Java Exception Handling Notes

This document provides comprehensive notes on Error and Exception handling in Java, based on the provided text.

Error vs. Exception

Q)What is the difference between Error and Exception?
—---------- Ans: —---

- **Error:** A problem preventing program execution, typically unrecoverable programmatically.
    - Generally two types:
        1. Compilation Errors
        2. Runtime Errors
    - **Compilation Error:** Identified by the compiler during compilation.
        1. Types of Compilation Errors:
            - **Lexical Errors:** Mistakes in tokens (e.g., misspelled keywords like `nit` instead of `int`).

              Plaintext

              ```
              EX:  int i = 10;  ---> Valid
                   nit i = 20;  ---> Lexical Error
              ```

            - **Syntax Errors:** Mistakes in language syntax (e.g., missing semicolon).

              Plaintext

              ```
              EX:  int i = 10;  -----> Valid
                   int i = 20  -----> Syntax Error
                   int = i 30;  -----> Syntax Errors
              ```

            - **Semantic Errors:** Meaningless statements, incompatible operations (e.g., adding an integer and a boolean).

              Plaintext

              ```
              EX:  int a = 10; int b = 20; int c = a + b;  --->
              Valid

              EX:  int a = 10; boolean b = true; char c = a + b;
              ----> Semantic Error
              ```

        2. Note: Languages may have additional compilation errors specific to their conventions (e.g., Unreachable Statement in Java).

- **Runtime Errors:** Problems occurring during application runtime that cannot be fixed programmatically.
    1. Examples: Unavailability of IO components, JVM Internal Problem, Insufficient Main Memory, StackOverflowError.

    - **Exception:** Problems occurring at runtime that *can* be handled programmatically.
- Examples: ArithmeticException, NullPointerException.

Java

```
int a = 100;
int b = 0;
if(b != 0){
    float f = a / b;
}

Date d = null;
if(d != null){
    System.out.println(d.toString());
}
```

—----------

Understanding Exceptions and Termination

- **Exception Definition:** An unexpected runtime event causing abnormal termination. Can originate from various sources like user input, database, network.
- **Application Termination Types:**
    1. **Smooth Termination:** Program ends normally at the end.
    2. **Abnormal Termination:** Program stops unexpectedly in the middle.
- In Java, Exceptions typically cause Abnormal Termination, leading to potential issues like OS crashes, network hangs, database collapses, server downtime.
- **Goal of Exception Handling:** To avoid abnormal termination and achieve smooth termination.

Java Robustness

Java is robust due to:

1. **Good Memory Management:** Dynamic Heap memory management system.
2. **Good Exception Handling:** Rich predefined libraries to handle frequent exceptions.
    - Supports defining and handling custom exceptions.

Types of Exceptions in Java

1. **Predefined Exceptions:** Defined by Java, represented by classes.
2. **User defined Exceptions:** Custom exceptions created by developers.

Predefined Exceptions

- Two main types: Checked and Unchecked.

Q)What is the difference between Checked Exception and Unchecked Exception? —---------- Ans: —---

- **Checked Exception:** Recognized by the compiler at compilation time. (Exception actually occurs at runtime).

- **Unchecked Exception:** Recognized by the JVM at runtime, not by the compiler at compilation time.
    - Examples: `RuntimeException` and its subclasses, `Error` and its subclasses. All other exception classes are typically checked.
      —----------

- **Subtypes of Checked Exceptions:**

Q) What is the difference between a Partially Checked exception and Pure Checked Exception? —---------- Ans: —-----

- **Pure Checked Exception:** A checked exception whose child classes are *only* checked exceptions.
    - Example: `IOException`
- **Partially Checked Exception:** A checked exception with *at least one* unchecked child exception.
    - Examples: `Throwable`, `Exception` —----------

Common Predefined Exceptions Overview

(Examples include code and typical output/messages)

1. `java.lang.ArithmeticException`: Division by zero.

   Java

   ```
   public class Main {
     public static void main(String[] args){
          int a = 100;
          int b = 0;
          float f = a/b; // Exception occurs here
     }
   }
   ```

   Plaintext

   ```
   Exception in thread "main" java.lang.ArithmeticException: /
   by zero
       at Main.main(Main.java:5)
   ```

    - Message Parts: Name, Description, Location.

2. `java.lang.NullPointerException`: Accessing member on a null reference.

Java

```
import java.util.Date;

public class Main {
  public static void main(String[] args){
       Date date = null;
       System.out.println(date.toString()); // Exception occurs here
  }
}
```

Plaintext

```
Exception Name : java.lang.NullPointerException
Exception Description : Cannot invoke "java.util.Date.toString()"
because "date" is null
Exception Location : Main.java: 6
```

3. `java.lang.ArrayIndexOutOfBoundsException`: Accessing array element outside bounds.

Java

```
public class Main {
  public static void main(String[] args){
       int[] ints = {10,20,30,40,50};
       System.out.println(ints[2]);  // Valid
       System.out.println(ints[10]); // Exception occurs here
  }
}
```

Plaintext

```
30
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: Index 10 out of bounds for
length 5
    at Main.main(Main.java:7)
```

4. `java.lang.StringIndexOutOfBoundsException`: String operation with index outside bounds.

Java

```
public class Main {
  public static void main(String[] args){
       String data = "Durgasoft";
       System.out.println(data.charAt(5));  // Valid
       System.out.println(data.charAt(20)); // Exception occurs here
  }
}
```

Plaintext

```
Exception Name : java.lang.StringIndexOutOfBoundsException
```

```
Exception Description : Index 20 out of bounds for length 9
Exception Location : Main.java:7
```

5. `java.lang.ClassNotFoundException`: Class.forName() cannot find class.

Java

```
class A{
  static{
       System.out.println("Class A Loading");
  }
}
public class Main {
  public static void main(String[] args)throws Exception{
       Class.forName("B"); // Exception occurs here if 'B' not found
  }
}
```

Plaintext

```
Exception Name : java.lang.ClassNotFoundException
Exception Description : B
Exception Location : Main.java: 7
```

6. `java.lang.InstantiationException`: Class.newInstance() fails (e.g., no 0-arg constructor).

Java

```
class A{
  public A(int i){
       System.out.println("A-con");
  }
}
public class Main {
  public static void main(String[] args)throws Exception{
       Class cls = Class.forName("A");
       cls.newInstance(); // Exception occurs here (no 0-arg
constructor)
  }
}
```

Plaintext

```
Exception Name : java.lang.InstantiationException
Exception Description : A
Exception Location : Main.java: 9
```

7. `java.lang.IllegalAccessException`: Class.newInstance() tries to access private constructor.

Java

```
class A{
  private A(){
```

```
        System.out.println("A-con");
  }
}
public class Main {
  public static void main(String[] args)throws Exception{
        Class cls = Class.forName("A");
        cls.newInstance(); // Exception occurs here (constructor is
private)
  }
}
```

Plaintext

```
Exception Name : java.lang.IllegalAccessException
Exception Description : class Main cannot access a member of
  class A with modifiers "private"
Exception Location : Main.java: 9
```

8. `java.lang.ClassCastException`: Invalid object casting (e.g., superclass object to subclass reference).

Java

```
class A{ }
class B extends A{ }
public class Main {
  public static void main(String[] args){
        A a = new A();
        B b = (B) a; // Exception occurs here (A object cannot be cast
to B)
  }
}
```

Plaintext

```
Exception Name : java.lang.ClassCastException
Exception Description : class A cannot be cast to class B
Exception Location : Main.java: 10
```

9. `java.io.FileNotFoundException`: File not found when using stream/reader.

Java

```
import java.io.FileInputStream;

public class Main {
  public static void main(String[] args)throws Exception{
        FileInputStream fis =new
FileInputStream("E:/abc/xyz/welcome.txt"); // Exception if file not
found
  }
}
```

Plaintext

```
Exception Name: java.io.FileNotFoundException
```

```
Exception Description: E:/abc/xyz/welcome.txt (No such file or
 directory)
 Exception Location : Main.java: 5
```

Exception Handling Mechanisms

Two main ways to handle exceptions:

1. `throws` keyword
2. `try-catch-finally` block

`throw` keyword

- Purpose: To explicitly raise an exception object.
- Usage: Inside method body.
- Allows: Only one exception object at a time.
- Syntax: `throw new ExceptionName([ParamValues]);`

