
Java Exceptions

In this tutorial, we will learn about exceptions in Java. We will cover errors, exceptions and different types of exceptions in Java.

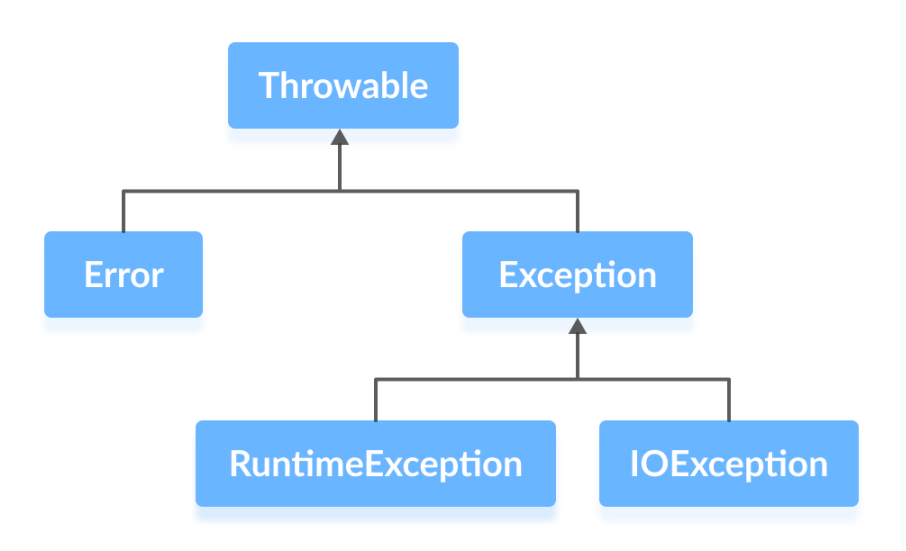
An exception is an unexpected event that occurs during program execution. It affects the flow of the program instructions which can cause the program to terminate abnormally.

An exception can occur for many reasons. Some of them are:

- Invalid user input
- Device failure
- Loss of network connection
- Physical limitations (out of disk memory)
- Code errors
- Opening an unavailable file

Java Exception hierarchy

Here is a simplified diagram of the exception hierarchy in Java.



As you can see from the image above, the `Throwable` class is the root class in the hierarchy.

Note that the hierarchy splits into two branches: Error and Exception.

Errors

Errors represent irrecoverable conditions such as Java virtual machine (JVM) running out of memory, memory leaks, stack overflow errors, library incompatibility, infinite recursion, etc.

Errors are usually beyond the control of the programmer and we should not try to handle errors.

Exceptions

Exceptions can be caught and handled by the program.

When an exception occurs within a method, it creates an object. This object is called the exception object.

It contains information about the exception such as the name and description of the exception and state of the program when the exception occurred.

We will learn how to handle these exceptions in the next tutorial. In this tutorial, we will now focus on different types of exceptions in Java.

Java Exception Types

The exception hierarchy also has two branches: `RuntimeException` and `IOException`.

---

1. RuntimeException

A **runtime exception** happens due to a programming error. They are also known as **unchecked exceptions**.

These exceptions are not checked at compile-time but run-time. Some of the common runtime exceptions are:

- Improper use of an API - `IllegalArgumentException`
- Null pointer access (missing the initialization of a variable) - `NullPointerException`
- Out-of-bounds array access - `ArrayIndexOutOfBoundsException`
- Dividing a number by 0 - `ArithmeticException`

You can think about it in this way. “If it is a runtime exception, it is your fault”.

The `NullPointerException` would not have occurred if you had checked whether the variable was initialized or not before using it.

An `ArrayIndexOutOfBoundsException` would not have occurred if you tested the array index against the array bounds.

---

2. IOException

An `IOException` is also known as a **checked exception**. They are checked by the compiler at the compile-time and the programmer is prompted to handle these exceptions.

Some of the examples of checked exceptions are:

- Trying to open a file that doesn’t exist results in `FileNotFoundException`
- Trying to read past the end of a file

Java Exception Handling

In the tutorial, we will learn about different approaches of exception handling in Java with the help of examples.

In the last tutorial, we learned about Java exceptions. We know that exceptions abnormally terminate the execution of a program.

This is why it is important to handle exceptions. Here's a list of different approaches to handle exceptions in Java.

- try...catch block

- finally block
- throw and throws keyword

---

1. Java try...catch block

The try-catch block is used to handle exceptions in Java. Here's the syntax of `try...catch` block:

```
try {
  // code
}catch(Exception e) {
  // code
}
```

Here, we have placed the code that might generate an exception inside the `try` block. Every `try` block is followed by a `catch` block.

