# Exceptions

## CENG522 – Advanced Object Oriented Programming

Dr.Serdar ARSLAN

# Exception

- An exception (or exceptional event) is a problem that arises during the execution of a program.
- When an **Exception** occurs;
  - the normal flow of the program is disrupted
  - the program/Application terminates abnormally.
- An exception can occur for many different reasons.
  - A user has entered an invalid data.
  - A file that needs to be opened cannot be found.
  - A network connection has been lost in the middle of communications
  - The JVM has run out of memory.

# Handling Exceptions

- An exception is an object that is generated as the result of an error or an unexpected event.
- Exception are said to have been “thrown.”
- It is the programmers responsibility to write code that detects and handles exceptions.
- Unhandled exceptions will crash a program.
- Java allows you to create exception handlers.

# Handling Exceptions

- An *exception handler* is a section of code that gracefully responds to exceptions.
- The process of intercepting and responding to exceptions is called *exception handling*.
- The *default exception handler* deals with unhandled exceptions.
- The default exception handler prints an error message and crashes the program.

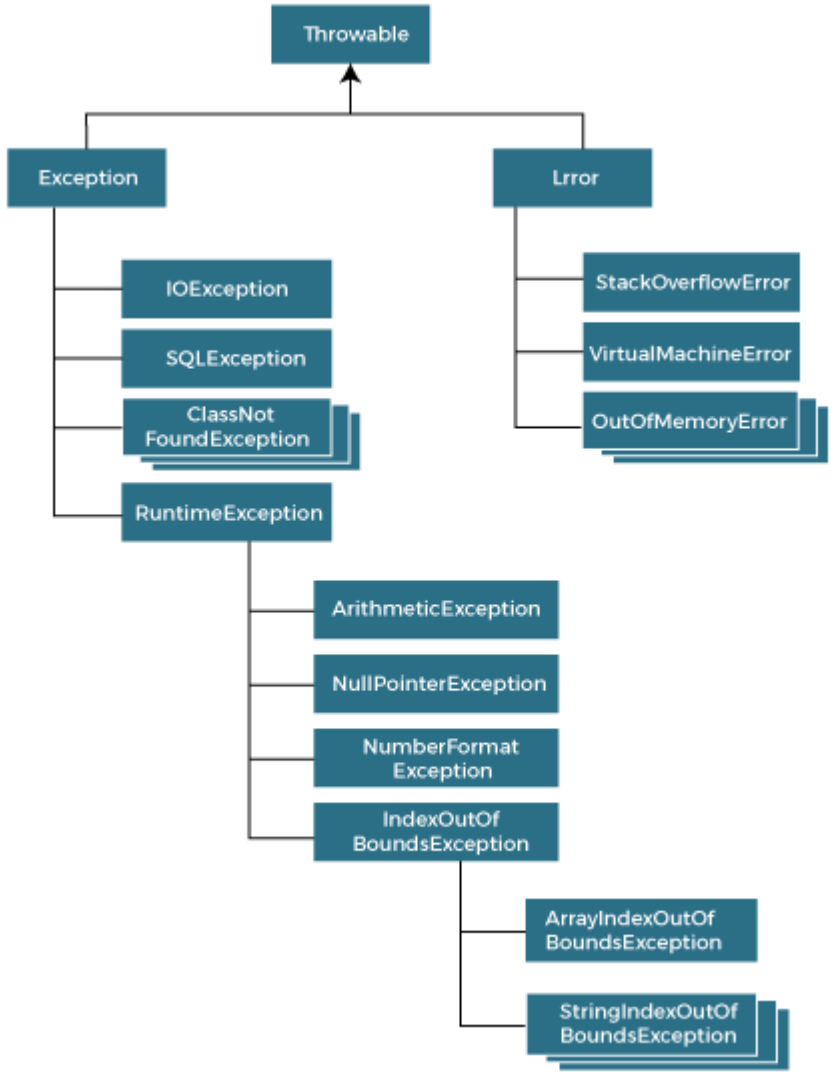
# Exception Classes

- An exception is an object.
- Exception objects are created from classes in the Java API hierarchy of exception classes.
- All of the exception classes in the hierarchy are derived from the `Throwable` class.
- `Error` and `Exception` are derived from the `Throwable` class.