Java

```
class Student{
  String sid, sname, saddr;
  float smarksInPercentage;

  public Student(String sid, String sname, String saddr, float
smarksInPercentage) {
       this.sid = sid;
       this.sname =sname;
       this.saddr =saddr;
       this.smarksInPercentage =smarksInPercentage;
  }

  public void getStudentDetails(){
       System.out.println("Student Details");
       System.out.println("-------------------------");
       System.out.println("Student Id           : "+sid);
       System.out.println("Student Name         : "+sname);
       System.out.println("Student Address      : "+saddr);
       System.out.println("Student Marks In Percentage :
"+smarksInPercentage);
       if(smarksInPercentage < 0 || smarksInPercentage > 100){
           throw new RuntimeException("Invalid Student Marks, Provide
Student Marks between 0 to 100");
       }else{
           System.out.println("Status             : Student Marks are
Valid");
       }
  }
}
public class Main {
  public static void main(String[] args)throws Exception{

       Student student1 = new Student("S-111", "Durga", "Hyd", 78.0f);
       student1.getStudentDetails();
       System.out.println();

       Student student2 = new Student("S-222", "Ramana", "Hyd", 150.0f);
       student2.getStudentDetails(); // This call will throw the exception
```

```
  }
}
```

Plaintext

```
Student Details
-------------------------
Student Id           : S-111
Student Name         : Durga
Student Address      : Hyd
Student Marks In Percentage : 78.0
Status             : Student Marks are Valid

Student Details
-------------------------
Student Id           : S-222
Student Name         : Ramana
Student Address      : Hyd
Student Marks In Percentage : 150.0
Exception in thread "main" java.lang.RuntimeException:
Invalid Student Marks, Provide Student Marks between 0 to 100
    at Student.getStudentDetails(Main.java:23)
    at Main.main(Main.java:37)
```

`throws` keyword

- Purpose: To declare that a method might throw an exception, bypassing it to the caller. Not an actual handler.
- Usage: In method signature/prototype.
- Allows: One or more exception names (comma-separated).

Java

```
import java.io.IOException;

class A{
  void add()throws Exception { // Declares it might throw Exception (or its
subclasses)
       concat();
  }
  void concat()throws IOException { // Declares it might throw IOException
       throw new
IOException("My OWN IOException"); // Throws IOException
  }
}
class Test {
   public static void main(String[] args)throws Throwable { // Declares it
might throw Throwable (or its subclasses)
       A a = new A();
       a.add(); // Calls add(), which might throw IOException. main
declares it can handle Throwable (a superclass).
   }
}
```

Plaintext

```
nagoorn@Nagoors-MacBook-Pro COREJAVA-830 % javac Test.java
nagoorn@Nagoors-MacBook-Pro COREJAVA-830 % java Test
Exception in thread "main" java.io.IOException: My OWN
IOException
         at A.concat(Test.java:8)
         at A.add(Test.java:5)
```

```
        at Test.main(Test.java:14)
```

Comparison: `throw` vs. `throws`

Q) What are the differences between ‘throws’ and ‘throw’ keywords? —---------- Ans: —--- See table below for comparison. —----------

+--------------------+----------------------------------+----------------------------------+ | Feature | `throw` keyword | `throws` keyword | +====================+==================================+==================================+ | **Purpose** | Explicitly raise an exception | Declare exceptions a method might | | | object. | throw, delegating handling. | +--------------------+----------------------------------+----------------------------------+ | **Usage Location** | Inside method body/implementation | In method signature (after params) | +--------------------+----------------------------------+----------------------------------+ | **Operand** | Single exception object | One or more exception class names | | | (`throw new MyException();`) | (`throws Exception1, Exception2`) | +--------------------+----------------------------------+----------------------------------+

`try-catch-finally` Block

- Used for exact exception handling at the point of occurrence.
- **Structure:**

  Plaintext

  ```
  try {
      // Code that might throw exceptions
  } catch (ExceptionType1 e1) {
      // Handle ExceptionType1
  } catch (ExceptionType2 e2) {
      // Handle ExceptionType2
  } finally {
      // Code that *always* executes
  }
  ```

`try` block

- Contains code potentially throwing exceptions.
- Only "doubtful" code needs to be in `try`.
- If exception occurs, remaining `try` code skipped, jump to `catch`.
- If no exception, `try` completes, jump to `finally` (skipping `catch`).

`catch` block

- Catches exceptions from the preceding `try`.
- Handles the exception (e.g., display info, logging, recovery).

- Executed *only* if an exception matching the type is thrown in `try`.

- **Ways to display Exception Details:**
  1. `e.printStackTrace()`: Prints full stack trace (name, description, location).
  2. `e.toString()`: Prints exception name and description (default for `System.out.println(e)`).
  3. `e.getMessage()`: Prints only the exception description.

Java

```
class Test {
  public static void main(String[] args){
       try{
           int a = 100;
           int b = 0;
           float f = a/b; // ArithmeticException
       }catch(Exception e){
           e.printStackTrace(); // Full stack trace
           System.out.println();
           System.out.println(e); // toString() output
           System.out.println();
           System.out.println(e.getMessage()); // Message only
       }finally{
           // finally block content
       }
   }
}
```

Plaintext

```
nagoorn@Nagoors-MacBook-Pro COREJAVA-830 % javac Test.java
nagoorn@Nagoors-MacBook-Pro COREJAVA-830 % java Test
java.lang.ArithmeticException: / by zero
         at Test.main(Test.java:7)

java.lang.ArithmeticException: / by zero

/ by zero
```

`finally` block

- Contains code guaranteed to execute regardless of whether an exception occurred or was caught.

- Purpose: Resource cleanup (closing files, connections, etc.).

- **Execution Flow Examples:**
  - **No Exception:** `try` -> `finally` -> Code after `finally`

    Java

    ```
    public class Main {
      public static void main(String[] args) {
           System.out.println("Before try");
           try{
    ```

```
            System.out.println("Inside try");
        }catch (Exception e){
            System.out.println("Inside catch");
        }finally{
            System.out.println("Inside finally");
        }
        System.out.println("After finally");
    }
}
```

Plaintext

```
Before try
Inside try
Inside finally
After finally
```

- **Exception Caught:** `try` -> `catch` -> `finally` -> Code after `finally`

Java

```
public class Main {
  public static void main(String[] args) {
        System.out.println("Before try");
        try{
            System.out.println("Inside try, Before Exception");
            float f = 100/0; // Exception occurs
            System.out.println("Inside try, After Exception");
// Skipped
        }catch (Exception e){
            System.out.println("Inside catch"); // Executed
        }finally{
            System.out.println("Inside finally"); // Executed
        }
        System.out.println("After finally"); // Executed
  }
}
```

Plaintext

```
Before try
Inside try, Before Exception
Inside catch
Inside finally
After finally
```

`try-finally` block

- A `try` can be followed by `finally` without a `catch`.

Java

```
try {
    // Code that might throw
} finally {
    // Cleanup code
```

```
}
```

- If an exception occurs in `try`, `finally` executes, then the exception is re-thrown to the caller/JVM.

Java

```
public class Main {
  public static void main(String[] args) {
       try{
           System.out.println("Inside try");
           float f = 100/0; // Exception occurs
       }finally{
           System.out.println("Inside finally"); // Executed
       }
       // Code here is NOT reached because exception is re-thrown
  }
}
```

Plaintext

```
Inside try
Inside finally
Exception in thread "main" java.lang.ArithmeticException: /
by zero
    at Main.main(Main.java:5)
```

`try-catch` block

- A `try` can be followed by `catch` without a `finally`.

Java

```
try {
    // Code that might throw
} catch (Exception e) {
    // Handle exception
}
```

Nested `try-catch-finally`

- Allowed within `try`, `catch`, or `finally` blocks.

Plaintext

```
// Example structure (simplified)
try {
    // Outer try code
    try {
        // Inner try code
    } catch (InnerException e) {
        // Inner catch
    }
} catch (OuterException e) {
    // Outer catch
```

```
} finally {
    // Outer finally
}
```

Multiple `catch` blocks

- A single `try` block can have multiple `catch` blocks for different exception types.
- **Ordering Rules:**
    1. **No Inheritance:** If catch exception types are unrelated, order doesn't matter.

       Java

       ```
       try{ /* ... */
       }catch(ArithmeticException e){ /* ... */
       }catch(NullPointerException e){ /* ... */
       }catch(ArrayIndexOutOfBoundsException e){ /* ... */
       }
       // Valid in any order
       ```

    2. **With Inheritance:** Catch blocks must be ordered from most specific (subclass) to least specific (superclass).

       Java

       ```
       try{ /* ... */
       }catch(ArithmeticException e){ /* Handle specific */
       }catch(RuntimeException e){    /* Handle less specific */
       }catch(Exception e){         /* Handle most general */
       }
       // Status: Valid (Specific to General)

       try{ /* ... */
       }catch(Exception e){         /* Most general first */
       }catch(RuntimeException e){    /* Subclass is unreachable */
       }catch(ArithmeticException e){ /* Subclass is unreachable */
       }
       // Status: Invalid (Compile-time error: Unreachable catch
       block)
       ```

    3. **Pure Checked Exceptions:** A `catch` block for a pure checked exception requires the corresponding `try` block (or calling method) to be able to throw that *specific* checked exception.