When an exception occurs, it is caught by the `catch` block. The `catch` block cannot be used without the `try` block.

Example: Exception handling using try...catch

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


    try {


      // code that generate exception
      int divideByZero = 5 / 0;
      System.out.println("Rest of code in try block");
    }


    catch (ArithmeticException e) {
      System.out.println("ArithmeticException => " + e.getMessage());
```

```
    }
  }
}
```

Output

```
ArithmeticException => / by zero
```

In the example, we are trying to divide a number by `0`. Here, this code generates an exception.

To handle the exception, we have put the code, `5 / 0` inside the `try` block. Now when an exception occurs, the rest of the code inside the `try` block is skipped.

The `catch` block catches the exception and statements inside the catch block is executed.

If none of the statements in the `try` block generates an exception, the `catch` block is skipped.

---

2. Java finally block

In Java, the `finally` block is always executed no matter whether there is an exception or not.

The `finally` block is optional. And, for each `try` block, there can be only one `finally` block.

The basic syntax of `finally` block is:

```
try {
  //code
}catch (ExceptionType1 e1) {
  // catch block
}finally {
  // finally block always executes
}
```

If an exception occurs, the `finally` block is executed after the `try...catch` block. Otherwise, it is executed after the try block. For each `try` block, there can be only one `finally` block.

Example: Java Exception Handling using finally block

```
class Main {

  public static void main(String[] args) {

    try {

      // code that generates exception

      int divideByZero = 5 / 0;

    }



    catch (ArithmeticException e) {

      System.out.println("ArithmeticException => " + e.getMessage());

    }



    finally {

      System.out.println("This is the finally block");

    }

  }

}
```

Output

```
ArithmeticException => / by zero

This is the finally block
```

In the above example, we are dividing a number by **0** inside the `try` block. Here, this code generates an `ArithmeticException`.

The exception is caught by the `catch` block. And, then the `finally` block is executed.

Note: It is a good practice to use the `finally` block. It is because it can include important cleanup codes like,

- code that might be accidentally skipped by return, continue or break
- closing a file or connection

---

3. Java throw and throws keyword

The Java `throw` keyword is used to explicitly throw a single exception.

When we `throw` an exception, the flow of the program moves from the `try` block to the `catch` block.

Example: Exception handling using Java throw

```
class Main {

  public static void divideByZero() {


    // throw an exception

    throw new ArithmeticException("Trying to divide by 0");

  }


  public static void main(String[] args) {

    divideByZero();

  }

}
```

Output

```
Exception in thread "main" java.lang.ArithmeticException: Trying to divide by 0

        at Main.divideByZero(Main.java:5)

        at Main.main(Main.java:9)
```

In the above example, we are explicitly throwing the `ArithmeticException` using the `throw` keyword.

Similarly, the `throws` keyword is used to declare the type of exceptions that might occur within the method. It is used in the method declaration.

Example: Java throws keyword

```
import java.io.*;

class Main {
```

```
  // declareing the type of exception
  public static void findFile() throws IOException {


    // code that may generate IOException
    File newFile = new File("test.txt");
    FileInputStream stream = new FileInputStream(newFile);
  }


  public static void main(String[] args) {
    try {
      findFile();
    }
    catch (IOException e) {
      System.out.println(e);
    }
  }
}
```

Output

```
java.io.FileNotFoundException: test.txt (The system cannot find the file specified)
```

When we run this program, if the file **test.txt** does not exist, `FileInputStream` throws a `FileNotFoundException` which extends the `IOException` class.

The `findFile()` method specifies that an `IOException` can be thrown. The `main()` method calls this method and handles the exception if it is thrown.

If a method does not handle exceptions, the type of exceptions that may occur within it must be specified in the `throws` clause.

Java try...catch

In this tutorial, we will learn about the try catch statement in Java with the help of examples.

The `try...catch` block in Java is used to handle exceptions and prevents the abnormal termination of the program.

Here's the syntax of a `try...catch` block in Java.

```
try{
  // code
}catch(exception) {
  // code
}
```

The `try` block includes the code that might generate an exception.

The `catch` block includes the code that is executed when there occurs an exception inside the `try` block.

Example: Java try...catch block

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

    try {
      int divideByZero = 5 / 0;
      System.out.println("Rest of code in try block");
    }

    catch (ArithmeticException e) {
      System.out.println("ArithmeticException => " + e.getMessage());
    }
  }
}
```

Output

```
ArithmeticException => / by zero
```

In the above example, notice the line,

```
int divideByZero = 5 / 0;
```

Here, we are trying to divide a number by **zero**. In this case, an exception occurs. Hence, we have enclosed this code inside the `try` block.

