IOException
    }
}

class Sub extends Super{
    public void method1() throws _____{
        //some code
    }
}
```

- When a sub class overrides a method that declares an exception, the sub class method can
  - Throw the same exception as superclass method throws
  - Throw a sub class of the exception that superclass method throws
  - **Throw no Exception**



## USING THE THROW STATEMENT

- throw keyword is used to explicitly throw an exception
- Typically useful for throwing user defined exceptions

```
public double divide(int dividend, int divisor) throws ArithmeticException {  
    if(divisor == 0) {  
        throw new ArithmeticException("Divide by 0 error");  
    }  
    return dividend / divisor;  
}
```

- throw can also be used to throw any predefined exceptions in java



## RE THROWING EXCEPTIONS

- Exception caught in catch clause can be thrown again using throw keyword.
- Re-thrown exception must be handled in the program, otherwise program will terminate abruptly.

```
class InvalidDataException extends Exception{}  
class IllegalDataException extends Exception{}  
  
public void validate(String test) throws Exception{  
    try{  
        //some code ...  
        if(condition1) throw new InvalidDataException();  
  
        if(condition2) throw new IllegalDataException();  
        //some code ...  
    }catch(Exception e){  
        throw e;  
    }  
}
```



**Which keyword is used to explicitly throw an exception?**

- ☐ try
- ☐ catch
- ☐ throws
- ☐ throw





**Which keyword is used in a method declaration to specify the exception the method might throw?**

- ☐ catch
- ☐ throw
- ☐ throws
- ☐ finally





## CONCEPT

*Custom  
exceptions*



## CUSTOM EXCEPTIONS

DEMO  
Class :  
NegativeAgeException, CustomTe

- Program can run into a problem that is not adequately described by any of the java exception classes like any business validations
- A Custom Exception class can be created by extending any of the standard exception classes

```
class InvalidDataException extends IOException{  
    public InvalidDataException() {  
        super();  
    }  
    public InvalidDataException(String info){  
        super(info);  
    }  
}
```

- This custom exception can be thrown and caught like any other exception





## 1. To create our own exception class , we have to

- ☐ Extend exception class
- ☐ Create our own try and catch block
- ☐ use finally block
- ☐ Use throws keyword





What will be the output of the program?

```
class Exc0 extends Exception { }  
class Exc1 extends Exc0 { }          /* Line 2 */  
public class Test {  
    public static void main(String args[]) {  
        try {  
            throw new Exc1();          /* Line 9 */  
        } catch (Exc0 e0) {            /* Line 11 */  
            System.out.println("Ex0 caught");  
        } catch (Exception e) {  
            System.out.println("exception caught");  
        }  
    }  
}
```

- ☐ Ex0 caught
- ☐ exception caught
- ☐ Compilation fails because of an error at line 2.
- ☐ Compilation fails because of an error at line 9.



## CONCEPT *Assertions*



## ASSERTIONS

- It is common practice to have debug statements in programs to check whether conditions are being satisfied

```
private double calcInterest(double balance){  
    if(balance > 0)  
        double interest = balance * interestRate * tenure * 0.01);  
        return interest;  
    } else  
        System.out.println("Interest cannot be calculated on Zero Balance");  
    }  
}
```

- Assertions helps to test assumptions during development
- Assertion code can be disabled when the program is deployed
- No overhead of debugging code to track down and remove



## ASSERTIONS

- Rewritten calcInterest() using assertions to validate that the balance is not zero

```
private double calcInterest(double balance){  
    assert(balance > 0) ;  
    double interest = balance * interestRate * tenure * 0.01;  
    return interest;  
}
```

- Assertions helps to keep code cleaner and tighter
- Assertions are inactive unless specifically "turned on" (enabled)

```
private double calcInterest(double balance){  
    double interest = balance * interestRate * tenure * 0.01;  
    return interest;  
}
```



## ASSERTIONS

- Assertions can be defined in two ways :
  - `assert(exp1);`
  - `assert(exp1) : exp2;`
    - `exp1` has to return a boolean value
    - `exp2` can be anything that results in a value

```
assert(x > y)
```

```
assert(x > y) : "y is " + y + " x is " + x;
```

- Enabling assertions at runtime
  - `java -ea TestClass`
  - `java -enableassertions TestClass`



## References

- Refer following demo videos on EduNxt
  - M5I1I3 Demonstration Of Unchecked And Checked Exceptions – Demo
  - M5I1I5 Demonstrating Exception Handling – Demo
  - M5I1I7 Using Multiple Catches With One Try Block – Demo
  - M5I2I2 Using Finally – Demo
  - M5I3I2 Throwing Exceptions – Demo
  - M5I4I2 Creating Custom Exception Classes - Demo





## SUMMARY

### *Exception Handling*





## SUMMARY



In this lesson, you've learned to:

- Handle Exception using try catch
- Use multiple catch clause
- Use finally block
- Declare Exception using throws
- Create custom exceptions