       Java

       ```
       EX:
       try{
           throw new ArithmeticException(); // Throws RuntimeException
       (Unchecked)
       }catch(ArithmeticException e){ /* OK */
       }catch(IOException e){ // IOException is Checked. Try block
       doesn't throw IOException.
       }catch(NullPointerException e){ /* OK */
       }
       // Status: Invalid (IOException catch is unreachable/compiler
       error)
       ```

Java

```
EX:
try{
    throw new IOException(); // Throws IOException (Checked)
}catch(ArithmeticException e){ /* OK */
}catch(IOException e){ // IOException is thrown and caught
here.
}catch(NullPointerException e){ /* OK */
}
// Status: Valid (Checked exception is thrown and caught)
```

Java Exception Handling Notes

This document provides comprehensive notes on Error and Exception handling in Java, based on the provided text.

Error vs. Exception

Q)What is the difference between Error and Exception?
—---------- Ans: —---

- **Error:** A problem preventing program execution, typically unrecoverable programmatically.
    - Generally two types:
        1. Compilation Errors
        2. Runtime Errors
    - **Compilation Error:** Identified by the compiler during compilation.
        1. Types of Compilation Errors:
            - **Lexical Errors:** Mistakes in tokens (e.g., misspelled keywords like `nit` instead of `int`).

              Plaintext

              ```
              EX:  int i = 10;  ---> Valid
                   nit i = 20;  ---> Lexical Error
              ```

            - **Syntax Errors:** Mistakes in language syntax (e.g., missing semicolon).

              Plaintext

              ```
              EX:  int i = 10;  -----> Valid
                   int i = 20  -----> Syntax Error
                   int = i 30;  -----> Syntax Errors
              ```

            - **Semantic Errors:** Meaningless statements, incompatible operations (e.g., adding an integer and a boolean).

Plaintext

```
EX:  int a = 10; int b = 20; int c = a + b;  ---> Valid

EX:  int a = 10; boolean b = true; char c = a + b; ----> Semantic Error
```

2. Note: Languages may have additional compilation errors specific to their conventions (e.g., Unreachable Statement in Java).

- **Runtime Errors:** Problems occurring during application runtime that cannot be fixed programmatically.
    1. Examples: Unavailability of IO components, JVM Internal Problem, Insufficient Main Memory, StackOverflowError.

    - **Exception:** Problems occurring at runtime that *can* be handled programmatically.
- Examples: ArithmeticException, NullPointerException.

Java

```
int a = 100;
int b = 0;
if(b != 0){
    float f = a / b;
}

Date d = null;
if(d != null){
    System.out.println(d.toString());
}
```

—--

Understanding Exceptions and Termination

- **Exception Definition:** An unexpected runtime event causing abnormal termination. Can originate from various sources like user input, database, network.
- **Application Termination Types:**
    1. **Smooth Termination:** Program ends normally at the end.
    2. **Abnormal Termination:** Program stops unexpectedly in the middle.
- In Java, Exceptions typically cause Abnormal Termination, leading to potential issues like OS crashes, network hangs, database collapses, server downtime.
- **Goal of Exception Handling:** To avoid abnormal termination and achieve smooth termination.

Java Robustness

Java is robust due to:

1. **Good Memory Management:** Dynamic Heap memory management system.

2. **Good Exception Handling:** Rich predefined libraries to handle frequent exceptions.
    - Supports defining and handling custom exceptions.

Types of Exceptions in Java

1. **Predefined Exceptions:** Defined by Java, represented by classes.
2. **User defined Exceptions:** Custom exceptions created by developers.

Predefined Exceptions

- Two main types: Checked and Unchecked.

Q)What is the difference between Checked Exception and Unchecked Exception?
—---------- Ans: —---

- **Checked Exception:** Recognized by the compiler at compilation time. (Exception actually occurs at runtime).

- **Unchecked Exception:** Recognized by the JVM at runtime, not by the compiler at compilation time.
    - Examples: `RuntimeException` and its subclasses, `Error` and its subclasses. All other exception classes are typically checked.
      —----------

- **Subtypes of Checked Exceptions:**

Q) What is the difference between a Partially Checked exception and Pure Checked Exception? —---------- Ans: —----

- **Pure Checked Exception:** A checked exception whose child classes are *only* checked exceptions.
    - Example: `IOException`
- **Partially Checked Exception:** A checked exception with *at least one* unchecked child exception.
    - Examples: `Throwable`, `Exception` —----------

Common Predefined Exceptions Overview

(Examples include code and typical output/messages)

1. `java.lang.ArithmeticException`: Division by zero.

   Java

   ```
   public class Main {
     public static void main(String[] args){
          int a = 100;
          int b = 0;
          float f = a/b; // Exception occurs here
     }
   ```

```
}
```

Plaintext

```
Exception in thread "main" java.lang.ArithmeticException: /
by zero
    at Main.main(Main.java:5)
```

- Message Parts: Name, Description, Location.

2. `java.lang.NullPointerException`: Accessing member on a null reference.

Java

```
import java.util.Date;

public class Main {
  public static void main(String[] args){
       Date date = null;
       System.out.println(date.toString()); // Exception occurs here
  }
}
```

Plaintext

```
Exception Name : java.lang.NullPointerException
Exception Description : Cannot invoke "java.util.Date.toString()"
because "date" is null
Exception Location : Main.java: 6
```

3. `java.lang.ArrayIndexOutOfBoundsException`: Accessing array element outside bounds.

Java

```
public class Main {
  public static void main(String[] args){
       int[] ints = {10,20,30,40,50};
       System.out.println(ints[2]);  // Valid
       System.out.println(ints[10]); // Exception occurs here
  }
}
```

Plaintext

```
30
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: Index 10 out of bounds for
length 5
    at Main.main(Main.java:7)
```

4. `java.lang.StringIndexOutOfBoundsException`: String operation with index outside bounds.

Java

```
public class Main {
  public static void main(String[] args){
       String data = "Durgasoft";
       System.out.println(data.charAt(5));  // Valid
       System.out.println(data.charAt(20)); // Exception occurs here
  }
}
```

Plaintext

```
Exception Name : java.lang.StringIndexOutOfBoundsException
Exception Description : Index 20 out of bounds for length 9
Exception Location : Main.java:7
```

5. `java.lang.ClassNotFoundException`: Class.forName() cannot find class.

Java

```
class A{
  static{
       System.out.println("Class A Loading");
  }
}
public class Main {
  public static void main(String[] args)throws Exception{
       Class.forName("B"); // Exception occurs here if 'B' not found
  }
}
```

Plaintext

```
Exception Name : java.lang.ClassNotFoundException
Exception Description : B
Exception Location : Main.java: 7
```

6. `java.lang.InstantiationException`: Class.newInstance() fails (e.g., no 0-arg constructor).

Java

```
class A{
  public A(int i){
       System.out.println("A-con");
  }
}
public class Main {
  public static void main(String[] args)throws Exception{
       Class cls = Class.forName("A");
       cls.newInstance(); // Exception occurs here (no 0-arg
constructor)
  }
}
```

Plaintext

```
Exception Name : java.lang.InstantiationException
```

```
Exception Description : A
Exception Location : Main.java: 9
```

7. `java.lang.IllegalAccessException`: Class.newInstance() tries to access private constructor.

Java

```
class A{
  private A(){
       System.out.println("A-con");
  }
}
public class Main {
  public static void main(String[] args)throws Exception{
       Class cls = Class.forName("A");
       cls.newInstance(); // Exception occurs here (constructor is
private)
  }
}
```

Plaintext

```
Exception Name : java.lang.IllegalAccessException
Exception Description : class Main cannot access a member of
  class A with modifiers "private"
Exception Location : Main.java: 9
```

8. `java.lang.ClassCastException`: Invalid object casting (e.g., superclass object to subclass reference).

Java

```
class A{ }
class B extends A{ }
public class Main {
  public static void main(String[] args){
       A a = new A();
       B b = (B) a; // Exception occurs here (A object cannot be cast
to B)
  }
}
```

Plaintext

```
Exception Name : java.lang.ClassCastException
Exception Description : class A cannot be cast to class B
Exception Location : Main.java: 10
```

9. `java.io.FileNotFoundException`: File not found when using stream/reader.