When the program encounters this code, `ArithmeticException` occurs. And, the exception is caught by the `catch` block and executes the code inside the `catch` block.

The `catch` block is only executed if there exists an exception inside the `try` block.

> **Note**: In Java, we can use a `try` block without a `catch` block. However, we cannot use a `catch` block without a `try` block.

Java try...finally block

We can also use the `try` block along with a finally block.

In this case, the finally block is always executed whether there is an exception inside the try block or not.

Example: Java try...finally block

```
class Main {

  public static void main(String[] args) {

    try {

      int divideByZero = 5 / 0;

    }


    finally {

      System.out.println("Finally block is always executed");

    }

  }

}
```

Output

```
Finally block is always executed

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

        at Main.main(Main.java:4)
```

In the above example, we have used the `try` block along with the `finally` block. We can see that the code inside the `try` block is causing an exception.

However, the code inside the `finally` block is executed irrespective of the exception.

Java try...catch...finally block

In Java, we can also use the finally block after the `try...catch` block. For example,

```
import java.io.*;

class ListOfNumbers {


  // create an integer array

  private int[] list = {5, 6, 8, 9, 2};


  // method to write data from array to a fila

  public void writeList() {

    PrintWriter out = null;


    try {

      System.out.println("Entering try statement");


      // creating a new file OutputFile.txt

      out = new PrintWriter(new FileWriter("OutputFile.txt"));


      // writing values from list array to Output.txt

      for (int i = 0; i < 7; i++) {

        out.println("Value at: " + i + " = " + list[i]);

      }
```

```
    }

    catch (Exception e) {
      System.out.println("Exception => " + e.getMessage());
    }

    finally {
      // checking if PrintWriter has been opened
      if (out != null) {
        System.out.println("Closing PrintWriter");
        // close PrintWriter
        out.close();
      }

      else {
        System.out.println("PrintWriter not open");
      }
    }

  }
}
class Main {
  public static void main(String[] args) {
    ListOfNumbers list = new ListOfNumbers();
    list.writeList();
  }
}
```

Output

```
Entering try statement
Exception => Index 5 out of bounds for length 5
Closing PrintWriter
```

In the above example, we have created an array named `list` and a file named `output.txt`. Here, we are trying to read data from the array and storing to the file.

Notice the code,

```
for (int i = 0; i < 7; i++) {

  out.println("Value at: " + i + " = " + list[i]);

}
```

Here, the size of the array is `5` and the last element of the array is at `list[4]`. However, we are trying to access elements at `a[5]` and `a[6]`.

Hence, the code generates an exception that is caught by the catch block.

Since the `finally` block is always executed, we have included code to close the `PrintWriter` inside the finally block.

It is a good practice to use finally block to include important cleanup code like closing a file or connection.

Note: There are some cases when a `finally` block does not execute:

- Use of `System.exit()` method
- An exception occurs in the `finally` block
- The death of a thread

Multiple Catch blocks

For each `try` block, there can be zero or more `catch` blocks. Multiple `catch` blocks allow us to handle each exception differently.

The argument type of each `catch` block indicates the type of exception that can be handled by it. For example,

```
class ListOfNumbers {

  public int[] arr = new int[10];


  public void writeList() {
```

```
    try {
      arr[10] = 11;
    }

    catch (NumberFormatException e1) {
      System.out.println("NumberFormatException => " + e1.getMessage());
    }

    catch (IndexOutOfBoundsException e2) {
      System.out.println("IndexOutOfBoundsException => " + e2.getMessage());
    }

  }
}
class Main {
  public static void main(String[] args) {
    ListOfNumbers list = new ListOfNumbers();
    list.writeList();
  }
}
```

Output

```
IndexOutOfBoundsException => Index 10 out of bounds for length 10
```

In this example, we have created an integer array named `arr` of size **10**.

Since the array index starts from **0**, the last element of the array is at `arr[9]`. Notice the statement,

```
arr[10] = 11;
```

Here, we are trying to assign a value to the index **10**.
Hence, `IndexOutOfBoundException` occurs.

When an exception occurs in the `try` block,

- The exception is thrown to the first `catch` block. The first `catch` block does not handle an `IndexOutOfBoundsException`, so it is passed to the next `catch` block.
- The second `catch` block in the above example is the appropriate exception handler because it handles an `IndexOutOfBoundsException`. Hence, it is executed.