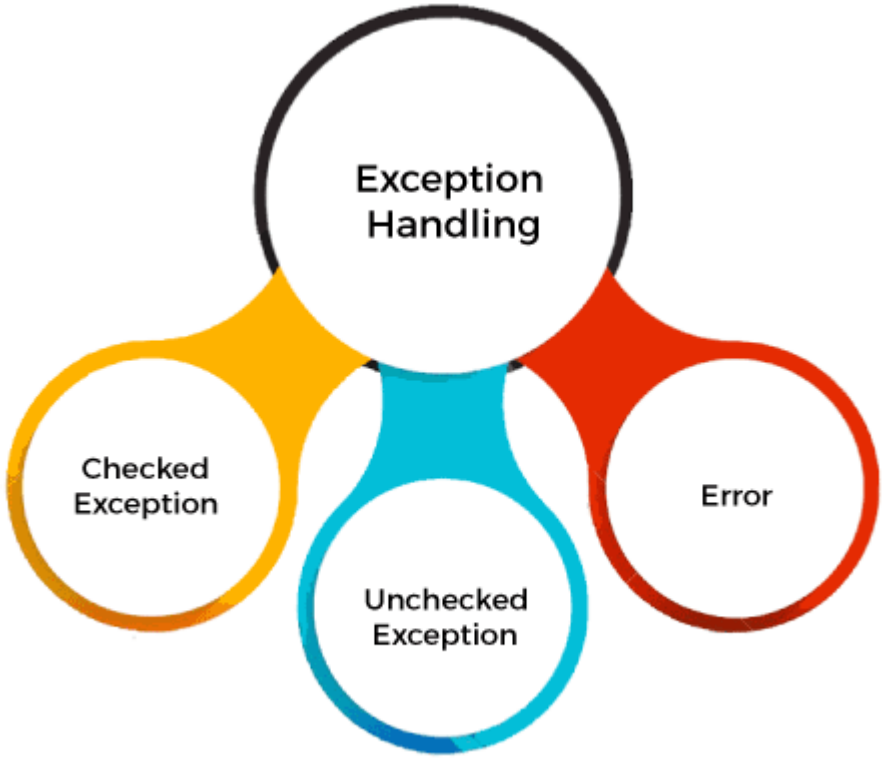
# Exception Classes

- Classes that are derived from `Error`:
  - are for exceptions that are thrown when critical errors occur. (i.e.)
    - an internal error in the Java Virtual Machine, or
    - running out of memory.
- Applications should not try to handle these errors because they are the result of a serious condition.
- Programmers should handle the exceptions that are instances of classes that are derived from the `Exception` class.

# Exception Class Hierarchy



# Types of Java Exceptions



# Java Exception Keywords

| Keyword | Description |
| --- | --- |
| try | The "try" keyword is used to specify a block where we should place an exception code. It means we can't use try block alone. The try block must be followed by either catch or finally. |
| catch | The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later. |
| finally | The "finally" block is used to execute the necessary code of the program. It is executed whether an exception is handled or not. |
| throw | The "throw" keyword is used to throw an exception. |
| throws | The "throws" keyword is used to declare exceptions. It specifies that there may occur an exception in the method. It doesn't throw an exception. It is always used with method signature. |

# Handling Exceptions

- To handle an exception, you use a *try* statement.

```
try
{
   (try block statements...)
}
catch (ExceptionType ParameterName)
{
   (catch block statements...)
}
```

- First the keyword `try` indicates a block of code will be attempted (the curly braces are required).
- This block of code is known as a *try block*.

# Handling Exceptions

- A *try block* is:
  - one or more statements that are executed, and
  - can potentially throw an exception.
- The application will not halt if the try block throws an exception.
- After the try block, a `catch` clause appears.