Java

```
import java.io.FileInputStream;
```

```
public class Main {
  public static void main(String[] args)throws Exception{
       FileInputStream fis =new
FileInputStream("E:/abc/xyz/welcome.txt"); // Exception if file not
found
  }
}
```

Plaintext

```
Exception Name: java.io.FileNotFoundException
Exception Description: E:/abc/xyz/welcome.txt (No such file or
 directory)
 Exception Location : Main.java: 5
```

Exception Handling Mechanisms

Two main ways to handle exceptions:

1. `throws` keyword
2. `try-catch-finally` block

`throw` keyword

- Purpose: To explicitly raise an exception object.
- Usage: Inside method body.
- Allows: Only one exception object at a time.
- Syntax: `throw new ExceptionName([ParamValues]);`

Java

```
class Student{
  String sid, sname, saddr;
  float smarksInPercentage;

  public Student(String sid, String sname, String saddr, float
smarksInPercentage) {
       this.sid = sid;
       this.sname =sname;
       this.saddr =saddr;
       this.smarksInPercentage =smarksInPercentage;
  }

  public void getStudentDetails(){
       System.out.println("Student Details");
       System.out.println("-------------------------");
       System.out.println("Student Id           : "+sid);
       System.out.println("Student Name         : "+sname);
       System.out.println("Student Address      : "+saddr);
       System.out.println("Student Marks In Percentage :
"+smarksInPercentage);
       if(smarksInPercentage < 0 || smarksInPercentage > 100){
           throw new RuntimeException("Invalid Student Marks, Provide
Student Marks between 0 to 100");
       }else{
           System.out.println("Status             : Student Marks are
Valid");
```

```
        }
  }
}
public class Main {
  public static void main(String[] args)throws Exception{

        Student student1 = new Student("S-111", "Durga", "Hyd", 78.0f);
        student1.getStudentDetails();
        System.out.println();

        Student student2 = new Student("S-222", "Ramana", "Hyd", 150.0f);
        student2.getStudentDetails(); // This call will throw the exception
  }
}
```

Plaintext

```
Student Details
-------------------------
Student Id           : S-111
Student Name         : Durga
Student Address      : Hyd
Student Marks In Percentage : 78.0
Status             : Student Marks are Valid

Student Details
-------------------------
Student Id           : S-222
Student Name         : Ramana
Student Address      : Hyd
Student Marks In Percentage : 150.0
Exception in thread "main" java.lang.RuntimeException:
Invalid Student Marks, Provide Student Marks between 0 to 100
    at Student.getStudentDetails(Main.java:23)
    at Main.main(Main.java:37)
```

`throws` keyword

- Purpose: To declare that a method might throw an exception, bypassing it to the caller. Not an actual handler.
- Usage: In method signature/prototype.
- Allows: One or more exception names (comma-separated).

Java

```
import java.io.IOException;

class A{
  void add()throws Exception { // Declares it might throw Exception (or its
subclasses)
        concat();
  }
  void concat()throws IOException { // Declares it might throw IOException
        throw new
IOException("My OWN IOException"); // Throws IOException
  }
}
class Test {
   public static void main(String[] args)throws Throwable { // Declares it
might throw Throwable (or its subclasses)
        A a = new A();
```

```
       a.add(); // Calls add(), which might throw IOException. main
declares it can handle Throwable (a superclass).
   }
}
```

Plaintext

```
nagoorn@Nagoors-MacBook-Pro COREJAVA-830 % javac Test.java
nagoorn@Nagoors-MacBook-Pro COREJAVA-830 % java Test
Exception in thread "main" java.io.IOException: My OWN
IOException
         at A.concat(Test.java:8)
         at A.add(Test.java:5)
         at Test.main(Test.java:14)
```

Comparison: `throw` vs. `throws`

Q) What are the differences between ‘throws’ and ‘throw’ keywords? —---------- Ans: —--- See table below for comparison. —----------

+--------------------+----------------------------------+----------------------------------+ | Feature | `throw` keyword | `throws` keyword | +====================+==================================+==================================+ | **Purpose** | Explicitly raise an exception | Declare exceptions a method might | | | object. | throw, delegating handling. | +--------------------+----------------------------------+----------------------------------+ | **Usage Location** | Inside method body/implementation | In method signature (after params) | +--------------------+----------------------------------+----------------------------------+ | **Operand** | Single exception object | One or more exception class names | | | (`throw new MyException();`) | (`throws Exception1, Exception2`) | +--------------------
+----------------------------------+----------------------------------+

`try-catch-finally` Block

- Used for exact exception handling at the point of occurrence.
- **Structure:**

  Plaintext

  ```
  try {
      // Code that might throw exceptions
  } catch (ExceptionType1 e1) {
      // Handle ExceptionType1
  } catch (ExceptionType2 e2) {
      // Handle ExceptionType2
  } finally {
      // Code that *always* executes
  }
  ```

`try` block

- Contains code potentially throwing exceptions.

- Only "doubtful" code needs to be in `try`.
- If exception occurs, remaining `try` code skipped, jump to `catch`.
- If no exception, `try` completes, jump to `finally` (skipping `catch`).

`catch` block

- Catches exceptions from the preceding `try`.
- Handles the exception (e.g., display info, logging, recovery).
- Executed *only* if an exception matching the type is thrown in `try`.
- **Ways to display Exception Details:**
  1. `e.printStackTrace()`: Prints full stack trace (name, description, location).
  2. `e.toString()`: Prints exception name and description (default for `System.out.println(e)`).
  3. `e.getMessage()`: Prints only the exception description.

Java

```
class Test {
  public static void main(String[] args){
       try{
           int a = 100;
           int b = 0;
           float f = a/b; // ArithmeticException
       }catch(Exception e){
           e.printStackTrace(); // Full stack trace
           System.out.println();
           System.out.println(e); // toString() output
           System.out.println();
           System.out.println(e.getMessage()); // Message only
       }finally{
           // finally block content
       }
   }
}
```

Plaintext

```
nagoorn@Nagoors-MacBook-Pro COREJAVA-830 % javac Test.java
nagoorn@Nagoors-MacBook-Pro COREJAVA-830 % java Test
java.lang.ArithmeticException: / by zero
         at Test.main(Test.java:7)

java.lang.ArithmeticException: / by zero

/ by zero
```

`finally` block

- Contains code guaranteed to execute regardless of whether an exception occurred or was caught.

- Purpose: Resource cleanup (closing files, connections, etc.).

- **Execution Flow Examples:**
  - **No Exception:** `try` -> `finally` -> Code after `finally`

    Java

    ```
    public class Main {
      public static void main(String[] args) {
            System.out.println("Before try");
            try{
                System.out.println("Inside try");
            }catch (Exception e){
                System.out.println("Inside catch");
            }finally{
                System.out.println("Inside finally");
            }
            System.out.println("After finally");
      }
    }
    ```

    Plaintext

    ```
    Before try
    Inside try
    Inside finally
    After finally
    ```

  - **Exception Caught:** `try` -> `catch` -> `finally` -> Code after `finally`

    Java

    ```
    public class Main {
      public static void main(String[] args) {
            System.out.println("Before try");
            try{
                System.out.println("Inside try, Before Exception");
                float f = 100/0; // Exception occurs
                System.out.println("Inside try, After Exception");
    // Skipped
            }catch (Exception e){
                System.out.println("Inside catch"); // Executed
            }finally{
                System.out.println("Inside finally"); // Executed
            }
            System.out.println("After finally"); // Executed
      }
    }
    ```

    Plaintext

    ```
    Before try
    Inside try, Before Exception
    Inside catch
    Inside finally
    After finally
    ```

`try-finally` block

- A `try` can be followed by `finally` without a `catch`.

Java

```
try {
    // Code that might throw
} finally {
    // Cleanup code
}
```

- If an exception occurs in `try`, `finally` executes, then the exception is re-thrown to the caller/JVM.

Java

```
public class Main {
  public static void main(String[] args) {
        try{
            System.out.println("Inside try");
            float f = 100/0; // Exception occurs
        }finally{
            System.out.println("Inside finally"); // Executed
        }
        // Code here is NOT reached because exception is re-thrown
  }
}
```

Plaintext

```
Inside try
Inside finally
Exception in thread "main" java.lang.ArithmeticException: /
by zero
    at Main.main(Main.java:5)
```

`try-catch` block

- A `try` can be followed by `catch` without a `finally`.

Java

```
try {
    // Code that might throw
} catch (Exception e) {
    // Handle exception
}
```

Nested `try-catch-finally`

- Allowed within `try`, `catch`, or `finally` blocks.

Plaintext

```
// Example structure (simplified)
try {
    // Outer try code
    try {
        // Inner try code
    } catch (InnerException e) {
        // Inner catch
    }
} catch (OuterException e) {
    // Outer catch
} finally {
    // Outer finally
}
```

Multiple `catch` blocks

- A single `try` block can have multiple `catch` blocks for different exception types.
- **Ordering Rules:**
    1. **No Inheritance:** If catch exception types are unrelated, order doesn't matter.

       Java

       ```
       try{ /* ... */
       }catch(ArithmeticException e){ /* ... */
       }catch(NullPointerException e){ /* ... */
       }catch(ArrayIndexOutOfBoundsException e){ /* ... */
       }
       // Valid in any order
       ```

    2. **With Inheritance:** Catch blocks must be ordered from most specific (subclass) to least specific (superclass).