Catching Multiple Exceptions

From Java SE 7 and later, we can now catch more than one type of exception with one `catch` block.

This reduces code duplication and increases code simplicity and efficiency.

Each exception type that can be handled by the `catch` block is separated using a vertical bar `|`.

Its syntax is:

```
try {
  // code
} catch (ExceptionType1 | Exceptiontype2 ex) {
  // catch block
}
```

To learn more, visit Java catching multiple exceptions.

Java try-with-resources statement

The **try-with-resources** statement is a try statement that has one or more resource declarations.

Its syntax is:

```
try (resource declaration) {
  // use of the resource
```

```
} catch (ExceptionType e1) {

  // catch block

}
```

The resource is an object to be closed at the end of the program. It must be declared and initialized in the try statement.

Let's take an example.

```
try (PrintWriter out = new PrintWriter(new FileWriter("OutputFile.txt")) {

  // use of the resource

}
```

The **try-with-resources** statement is also referred to as **automatic resource management**. This statement automatically closes all the resources at the end of the statement.

Java throw and throws

In this tutorial, we will learn to use throw and throws keyword for exception handling with the help of examples.

In Java, exceptions can be categorized into two types:

- **Unchecked Exceptions:** They are not checked at compile-time but at run-time.For example: `ArithmeticException`, `NullPointerException`, `ArrayIndexOutOfBoundsException`, exceptions under `Error` class, etc.
- **Checked Exceptions:** They are checked at compile-time. For example, `IOException`, `InterruptedException`, etc.

Refer to Java Exceptions to learn in detail about checked and unchecked exceptions.

Usually, we don't need to handle unchecked exceptions. It's because unchecked exceptions occur due to programming errors. And, it is a good practice to correct them instead of handling them.

This tutorial will now focus on how to handle checked exceptions using `throw` and `throws`.

Java throws keyword

We use the `throws` keyword in the method declaration to declare the type of exceptions that might occur within it.

Its syntax is:

```
accessModifier returnType methodName() throws ExceptionType1, ExceptionType2 … {

  // code

}
```

As you can see from the above syntax, we can use `throws` to declare multiple exceptions.

Example 1: Java throws Keyword

```
import java.io.*;class Main {

  public static void findFile() throws IOException {

    // code that may produce IOException

    File newFile=new File("test.txt");

    FileInputStream stream=new FileInputStream(newFile);

  }



  public static void main(String[] args) {

    try{

      findFile();

    } catch(IOException e){

      System.out.println(e);

    }

  }

}
```

Output

```
java.io.FileNotFoundException: test.txt (No such file or directory)
```

When we run this program, if the file `test.txt` does not exist, `FileInputStream` throws a `FileNotFoundException` which extends the `IOException` class.

If a method does not handle exceptions, the type of exceptions that may occur within it must be specified in the `throws` clause so that methods further up in the call stack can handle them or specify them using `throws` keyword themselves.

The `findFile()` method specifies that an `IOException` can be thrown. The `main()` method calls this method and handles the exception if it is thrown.

Throwing multiple exceptions

Here's how we can throw multiple exceptions using the `throws` keyword.

```
import java.io.*;class Main {

  public static void findFile() throws NullPointerException, IOException, InvalidClassException
{



    // code that may produce NullPointerException

    … … …



    // code that may produce IOException

    … … …



    // code that may produce InvalidClassException

    … … …

  }


  public static void main(String[] args) {

    try{

      findFile();

    } catch(IOException e1){
```

```
      System.out.println(e1.getMessage());
    } catch(InvalidClassException e2){
      System.out.println(e2.getMessage());
    }
  }
}
```

Here, the `findFile()` method specifies that it can throw `NullPointerException`, `IOException`, and `InvalidClassException` in its `throws` clause.

Note that we have not handled the `NullPointerException`. This is because it is an unchecked exception. It is not necessary to specify it in the `throws` clause and handle it.

---

throws keyword Vs. try...catch...finally

There might be several methods that can cause exceptions. Writing `try...catch` for each method will be tedious and code becomes long and less-readable.

`throws` is also useful when you have checked exception (an exception that must be handled) that you don't want to catch in your current method.

Java throw keyword

The `throw` keyword is used to explicitly throw a single exception.

When an exception is thrown, the flow of program execution transfers from the `try` block to the `catch` block. We use the `throw` keyword within a method.

Its syntax is:

```
throw throwableObject;
```

A throwable object is an instance of class `Throwable` or subclass of the `Throwable` class.