# Handling Exceptions

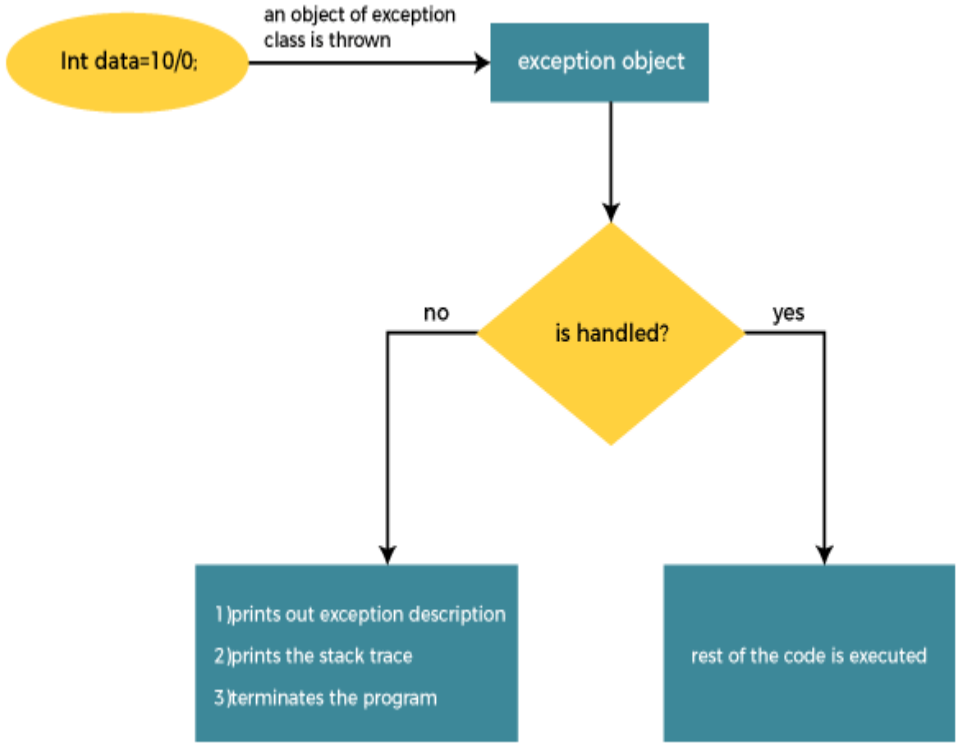
- A catch clause begins with the key word `catch`:

```
catch (ExceptionType ParameterName)
```

  - *`ExceptionType`* is the name of an exception class and
  - *`ParameterName`* is a variable name which will reference the exception object if the code in the try block throws an exception.

- The code that immediately follows the catch clause is known as a *catch block* (the curly braces are required).
- The code in the catch block is executed if the try block throws an exception.

# Java try-catch block




```java
public class TryCatchExample1 {

   public static void main(String[] args) {

      int data=50/0; //may throw exception

      System.out.println("rest of the code");

   }

}
```

**Output:**

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
```

# Java try-catch block

```java
public class TryCatchExample1 {

    public static void main(String[] args) {

        int data=50/0; //may throw exception

        System.out.println("rest of the code");

    }

}
```

**Output:**

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
```

```java
public class TryCatchExample2 {

    public static void main(String[] args) {
        try
        {
        int data=50/0; //may throw exception
        }
            //handling the exception
        catch(ArithmeticException e)
        {
            System.out.println(e);
        }
        System.out.println("rest of the code");
    }

}
```

**Output:**

```
java.lang.ArithmeticException: / by zero
rest of the code
```

# Java try-catch block

```
public class TryCatchExample2 {

   public static void main(String[] args) {
      try
      {
      int data=50/0; //may throw exception
      }
         //handling the exception
      catch(ArithmeticException e)
      {
         System.out.println(e);
      }
      System.out.println("rest of the code");
   }

}
```

**Output:**

```
java.lang.ArithmeticException: / by zero
rest of the code
```

```
public class TryCatchExample4 {

   public static void main(String[] args) {
      try
      {
      int data=50/0; //may throw exception
      }
         // handling the exception by using Exception class
      catch(Exception e)
      {
         System.out.println(e);
      }
      System.out.println("rest of the code");
   }

}
```

**Output:**

```
java.lang.ArithmeticException: / by zero
rest of the code
```

# Java try-catch block

```
public class TryCatchExample2 {

   public static void main(String[] args) {
      try
      {
      int data=50/0; //may throw exception
      }
         //handling the exception
      catch(ArithmeticException e)
      {
         System.out.println(e);
      }
      System.out.println("rest of the code");
   }

}
```