       Java

       ```
       try{ /* ... */
       }catch(ArithmeticException e){ /* Handle specific */
       }catch(RuntimeException e){    /* Handle less specific */
       }catch(Exception e){         /* Handle most general */
       }
       // Status: Valid (Specific to General)

       try{ /* ... */
       }catch(Exception e){         /* Most general first */
       }catch(RuntimeException e){    /* Subclass is unreachable */
       }catch(ArithmeticException e){ /* Subclass is unreachable */
       }
       // Status: Invalid (Compile-time error: Unreachable catch
       block)
       ```

    3. **Pure Checked Exceptions:** A `catch` block for a pure checked exception requires the corresponding `try` block (or calling method) to be able to throw that *specific* checked exception.

       Java

       ```
       EX:
       ```

```
try{
    throw new ArithmeticException(); // Throws RuntimeException
(Unchecked)
}catch(ArithmeticException e){ /* OK */
}catch(IOException e){ // IOException is Checked. Try block
doesn't throw IOException.
}catch(NullPointerException e){ /* OK */
}
// Status: Invalid (IOException catch is unreachable/compiler
error)
```

Java

```
EX:
try{
    throw new IOException(); // Throws IOException (Checked)
}catch(ArithmeticException e){ /* OK */
}catch(IOException e){ // IOException is thrown and caught
here.
}catch(NullPointerException e){ /* OK */
}
// Status: Valid (Checked exception is thrown and caught)
```

Custom Exceptions / User Defined Exceptions:

—----------

These Exceptions are defined by the developers as per their

application requirements.

To prepare User defined exceptions we have to use the following

steps.

1. Prepare User

Defined Exception class.

2. In Java

applications raise and handle the User defined exception.

Prepare User defined exception:

—----------

1. Declare an user

defined class.

2. Extend

java.lang.Exception to the User defined class.

3. In user defined

class declare a String parameterized constructor.

4. Inside the user

defined constructor, access the superclass constructor by using super keyword.

EX:

```
class MyException extends Exception{
    public MyException(String
description){
        super(description);
    }
}
```

In the above code super(description) is able to provide the user

defined exception description to the Exception class in order to set user

defined exception description to the printStackTrace() method, toString() and

getMessage() method.

In java applications raise and handle the user defined exception:

—--

To raise the User defined exception we will use throw keyword.

To handle the user defined exception we will use try-catch-finally

block.

EX:

Account.java

```
package com.durgasoft.entity;

public class Account {
  private String accNo;
  private String accName;
  private String accType;
  private long accBalance;

  public Account(String accNo,String accName, String accType, long accBalance) {
      this.accNo =accNo;
      this.accName =accName;
      this.accType =accType;
      this.accBalance =accBalance;
  }

  public String getAccNo() {
      return accNo;
```

```
 }

 public void setAccNo(String accNo)
{
     this.accNo =accNo;
 }

 public String getAccName() {
     return accName;
 }

 public void setAccName(String accName)
{
     this.accName =accName;
 }

 public String getAccType() {
     return accType;
 }

 public void setAccType(String accType)
{
     this.accType =accType;
 }
```

```
  public long getAccBalance() {
      return accBalance;
  }

  public void setAccBalance(long accBalance) {
      this.accBalance =accBalance;
  }
}
```

InsufficientFundsException.java

```
package com.durgasoft.exception;

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

Transaction.java

```
package com.durgasoft.entity;

import com.durgasoft.exception.InsufficientFundsException;

public class Transaction {
  private String transactionId;
```

```
  public Transaction(String transactionId) {
      this.transactionId =transactionId;
  }

  public void withdraw(Account account,int wdAmount)
{
      try{
          System.out.println("Transaction
Details");
          System.out.println("--------------------------");
          System.out.println("Transaction
Id      : " + transactionId);
          System.out.println("Account
Number      : "+account.getAccNo());
          System.out.println("Account
Name        : "+account.getAccName());
          System.out.println("Account
Type        : "+account.getAccType());
          System.out.println("Transaction
Type    : WITHDRAW");
          System.out.println("Withdraw
Amount     : "+wdAmount);
          if(wdAmount >account.getAccBalance()){
              System.out.println("Total
```

```
Balance       : "+account.getAccBalance());
                System.out.println("Transaction
Status  : FAILED");
                throw new InsufficientFundsException("Insufficient
Funds in the Account");
            }else{
                account.setAccBalance(account.getAccBalance() - wdAmount);
                System.out.println("Total
Balance       : "+account.getAccBalance());
                System.out.println("Transaction
Status  : SUCCESS");
            }
        }catch (InsufficientFundsException e){
            System.out.println("Reason    : "+e.getMessage());
        }finally{
            System.out.println("***********Thank
you, Visit Again***********");
        }
    }
}
```

Main.java

```
import com.durgasoft.entity.Account;
import com.durgasoft.entity.Transaction;
```

```
public class Main {
  public static void main(String[] args) {
      Account account1 = new Account("abc123", "Durga", "Savings", 20000);
      Transaction transaction1 = new Transaction("76594757595dja76");
      transaction1.withdraw(account1, 10000);
      System.out.println();

      Account account2 = new Account("xyz123", "Anil", "Savings", 10000);
      Transaction transaction2 = new Transaction("4532775759abc76");
      transaction2.withdraw(account2, 20000);

  }
}
```

```
Transaction Details
-------------------------
Transaction Id      :
76594757595dja76
Account Number      : abc123
Account Name        : Durga
Account Type        : Savings
Transaction Type    : WITHDRAW
Withdraw Amount     : 10000
Total Balance       : 10000
Transaction Status  : SUCCESS
```

**********Thank you, Visit Again**********

Transaction Details

Transaction Id :

4532775759abc76

Account Number : xyz123

Account Name : Anil

Account Type : Savings

Transaction Type : WITHDRAW

Withdraw Amount : 20000

Total Balance : 10000

Transaction Status : FAILED

Reason : Insufficient Funds

in the Account

**********Thank you, Visit Again**********

Java 7 version Features in Exception Handling:

—----------

1. Multi-Catch

block

2. try-with-resources

Multi-Catch block:

—----------

Up to JAVA 6 version, we are able to provide only one Exception class name in a catch block, but from JAVA 7 version onward, it is possible to provide more than one Exception class name in a single catch block that is Multi catch block.

Syntax:

```
try{
}catch(Exception-1 | Exception-2 | … | Exception-n refVar){
}
```

The above catch block is able to catch either of the specified exceptions from the try block.
In the above syntax, Exception-1, Exception-2,... Exception-n must not have inheritance relation otherwise the compiler will raise an exception.

EX:

```
import java.util.Date;

public class Main {
  public static void main(String[] args) {
      try{
          //float f = 100/0;
          /*Date d = null;
          System.out.println(d.toString());*/
          int[] ints = {10,20,30,40,50};
```

```
            System.out.println(ints[10]);
        }catch (ArithmeticException | NullPointerException |ArrayIndexOutOfBoundsException
e){
            e.printStackTrace();
        }
    }
}
```

try-with-resources:

—----------

In general, in java applications we will use some resources like streams,

Database connections, Network connections,....

If we perform operations with these resources we may get exceptions,

here to handle the exceptions if we use try-catch-finally then we have to use

the following conventions.

1. Declare the

resources before the try block.

2. Create the

resources inside the try block.

3. Close the

resources inside the finally block.

EX:

```
BufferedReader br = null;
Connection con = null;
```

```
Socket s = null;
try{
    br = new BufferedReader(new
InputStreamReader(System.in));
    con=DriverManager.getConnection(---,---,---);
    s = con.createStatement();
    —---
}catch(ExceptionName refVar){
    —--
}finally{
    try{
        br.close();
        con.close();
        s.close();
    }catch(ExceptionName
refVar){

    }
}
```

If we use the above conventions to manage the resources we are able to get the following problems.

1. Developers must remember to close the resources, it is not guaranteed.

2. close() methods

are raising Exceptions like IOException, SQLException,..., it requires try-catch-finally inside the finally block, it is a code confusion.

To overcome the above problems, JAVA has provided a feature like try-with-resources.

Syntax:

```
try(Resource-1; Resource-2;...;Resource-n;){
    —---
}catch(ExceptionName refVar){
    —----
}
```

In the above syntax, all the resources must be autocloseable resources, that is the resources must implement or extend java.lang.AutoCloseable marker interface.

In try-with-resources, all the resources are closed automatically when flow of execution is coming out from the try block.

In try-with-resources, all the resource reference variables are final variables by default.

EX:

```
try(
    BufferedReader br = new
BufferedReader(new
InputStreamReader(System.in));
Connection con = DriverManager.getConnection(---,---,---);
Socket s = new Socket(---,--);
){
}catch(Exception e){
    e.printStackTrace();
}
```

==========

Okay, I have processed the additional text about Custom Exceptions and Java 7 features and integrated it into the comprehensive notes.

Here is the updated document, formatted for easy copy-pasting:

Java Exception Handling Notes

This document provides comprehensive notes on Error and Exception handling in Java, based on the provided text.

Error vs. Exception

Q)What is the difference between Error and Exception?
——---------- Ans: ——---

- **Error:** A problem preventing program execution, typically unrecoverable programmatically.
    - Generally two types:
        1. Compilation Errors
        2. Runtime Errors
    - **Compilation Error:** Identified by the compiler during compilation.
        1. Types of Compilation Errors:
            - **Lexical Errors:** Mistakes in tokens (e.g., misspelled keywords like `nit` instead of `int`).

              Plaintext

              ```
              EX:  int i = 10;  ---> Valid
                   nit i = 20;  ---> Lexical Error
              ```

            - **Syntax Errors:** Mistakes in language syntax (e.g., missing semicolon).

              Plaintext

              ```
              EX:  int i = 10;  -----> Valid
                   int i = 20  -----> Syntax Error
                   int = i 30;  -----> Syntax Errors
              ```

            - **Semantic Errors:** Meaningless statements, incompatible operations (e.g., adding an integer and a boolean).

              Plaintext

              ```
              EX:  int a = 10; int b = 20; int c = a + b;  --->
              Valid

              EX:  int a = 10; boolean b = true; char c = a + b;
              ----> Semantic Error
              ```

        2. Note: Languages may have additional compilation errors specific to their conventions (e.g., Unreachable Statement in Java).

- **Runtime Errors:** Problems occurring during application runtime that cannot be fixed programmatically.
    1. Examples: Unavailability of IO components, JVM Internal Problem, Insufficient Main Memory, StackOverflowError.

    - **Exception:** Problems occurring at runtime that *can* be handled programmatically.
- Examples: ArithmeticException, NullPointerException.

Java

```
int a = 100;
int b = 0;
if(b != 0){
    float f = a / b;
}

Date d = null;
if(d != null){
    System.out.println(d.toString());
}
```

—----------

Understanding Exceptions and Termination

- **Exception Definition:** An unexpected runtime event causing abnormal termination. Can originate from various sources like user input, database, network.
- **Application Termination Types:**
    1. **Smooth Termination:** Program ends normally at the end.
    2. **Abnormal Termination:** Program stops unexpectedly in the middle.
- In Java, Exceptions typically cause Abnormal Termination, leading to potential issues like OS crashes, network hangs, database collapses, server downtime.
- **Goal of Exception Handling:** To avoid abnormal termination and achieve smooth termination.

Java Robustness

Java is robust due to:

1. **Good Memory Management:** Dynamic Heap memory management system.
2. **Good Exception Handling:** Rich predefined libraries to handle frequent exceptions.
    - Supports defining and handling custom exceptions.

Types of Exceptions in Java

1. **Predefined Exceptions:** Defined by Java, represented by classes.
2. **User defined Exceptions:** Custom exceptions created by developers.

Predefined Exceptions

- Represented by predefined classes.
- Two main types: Checked and Unchecked.

Q)What is the difference between Checked Exception and Unchecked Exception? —---------- Ans: —---

- **Checked Exception:** Recognized by the compiler at compilation time. (Exception actually occurs at runtime).

- **Unchecked Exception:** Recognized by the JVM at runtime, not by the compiler at compilation time.
    - Examples: `RuntimeException` and its subclasses, `Error` and its subclasses. All other exception classes are typically checked.
      —----------

- **Subtypes of Checked Exceptions:**

Q) What is the difference between a Partially Checked exception and Pure Checked Exception? —---------- Ans: —-----

- **Pure Checked Exception:** A checked exception whose child classes are *only* checked exceptions.
    - Example: `IOException`
- **Partially Checked Exception:** A checked exception with *at least one* unchecked child exception.
    - Examples: `Throwable`, `Exception` —----------

Common Predefined Exceptions Overview

(Examples include code and typical output/messages)

1. `java.lang.ArithmeticException`: Division by zero.

   Java

   ```
   public class Main {
     public static void main(String[] args){
          int a = 100;
          int b = 0;
          float f = a/b; // Exception occurs here
     }
   }
   ```

   Plaintext

   ```
   Exception in thread "main" java.lang.ArithmeticException: /
   by zero
       at Main.main(Main.java:5)
   ```

    - Message Parts: Name, Description, Location.

2. `java.lang.NullPointerException`: Accessing member on a null reference.

Java

```
import java.util.Date;

public class Main {
  public static void main(String[] args){
       Date date = null;
       System.out.println(date.toString()); // Exception occurs here
  }
}
```

Plaintext

```
Exception Name : java.lang.NullPointerException
Exception Description : Cannot invoke "java.util.Date.toString()"
because "date" is null
Exception Location : Main.java: 6
```

3. `java.lang.ArrayIndexOutOfBoundsException`: Accessing array element outside bounds.

Java

```
public class Main {
  public static void main(String[] args){
       int[] ints = {10,20,30,40,50};
       System.out.println(ints[2]);  // Valid
       System.out.println(ints[10]); // Exception occurs here
  }
}
```

Plaintext

```
30
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: Index 10 out of bounds for
length 5
    at Main.main(Main.java:7)
```

4. `java.lang.StringIndexOutOfBoundsException`: String operation with index outside bounds.

Java

```
public class Main {
  public static void main(String[] args){
       String data = "Durgasoft";
       System.out.println(data.charAt(5));  // Valid
       System.out.println(data.charAt(20)); // Exception occurs here
  }
}
```

Plaintext

```
Exception Name : java.lang.StringIndexOutOfBoundsException
Exception Description : Index 20 out of bounds for length 9
Exception Location : Main.java:7
```

5. `java.lang.ClassNotFoundException`: Class.forName() cannot find class.

Java

```
class A{
  static{
       System.out.println("Class A Loading");
  }
}
public class Main {
  public static void main(String[] args)throws Exception{
       Class.forName("B"); // Exception occurs here if 'B' not found
  }
}
```

Plaintext

```
Exception Name : java.lang.ClassNotFoundException
Exception Description : B
Exception Location : Main.java: 7
```

6. `java.lang.InstantiationException`: Class.newInstance() fails (e.g., no 0-arg constructor).

Java

```
class A{
  public A(int i){
       System.out.println("A-con");
  }
}
public class Main {
  public static void main(String[] args)throws Exception{
       Class cls = Class.forName("A");
       cls.newInstance(); // Exception occurs here (no 0-arg
constructor)
  }
}
```

Plaintext

```
Exception Name : java.lang.InstantiationException
Exception Description : A
Exception Location : Main.java: 9
```

7. `java.lang.IllegalAccessException`: Class.newInstance() tries to access private constructor.

Java

```
class A{
```

```
  private A(){
       System.out.println("A-con");
  }
}
public class Main {
  public static void main(String[] args)throws Exception{
       Class cls = Class.forName("A");
       cls.newInstance(); // Exception occurs here (constructor is
private)
  }
}
```

Plaintext

```
Exception Name : java.lang.IllegalAccessException
Exception Description : class Main cannot access a member of
  class A with modifiers "private"
Exception Location : Main.java: 9
```

8. `java.lang.ClassCastException`: Invalid object casting (e.g., superclass object to subclass reference).

Java

```
class A{ }
class B extends A{ }
public class Main {
  public static void main(String[] args){
       A a = new A();
       B b = (B) a; // Exception occurs here (A object cannot be cast
to B)
  }
}
```

Plaintext

```
Exception Name : java.lang.ClassCastException
Exception Description : class A cannot be cast to class B
Exception Location : Main.java: 10
```

9. `java.io.FileNotFoundException`: File not found when using stream/reader.

Java

```
import java.io.FileInputStream;

public class Main {
  public static void main(String[] args)throws Exception{
       FileInputStream fis =new
FileInputStream("E:/abc/xyz/welcome.txt"); // Exception if file not
found
  }
}
```

Plaintext

```
Exception Name: java.io.FileNotFoundException
Exception Description: E:/abc/xyz/welcome.txt (No such file or
 directory)
 Exception Location : Main.java: 5
```

Keywords for Exception Handling

In Java, there are two main ways to handle exceptions:

1. `throws` keyword
2. `try-catch-finally` block

`throw` keyword

- Purpose: To explicitly raise an exception object.
- Usage: Inside method body.
- Allows: Only one exception object at a time.
- Syntax: `throw new ExceptionName([ParamValues]);`

Java

```
class Student{
  String sid, sname, saddr;
  float smarksInPercentage;

  public Student(String sid, String sname, String saddr, float
smarksInPercentage) {
       this.sid = sid;
       this.sname =sname;
       this.saddr =saddr;
       this.smarksInPercentage =smarksInPercentage;
  }

  public void getStudentDetails(){
       System.out.println("Student Details");
       System.out.println("-------------------------");
       System.out.println("Student Id           : "+sid);
       System.out.println("Student Name         : "+sname);
       System.out.println("Student Address      : "+saddr);
       System.out.println("Student Marks In Percentage :
"+smarksInPercentage);
       if(smarksInPercentage < 0 || smarksInPercentage > 100){
           throw new RuntimeException("Invalid Student Marks, Provide
Student Marks between 0 to 100");
       }else{
           System.out.println("Status             : Student Marks are
Valid");
       }
  }
}
public class Main {
  public static void main(String[] args)throws Exception{