**Output:**

```
java.lang.ArithmeticException: / by zero
rest of the code
```

```
public class TryCatchExample5 {

   public static void main(String[] args) {
      try
      {
      int data=50/0; //may throw exception
      }
         // handling the exception
      catch(Exception e)
      {
            // displaying the custom message
         System.out.println("Can't divided by zero");
      }
   }

}
```

**Output:**

```
Can't divided by zero
```

# Java try-catch block

```
public class TryCatchExample7 {

   public static void main(String[] args) {

      try
      {
      int data1=50/0; //may throw exception

      }
         // handling the exception
      catch(Exception e)
      {

      int data2=50/0;

      }
   System.out.println("rest of the code");
   }
}
```

**Output:**

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
```

# Java try-catch block

```java
public class TryCatchExample9 {

   public static void main(String[] args) {
      try
      {
      int arr[]= {1,3,5,7};
      System.out.println(arr[10]); //may throw exception
      }
         // handling the array exception
      catch(ArrayIndexOutOfBoundsException e)
      {
         System.out.println(e);
      }
      System.out.println("rest of the code");
   }

}
```

**Output:**

```
java.lang.ArrayIndexOutOfBoundsException: 10
rest of the code
```

```java
public class TryCatchExample8 {

   public static void main(String[] args) {
      try
      {
      int data=50/0; //may throw exception

      }

      catch(ArrayIndexOutOfBoundsException e)
      {
         System.out.println(e);
      }
      System.out.println("rest of the code");
   }

}
```

**Output:**

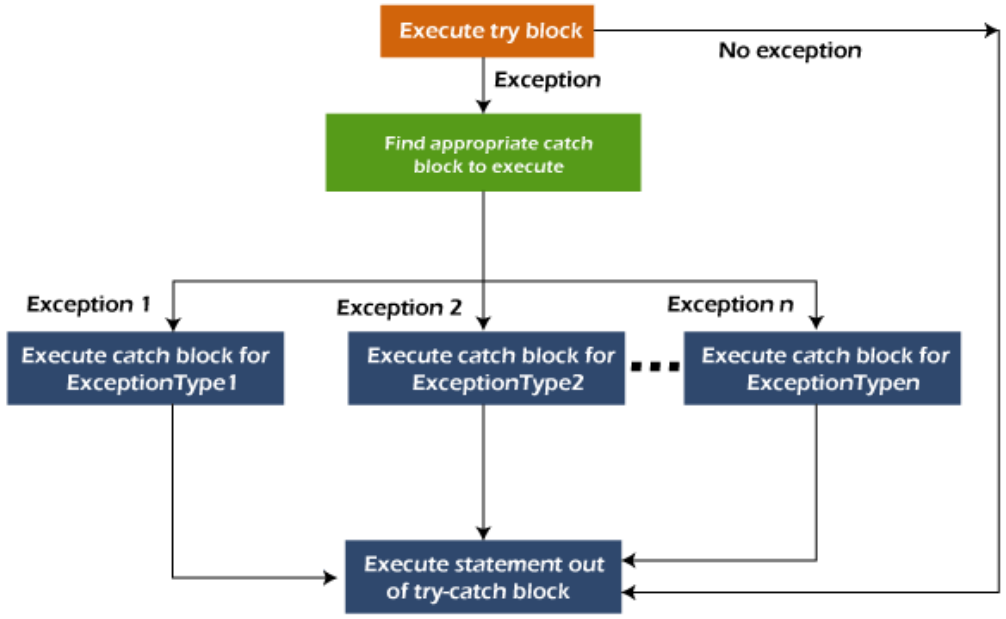
```
Exception in thread "main" java.lang.ArithmeticException: / by zero
```

# Handling Multiple Exceptions

- The code in the try block may be capable of throwing more than one type of exception.
- A `catch` clause needs to be written for each type of exception that could potentially be thrown.
- The JVM will run the first compatible `catch` clause found.
- The `catch` clauses must be listed from most specific to most general.

# Catch Multiple Exceptions

- At a time only one exception occurs and at a time only one catch block is executed.
- All catch blocks must be ordered from most specific to most general, i.e. catch for ArithmeticException must come before catch for Exception.



# Catch Multiple Exceptions