       Student student1 = new Student("S-111", "Durga", "Hyd", 78.0f);
       student1.getStudentDetails();
       System.out.println();

       Student student2 = new Student("S-222", "Ramana", "Hyd", 150.0f);
```

```
        student2.getStudentDetails(); // This call will throw the exception
    }
}
```

Plaintext

```
Student Details
-------------------------
Student Id           : S-111
Student Name         : Durga
Student Address      : Hyd
Student Marks In Percentage : 78.0
Status             : Student Marks are Valid

Transaction Details
-------------------------
Transaction Id       : 76594757595dja76
Account Number       : abc123
Account Name         : Durga
Account Type         : Savings
Transaction Type     : WITHDRAW
Withdraw Amount      : 10000
Total Balance        : 10000
Transaction Status   : SUCCESS
***********Thank you, Visit Again***********

Transaction Details
-------------------------
Transaction Id       : 4532775759abc76
Account Number       : xyz123
Account Name         : Anil
Account Type         : Savings
Transaction Type     : WITHDRAW
Withdraw Amount      : 20000
Total Balance        : 10000
Transaction Status   : FAILED
Reason             : Insufficient Funds in the Account
***********Thank you, Visit Again***********
```

(Note: The output section in the original text for the `throw` example seemed to combine outputs from different examples. I've included the relevant parts as provided.)

`throws` keyword

- Purpose: To declare that a method might throw an exception, bypassing it to the caller. Not an actual handler.
- Usage: In method signature/prototype.
- Allows: One or more exception names (comma-separated).

Java

```
import java.io.IOException;

class A{
  void add()throws Exception { // Declares it might throw Exception (or its
subclasses)
        concat();
  }
  void concat()throws IOException { // Declares it might throw IOException
```

```
       throw new
IOException("My OWN IOException"); // Throws IOException
  }
}
class Test {
   public static void main(String[] args)throws Throwable { // Declares it
might throw Throwable (or its subclasses)
       A a = new A();
       a.add(); // Calls add(), which might throw IOException. main
declares it can handle Throwable (a superclass).
   }
}
```

Plaintext

```
nagoorn@Nagoors-MacBook-Pro COREJAVA-830 % javac Test.java
nagoorn@Nagoors-MacBook-Pro COREJAVA-830 % java Test
Exception in thread "main" java.io.IOException: My OWN
IOException
         at A.concat(Test.java:8)
         at A.add(Test.java:5)
         at Test.main(Test.java:14)
```

Comparison: `throw` vs. `throws`

Q) What are the differences between ‘throws’ and ‘throw’ keywords? —---------- Ans: —--- See table below for comparison. —----------

+-------------------+-----------------------------------+-----------------------------------+ | Feature | `throw` keyword | `throws` keyword | +=====================+=====================================+=======
==+ | **Purpose** | Explicitly raise an exception | Declare exceptions a method might | | | object. | throw, delegating handling. | +-------------------+-----------------------------------+-----------------------------------+ | **Usage Location** | Inside method body/implementation | In method signature (after params) | +--------------------+-----------------------------------+-----------------------------------+ | **Operand** | Single exception object | One or more exception class names | | | (`throw new MyException();`) | (`throws Exception1, Exception2`) | +--------------------
+-----------------------------------+-----------------------------------+

`try-catch-finally` Block

- Used for exact exception handling at the point of occurrence.
- **Structure:**

  Plaintext

  ```
  try {
      // Code that might throw exceptions
  } catch (ExceptionType1 e1) {
      // Handle ExceptionType1
  } catch (ExceptionType2 e2) {
      // Handle ExceptionType2
  ```

```
} finally {
    // Code that *always* executes
}
```

`try` block

- Contains code potentially throwing exceptions.
- Only "doubtful" code needs to be in `try`.
- If exception occurs, remaining `try` code skipped, jump to `catch`.
- If no exception, `try` completes, jump to `finally` (skipping `catch`).

`catch` block

- Catches exceptions from the preceding `try`.
- Handles the exception (e.g., display info, logging, recovery).
- Executed *only* if an exception matching the type is thrown in `try`.
- **Ways to display Exception Details:**
  1. `e.printStackTrace()`: Prints full stack trace (name, description, location).
  2. `e.toString()`: Prints exception name and description (default for `System.out.println(e)`).
  3. `e.getMessage()`: Prints only the exception description.

Java

```
class Test {
  public static void main(String[] args){
       try{
           int a = 100;
           int b = 0;
           float f = a/b; // ArithmeticException
       }catch(Exception e){
           e.printStackTrace(); // Full stack trace
           System.out.println();
           System.out.println(e); // toString() output
           System.out.println();
           System.out.println(e.getMessage()); // Message only
       }finally{
           // finally block content
       }
   }
}
```

Plaintext

```
nagoorn@Nagoors-MacBook-Pro COREJAVA-830 % javac Test.java
nagoorn@Nagoors-MacBook-Pro COREJAVA-830 % java Test
java.lang.ArithmeticException: / by zero
         at Test.main(Test.java:7)

java.lang.ArithmeticException: / by zero

/ by zero
```

`finally` block

- Contains code guaranteed to execute regardless of whether an exception occurred or was caught.
- Purpose: Resource cleanup (closing files, connections, etc.).
- **Execution Flow Examples:**
  - **No Exception:** `try` -> `finally` -> Code after `finally`

    Java

    ```
    public class Main {
      public static void main(String[] args) {
            System.out.println("Before try");
            try{
                System.out.println("Inside try");
            }catch (Exception e){
                System.out.println("Inside catch");
            }finally{
                System.out.println("Inside finally");
            }
            System.out.println("After finally");
      }
    }
    ```

    Plaintext

    ```
    Before try
    Inside try
    Inside finally
    After finally
    ```

  - **Exception Caught:** `try` -> `catch` -> `finally` -> Code after `finally`

    Java

    ```
    public class Main {
      public static void main(String[] args) {
            System.out.println("Before try");
            try{
                System.out.println("Inside try, Before Exception");
                float f = 100/0; // Exception occurs
                System.out.println("Inside try, After Exception");
    // Skipped
            }catch (Exception e){
                System.out.println("Inside catch"); // Executed
            }finally{
                System.out.println("Inside finally"); // Executed
            }
            System.out.println("After finally"); // Executed
      }
    }
    ```

    Plaintext

```
Before try
Inside try, Before Exception
Inside catch
Inside finally
After finally
```

`try-finally` block

- A `try` can be followed by `finally` without a `catch`.

Java

```
try {
    // Code that might throw
} finally {
    // Cleanup code
}
```

- If an exception occurs in `try`, `finally` executes, then the exception is re-thrown to the caller/JVM.

Java

```
public class Main {
  public static void main(String[] args) {
       try{
           System.out.println("Inside try");
           float f = 100/0; // Exception occurs
       }finally{
           System.out.println("Inside finally"); // Executed
       }
       // Code here is NOT reached because exception is re-thrown
  }
}
```

Plaintext

```
Inside try
Inside finally
Exception in thread "main" java.lang.ArithmeticException: /
by zero
    at Main.main(Main.java:5)
```

`try-catch` block

- A `try` can be followed by `catch` without a `finally`.

Java

```
try {
    // Code that might throw
} catch (Exception e) {
    // Handle exception
}
```

Nested `try-catch-finally`

- Allowed within `try`, `catch`, or `finally` blocks.

  Plaintext

```
// Example structure (simplified)
try {
    // Outer try code
    try {
        // Inner try code
    } catch (InnerException e) {
        // Inner catch
    }
} catch (OuterException e) {
    // Outer catch
} finally {
    // Outer finally
}
```

Multiple `catch` blocks

- A single `try` block can have multiple `catch` blocks for different exception types.
- **Ordering Rules:**
  1. **No Inheritance:** If catch exception types are unrelated, order doesn't matter.

     Java

```
try{ /* ... */
}catch(ArithmeticException e){ /* ... */
}catch(NullPointerException e){ /* ... */
}catch(ArrayIndexOutOfBoundsException e){ /* ... */
}
// Valid in any order
```

  2. **With Inheritance:** Catch blocks must be ordered from most specific (subclass) to least specific (superclass).

     Java

```
try{ /* ... */
}catch(ArithmeticException e){ /* Handle specific */
}catch(RuntimeException e){    /* Handle less specific */
}catch(Exception e){         /* Handle most general */
}
// Status: Valid (Specific to General)

try{ /* ... */
}catch(Exception e){         /* Most general first */
}catch(RuntimeException e){    /* Subclass is unreachable */
}catch(ArithmeticException e){ /* Subclass is unreachable */
}
// Status: Invalid (Compile-time error: Unreachable catch
block)
```

3. **Pure Checked Exceptions:** A `catch` block for a pure checked exception requires the corresponding `try` block (or calling method) to be able to throw that *specific* checked exception.

Java