```
public static void main(String[] args) {

    try{
        int a[]=new int[5];
        a[5]=30/0;
       }
       catch(ArithmeticException e)
         {
          System.out.println("Arithmetic Exception occurs");
         }
       catch(ArrayIndexOutOfBoundsException e)
         {
          System.out.println("ArrayIndexOutOfBounds Exception occurs");
         }
       catch(Exception e)
         {
          System.out.println("Parent Exception occurs");
         }
       System.out.println("rest of the code");
}
```

**Output:**

```
Arithmetic Exception occurs
rest of the code
```

# Catch Multiple Exceptions

```
public static void main(String[] args) {

    try{
        int a[]=new int[5];

        System.out.println(a[10]);
       }
       catch(ArithmeticException e)
         {
          System.out.println("Arithmetic Exception occurs");
         }
       catch(ArrayIndexOutOfBoundsException e)
         {
          System.out.println("ArrayIndexOutOfBounds Exception occurs");
         }
       catch(Exception e)
         {
          System.out.println("Parent Exception occurs");
         }
       System.out.println("rest of the code");
}
```

**Output:**

```
ArrayIndexOutOfBounds Exception occurs
rest of the code
```

# Catch Multiple Exceptions

```
public static void main(String[] args) {

    try{
        int a[]=new int[5];
        a[5]=30/0;
        System.out.println(a[10]);
       }
       catch(ArithmeticException e)
         {
          System.out.println("Arithmetic Exception occurs");
         }
       catch(ArrayIndexOutOfBoundsException e)
         {
          System.out.println("ArrayIndexOutOfBounds Exception occurs");
         }
       catch(Exception e)
         {
          System.out.println("Parent Exception occurs");
         }
       System.out.println("rest of the code");
}
```

**Output:**

```
Arithmetic Exception occurs
rest of the code
```

# Catch Multiple Exceptions

```
public static void main(String[] args) {


    try{
        String s=null;
        System.out.println(s.length());
       }
       catch(ArithmeticException e)
         {
          System.out.println("Arithmetic Exception occurs");
         }
       catch(ArrayIndexOutOfBoundsException e)
         {
          System.out.println("ArrayIndexOutOfBounds Exception occurs");
         }
       catch(Exception e)
         {
          System.out.println("Parent Exception occurs");
         }
       System.out.println("rest of the code");
}
```

**Output:**

```
Parent Exception occurs
rest of the code
```

# Catch Multiple Exceptions

```
 9  public class Main
10  {
11      public static void main(String args[]){
12         try{
13              int a[]=new int[5];
14              a[5]=30/0;
15         }
16         catch(Exception e){
17             System.out.println("common task completed");
18
19         }
20         catch(ArithmeticException e){
21             System.out.println("task1 is completed");
22         }
23         catch(ArrayIndexOutOfBoundsException e){
24             System.out.println("task 2 completed");
25         }
26         System.out.println("rest of the code...");
27       }
28  }
```

Compilation failed due to following error(s).

```
Main.java:17: error: exception ArithmeticException has already been caught
   catch(ArithmeticException e){System.out.println("task1 is completed");}
   ^
Main.java:18: error: exception ArrayIndexOutOfBoundsException has already been caught
   catch(ArrayIndexOutOfBoundsException e){System.out.println("task 2 completed");}
   ^
2 errors
```

# Nested try-catch Blocks

```
....
//Main try block
try {
   statement 1;
   statement 2;
   //try-catch block inside another try block
   try {
      statement 3;
      statement 4;
      //try-catch block inside nested try block
      try {
         statement 5;
         statement 6;
      }
      catch(Exception e2) {
         //Exception Message
      }
   }
   catch(Exception e1) {
       //Exception Message
   }

}
//Catch of Main(parent) try block
catch(Exception e3) {
      //Exception Message
}
....
```

# Nested try-catch Blocks

```
class NestedTry {

    // main method
    public static void main(String args[])
    {
        // Main try block
        try {

            // initializing array
            int a[] = { 1, 2, 3, 4, 5 };

            // trying to print element at index 5
            System.out.println(a[5]);