```
EX:
try{
    throw new ArithmeticException(); // Throws RuntimeException
(Unchecked)
}catch(ArithmeticException e){ /* OK */
}catch(IOException e){ // IOException is Checked. Try block
doesn't throw IOException.
}catch(NullPointerException e){ /* OK */
}
// Status: Invalid (IOException catch is unreachable/compiler
error)
```

Java

```
EX:
try{
    throw new IOException(); // Throws IOException (Checked)
}catch(ArithmeticException e){ /* OK */
}catch(IOException e){ // IOException is thrown and caught
here.
}catch(NullPointerException e){ /* OK */
}
// Status: Valid (Checked exception is thrown and caught)
```

Custom Exceptions (User Defined Exceptions)

- Exceptions created by developers based on application needs.

- **Steps to Prepare:**
    1. Declare a custom class.
    2. Extend `java.lang.Exception` (or a subclass like `RuntimeException`).
    3. Declare a `String` parameterized constructor.
    4. Inside the constructor, call the superclass constructor using `super(description);` to set the exception message accessible by `printStackTrace()`, `toString()`, and `getMessage()`.

Java

```
EX:
class MyException extends Exception{
     public MyException(String description){
         super(description);
     }
}
```

- **Raising and Handling Custom Exceptions:**

    1. Raise using the `throw` keyword.

2. Handle using `try-catch-finally`.

- **Example:** (Bank Account Withdrawal Scenario)

1. **Account.java**

Java

```
package com.durgasoft.entity;

public class Account {
  private String accNo;
  private String accName;
  private String accType;
  private long accBalance;

  public Account(String accNo,String accName, String accType,
long accBalance) {
        this.accNo =accNo;
        this.accName =accName;
        this.accType =accType;
        this.accBalance =accBalance;
  }

  public String getAccNo() {
        return accNo;
  }

  public void setAccNo(String accNo)
{
        this.accNo =accNo;
  }

  public String getAccName() {
        return accName;
  }

  public void setAccName(String accName)
{
        this.accName =accName;
  }

  public String getAccType() {
        return accType;
  }

  public void setAccType(String accType)
{
        this.accType =accType;
  }

  public long getAccBalance() {
        return accBalance;
  }

  public void setAccBalance(long accBalance) {
        this.accBalance =accBalance;
  }
}
```

2. **InsufficientFundsException.java** (Custom Exception)

Java

```
package com.durgasoft.exception;

public class InsufficientFundsException extends Exception { //
Extends Exception (Checked Exception)
  public InsufficientFundsException(String message) {
       super(message); // Pass message to parent Exception
class
  }
}
```

3. **Transaction.java**

Java

```
package com.durgasoft.entity;

import com.durgasoft.exception.InsufficientFundsException;

public class Transaction {
  private String transactionId;

  public Transaction(String transactionId) {
       this.transactionId =transactionId;
  }

  public void withdraw(Account account,int wdAmount)
{
       try{ // Code that might throw InsufficientFundsException
           System.out.println("Transaction Details");
           System.out.println("-------------------------");
           System.out.println("Transaction Id       : " +
transactionId);
           System.out.println("Account Number       :
"+account.getAccNo());
           System.out.println("Account Name         :
"+account.getAccName());
           System.out.println("Account Type         :
"+account.getAccType());
           System.out.println("Transaction Type     :
WITHDRAW");
           System.out.println("Withdraw Amount      :
"+wdAmount);
           if(wdAmount > account.getAccBalance()){
               System.out.println("Total Balance        :
"+account.getAccBalance());
               System.out.println("Transaction Status   :
FAILED");
               throw new
InsufficientFundsException("Insufficient Funds in the
Account"); // Raise custom exception
           }else{
               account.setAccBalance(account.getAccBalance() -
wdAmount);
               System.out.println("Total Balance        :
"+account.getAccBalance());
```

```
                System.out.println("Transaction Status   :
SUCCESS");
            }
        }catch (InsufficientFundsException e){ // Catch the
custom exception
            System.out.println("Reason             :
"+e.getMessage()); // Display message
        }finally{
            System.out.println("***********Thank you, Visit
Again***********"); // Always executes
        }
    }
}
```

4. **Main.java**

Java

```
import com.durgasoft.entity.Account;
import com.durgasoft.entity.Transaction;

public class Main {
  public static void main(String[] args) {
        Account account1 = new Account("abc123", "Durga",
"Savings", 20000);
        Transaction transaction1 = new
Transaction("76594757595dja76");
        transaction1.withdraw(account1, 10000); // Successful
withdrawal

        System.out.println();

        Account account2 = new Account("xyz123", "Anil",
"Savings", 10000);
        Transaction transaction2 = new
Transaction("4532775759abc76");
        transaction2.withdraw(account2, 20000); // Insufficient
funds, exception caught
  }
}
```

5. **Output:**

Plaintext

```
Transaction Details
--------------------------
Transaction Id       : 76594757595dja76
Account Number       : abc123
Account Name         : Durga
Account Type         : Savings
Transaction Type     : WITHDRAW
Withdraw Amount      : 10000
Total Balance        : 10000
Transaction Status   : SUCCESS
***********Thank you, Visit Again***********

Transaction Details
--------------------------
```

```
Transaction Id       : 4532775759abc76
Account Number       : xyz123
Account Name         : Anil
Account Type         : Savings
Transaction Type     : WITHDRAW
Withdraw Amount      : 20000
Total Balance        : 10000
Transaction Status   : FAILED
Reason             : Insufficient Funds in the Account
***********Thank you, Visit Again***********
```

Java 7 Exception Handling Features

Introduced in Java 7:

1. Multi-Catch block
2. try-with-resources

Multi-Catch block

- Allows catching multiple, unrelated exception types in a single `catch` block.
- Reduces code duplication when handling several exceptions similarly.
- Syntax:

Java

```
try {
    // Code that might throw Exception-1, Exception-2, ...
} catch (Exception-1 | Exception-2 | ... | Exception-n refVar) {
    // Handle any of the listed exceptions
}
```

- **Constraint:** The exception types combined with `|` must *not* have an inheritance relationship with each other.

Java

```
EX:
import java.util.Date;

public class Main {
  public static void main(String[] args) {
       try{
           // Uncomment one line at a time to see different
exceptions caught
           //float f = 100/0; // ArithmeticException
           //Date d = null; System.out.println(d.toString()); //
NullPointerException
           int[] ints = {10,20,30,40,50};
           System.out.println(ints[10]); //
ArrayIndexOutOfBoundsException
       }catch (ArithmeticException | NullPointerException |
ArrayIndexOutOfBoundsException e){
```

```
            // Catches any of the three specified exceptions
            e.printStackTrace();
        }
    }
}
```

try-with-resources

- Simplifies resource management (streams, connections, etc.) by ensuring resources are automatically closed.

- Addresses issues with manual closing in `finally`:
    - Forgetting to close resources.
    - Handling exceptions thrown by `close()` methods within the `finally` block, leading to nested `try-catch-finally`.

- **Concept:** Resources declared in the `try` statement header are automatically closed when the `try` block is exited (normally or due to an exception).

- **Requirement:** Resources must implement the `java.lang.AutoCloseable` interface.

- Resources declared in the `try` header are implicitly `final`.

- **Syntax:**

  Java

  ```
  try (Resource-1; Resource-2; ...; Resource-n) {
      // Use the resources
  } catch (ExceptionName refVar) {
      // Handle exceptions
  }
  // Resources are automatically closed here
  ```

- **Comparison:**
    - **Old way (pre-Java 7):** Manual closing in `finally` block.

      Java

      ```
      BufferedReader br = null;
      Connection con = null;
      Socket s = null;
      try{
          br = new BufferedReader(new InputStreamReader(System.in));
          con = DriverManager.getConnection(---,---,---); //
      Placeholder for connection details
          // s = con.createStatement(); // Typo? Statement is from
      Connection, not Socket usually. Corrected based on type Socket
      s.
          s = new Socket("host", 1234); // Example Socket creation
          // Use resources...
      }catch(Exception e){ // Catch relevant exceptions
          e.printStackTrace();
      }finally{ // Mandatory cleanup
      ```

```
    try{
        if (br != null) br.close(); // Need null checks
        if (con != null) con.close();
        if (s != null) s.close();
    }catch(IOException | SQLException e){ // Need catch for
close exceptions
        e.printStackTrace(); // Or handle appropriately
    }
}
```

- **New way (Java 7+ try-with-resources):** Automatic closing.

  Java

```
EX:
try(
     // Resources declared here must be AutoCloseable
     BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
     Connection con = DriverManager.getConnection(---,---,---);
// Placeholder
     Socket s = new Socket("host", 1234); // Example
){
    // Use resources (br, con, s are available and effectively
final)
    // No explicit close() calls needed
}catch(Exception e){ // Catch exceptions from try body or
resource closing
    e.printStackTrace();
}
// Resources are automatically closed after the try block
finishes or throws an exception
```