            // try-block2 inside another try block
            try {

                // performing division by zero
                int x = a[2] / 0;
            }
            catch (ArithmeticException e2) {
                System.out.println("division by zero is not possible");
            }
        }
        catch (ArrayIndexOutOfBoundsException e1) {
            System.out.println("ArrayIndexOutOfBoundsException");
            System.out.println("Element at such index does not exists");
        }
    }
    // end of main method
}
```

**Output:**

```
ArrayIndexOutOfBoundsException
Element at such index does not exists
```

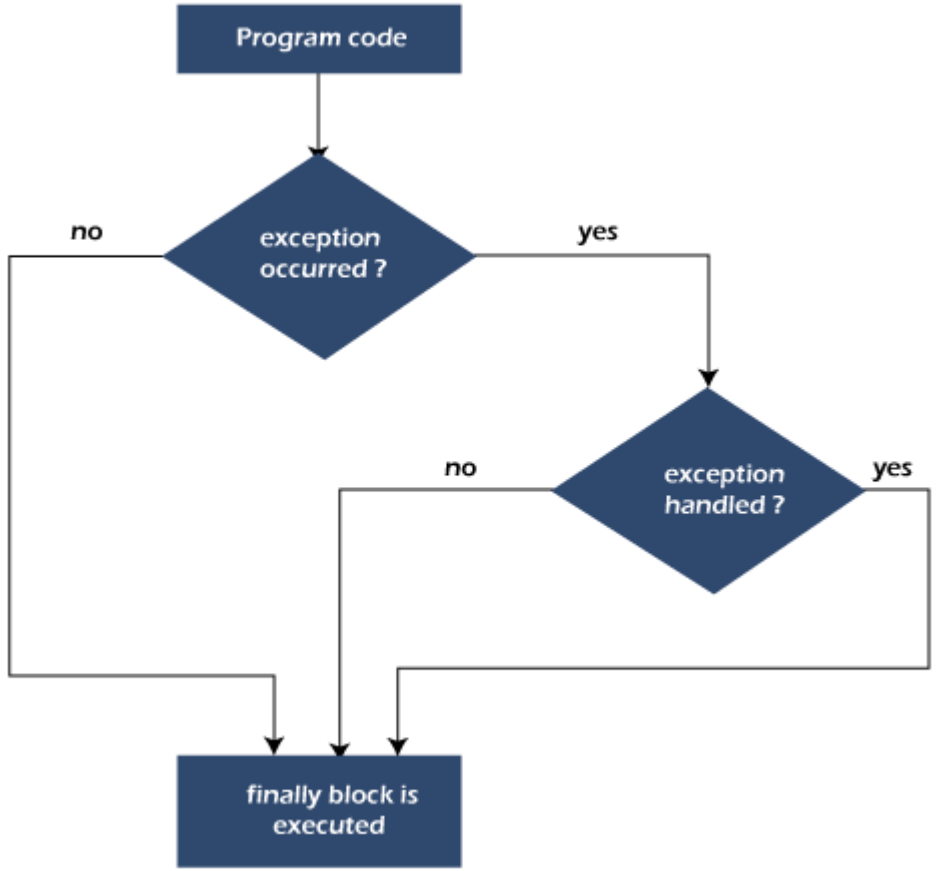
```
class Nesting {
    // main method
    public static void main(String args[])
    {
        // main try-block
        try {

            // try-block2
            try {

                // try-block3
                try {
                    int arr[] = { 1, 2, 3, 4 };
                    System.out.println(arr[10]);
                }

                // handles ArithmeticException if any
                catch (ArithmeticException e) {
                    System.out.println("Arithmetic exception");
                    System.out.println(" try-block1");
                }
            }

            // handles ArithmeticException if any
            catch (ArithmeticException e) {
                System.out.println("Arithmetic exception");
                System.out.println(" try-block2");
            }
        }

        // handles ArrayIndexOutOfBoundsException if any
        catch (ArrayIndexOutOfBoundsException e4) {
            System.out.print("ArrayIndexOutOfBoundsException");
            System.out.println(" main try-block");
        }
        catch (Exception e5) {
            System.out.print("Exception");
            System.out.println(" handled in main try-block");
        }
    }
}
```

**Output:**

```
ArrayIndexOutOfBoundsException main try-block
```

# finally block

- Java finally block is always executed whether an exception is handled or not.
- The finally block follows the try-catch block.
- If you don't handle the exception, before terminating the program, JVM executes finally block (if any).
- finally block in Java can be used to put "**cleanup**" code such as closing a file, closing connection, etc.
- For each try block there can be zero or more catch blocks, but only one finally block.



# The `finally` Clause

- The try statement may have an optional `finally` clause.
- If present, the `finally` clause must appear after all of the `catch` clauses.

```
try
{
   (try block statements...)
}
catch (ExceptionType ParameterName)
{
   (catch block statements...)
}
finally
{
   (finally block statements...)
}
```

# The `finally` Clause

- The *finally block* is one or more statements,
  - that are always executed after the try block has executed and
  - after any catch blocks have executed if an exception was thrown.
- The statements in the finally block execute whether an exception occurs or not.

```java
class TestFinallyBlock {
 public static void main(String args[]){
 try{
//below code do not throw any exception
  int data=25/5;
  System.out.println(data);
 }
//catch won't be executed
 catch(NullPointerException e){
System.out.println(e);
}
//executed regardless of exception occurred or not
 finally {
System.out.println("finally block is always executed");
}

System.out.println("rest of phe code...");
 }
}
```

```
5
finally block is always executed
rest of the code...
```

```java
public class TestFinallyBlock1{
    public static void main(String args[]){

    try {

     System.out.println("Inside the try block");

     //below code throws divide by zero exception
     int data=25/0;
     System.out.println(data);
    }
    //cannot handle Arithmetic type exception
    //can only accept Null Pointer type exception
    catch(NullPointerException e){
     System.out.println(e);
    }

    //executes regardless of exception occured or not
    finally {
     System.out.println("finally block is always executed");
    }

    System.out.println("rest of the code...");
    }
  }
```

```
Inside the try block
finally block is always executed
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at TestFinallyBlock1.main(TestFinallyBlock1.java:9)
```

```java
public class TestFinallyBlock2{
    public static void main(String args[]){

    try {

     System.out.println("Inside try block");

     //below code throws divide by zero exception
     int data=25/0;
     System.out.println(data);
    }

    //handles the Arithmetic Exception / Divide by zero exception
    catch(ArithmeticException e){
     System.out.println("Exception handled");
     System.out.println(e);
    }

    //executes regardless of exception occured or not
    finally {
     System.out.println("finally block is always executed");
    }

    System.out.println("rest of the code...");
    }
  }
```

```
Inside try block
Exception handled
java.lang.ArithmeticException: / by zero
finally block is always executed
rest of the code...
```

# The Stack Trace

- The *call stack* is an internal list of all the methods that are currently executing.
- A *stack trace* is a list of all the methods in the call stack.
- It indicates:
  - the method that was executing when an exception occurred and
  - all of the methods that were called in order to execute that method.

# Multi-Catch (Java 7)

- Beginning in Java 7, you can specify more than one exception in a `catch` clause:

```
try
{
}
catch(NumberFormatException | InputMismatchException ex)
{
}
```

Separate the exceptions with the | character.

# Checked and Unchecked Exceptions

- These exceptions can be avoided with properly written code.
- Unchecked exceptions, in most cases, should not be handled.
- All exceptions that are *not* derived from `Error` or `RuntimeException` are *checked exceptions*.

# Checked and Unchecked Exceptions

- If the code in a method can throw a checked exception, the method:
  - must handle the exception, or
  - it must have a `throws` clause listed in the method header.
- The `throws` clause informs the compiler what exceptions can be thrown from a method.

# Checked and Unchecked Exceptions

```
// This method will not compile!
public void displayFile(String name)
{
   // Open the file.
   File file = new File(name);
   Scanner inputFile = new Scanner(file);
   // Read and display the file's contents.
   while (inputFile.hasNext())
   {
     System.out.println(inputFile.nextLine());
   }
   // Close the file.
   inputFile.close();
}
```

# Checked and Unchecked Exceptions

- The code in this method is capable of throwing checked exceptions.
- The keyword `throws` can be written at the end of the method header, followed by a list of the types of exceptions that the method can throw.

```
public void displayFile(String name)
      throws FileNotFoundException
```

# Throwing Exceptions

- You can write code that:
  - throws one of the standard Java exceptions, or
  - an instance of a custom exception class that you have designed.
- The `throw` statement is used to manually throw an exception.

```
throw new ExceptionType(MessageString);
```

- The `throw` statement causes an exception object to be created and thrown.

# Throwing Exceptions

- The *MessageString* argument contains a custom error message that can be retrieved from the exception object's `getMessage` method.
- If you do not pass a message to the constructor, the exception will have a null message.

```
throw new Exception("Out of fuel");
```

  - *Note: Don't confuse the `throw` statement with the `throws` clause.*
- Example: DieExceptionDemo.java

```java
import java.util.Random;

/**
   The Die class simulates a six-sided die.
*/

public class Die
{
   private final int MIN_SIDES = 4;
   private int sides;   // Number of sides
   private int value;   // The die's value

   /**
      The constructor performs an initial
      roll of the die.
      @param numSides The number of sides for this
   */

   public Die(int numSides)
   {
      // Validate the number of sides.
      if (numSides < MIN_SIDES)
      {
         throw new IllegalArgumentException(
            "The die must have at least " +
            MIN_SIDES + " sides.");
      }

      // Store the number of sides and roll.
      sides = numSides;
      roll();
   }

   /**
      The roll method simulates the rolling of
      the die.
   */

   public void roll()
   {
      // Create a Random object.
      Random rand = new Random();

      // Get a random value for the die.
      value = rand.nextInt(sides) + 1;
   }

   /**
      getSides method
      @return The number of sides for this die.
   */

   public int getSides()
   {
      return sides;
   }

   /**
      getValue method
      @return The value of the die.
   */

   public int getValue()
   {
      return value;
   }
}
```

```java
/**
   This program demonstrates how the Die class throws
   an exception when an invalid value is passed to the
   constructor.
*/

public class DiceExceptionDemo
{
   public static void main(String[] args)
   {
      final int DIE_SIDES = 1;  // Number of sides

      // Create an instance of the Die class.
      Die die = new Die(DIE_SIDES);

      System.out.println("Initial value of the die:");
      System.out.println(die.getValue());
   }
}
```

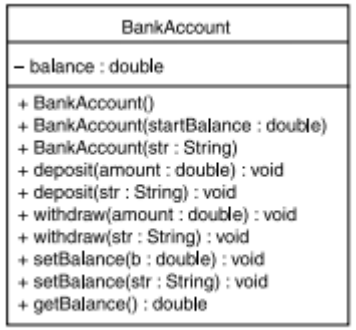
# Creating Exception Classes

- You can create your own exception classes by deriving them from the `Exception` class or one of its derived classes.

# Creating Exception Classes

- Some examples of exceptions that can affect a bank account:
  - A negative starting balance is passed to the constructor.
  - A negative interest rate is passed to the constructor.
  - A negative number is passed to the deposit method.
  - A negative number is passed to the withdraw method.
  - The amount passed to the withdraw method exceeds the account's balance.
- We can create exceptions that represent each of these error conditions.

# Creating Exception Classes




```
/**
   NegativeStartingBalance exceptions are thrown by the
   BankAccount class when a negative starting balance is
   passed to the constructor.
*/

public class NegativeStartingBalance
                     extends Exception
{
   /**
      This constructor uses a generic
      error message.
   */

   public NegativeStartingBalance()
   {
      super("Error: Negative starting balance");
   }

   /**
      This constructor specifies the bad starting
      balance in the error message.
      @param The bad starting balance.
   */

   public NegativeStartingBalance(double amount)
   {
      super("Error: Negative starting balance: " +
            amount);
   }
}
```

```
public BankAccount(double startBalance)
                throws NegativeStartingBalance
{
   if (startBalance < 0)
      throw new NegativeStartingBalance(startBalance);

   balance = startBalance;
}
```

```
/**
   This program demonstrates how the BankAccount
   class constructor throws custom exceptions.
*/

public class AccountTest
{
   public static void main(String [] args)
   {
      // Force a NegativeStartingBalance exception.
      try
      {
         BankAccount account =
                     new BankAccount(-100.0);
      }
      catch(NegativeStartingBalance e)
      {
         System.out.println(e.getMessage());
      }
   }
}
```