Example 2: Java throw keyword

```
class Main {

  public static void divideByZero() {

    throw new ArithmeticException("Trying to divide by 0");

  }


  public static void main(String[] args) {

    divideByZero();

  }

}
```

Output

```
Exception in thread "main" java.lang.ArithmeticException: Trying to divide by 0

    at Main.divideByZero(Main.java:3)

    at Main.main(Main.java:7)

exit status 1
```

In this example, we are explicitly throwing an `ArithmeticException.`

Note: `ArithmeticException` is an unchecked exception. It's usually not necessary to handle unchecked exceptions.

Example 3: Throwing checked exception

```
import java.io.*;class Main {

  public static void findFile() throws IOException {

    throw new IOException("File not found");

  }


  public static void main(String[] args) {

    try {
```

```
      findFile();
      System.out.println("Rest of code in try block");
    } catch (IOException e) {
      System.out.println(e.getMessage());
    }
  }
}
```

Output

```
File not found
```

The `findFile()` method throws an `IOException` with the message we passed to its constructor.

Note that since it is a checked exception, we must specify it in the `throws` clause.

The methods that call this `findFile()` method need to either handle this exception or specify it using `throws` keyword themselves.

We have handled this exception in the `main()` method. The flow of program execution transfers from the `try` block to `catch` block when an exception is thrown. So, the rest of the code in the `try` block is skipped and statements in the `catch` block are executed.

Java catch Multiple Exceptions

In this tutorial, we will learn to handle multiple exceptions in Java with the help of examples.

Before Java 7, we had to write multiple exception handling codes for different types of exceptions even if there was code redundancy.

Let's take an example.

Example 1: Multiple catch blocks

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

```
    try {
      int array[] = new int[10];
      array[10] = 30 / 0;
    } catch (ArithmeticException e) {
      System.out.println(e.getMessage());
    } catch (ArrayIndexOutOfBoundsException e) {
      System.out.println(e.getMessage());
    }
  }
}
```

Output

```
/ by zero
```

In this example, two exceptions may occur:

- `ArithmeticException` because we are trying to divide a number by 0.
- `ArrayIndexOutOfBoundsException` because we have declared a new integer array with array bounds 0 to 9 and we are trying to assign a value to index 10.

We are printing out the exception message in both the `catch` blocks i.e. duplicate code.

The associativity of the assignment operator `=` is right to left, so an `ArithmeticException` is thrown first with the message `/ by zero`.

Handle Multiple Exceptions in a catch Block

In Java SE 7 and later, we can now catch more than one type of exception in a single `catch` block.

Each exception type that can be handled by the `catch` block is separated using a vertical bar or pipe `|`.

Its syntax is:

```
try {
```

```
  // code
} catch (ExceptionType1 | Exceptiontype2 ex) {
  // catch block
}
```

Example 2: Multi-catch block

```
class Main {
  public static void main(String[] args) {
    try {
      int array[] = new int[10];
      array[10] = 30 / 0;
    } catch (ArithmeticException | ArrayIndexOutOfBoundsException e) {
      System.out.println(e.getMessage());
    }
  }
}
```

Output

```
/ by zero
```

Catching multiple exceptions in a single `catch` block reduces code duplication and increases efficiency.

The bytecode generated while compiling this program will be smaller than the program having multiple `catch` blocks as there is no code redundancy.

Note: If a `catch` block handles multiple exceptions, the catch parameter is implicitly `final`. This means we cannot assign any values to catch parameters.

Catching base Exception

When catching multiple exceptions in a single `catch` block, the rule is generalized to specialized.

This means that if there is a hierarchy of exceptions in the `catch` block, we can catch the base exception only instead of catching multiple specialized exceptions.

Let's take an example.

Example 3: Catching base exception class only

```
class Main {

  public static void main(String[] args) {

    try {

      int array[] = new int[10];

      array[10] = 30 / 0;

    } catch (Exception e) {

      System.out.println(e.getMessage());

    }

  }

}
```

Output

```
/ by zero
```

We know that all the exception classes are subclasses of the `Exception` class. So, instead of catching multiple specialized exceptions, we can simply catch the `Exception` class.

If the base exception class has already been specified in the `catch` block, do not use child exception classes in the same `catch` block. Otherwise, we will get a compilation error.

Let's take an example.

Example 4: Catching base and child exception classes

```
class Main {
```

```
  public static void main(String[] args) {
    try {
      int array[] = new int[10];
      array[10] = 30 / 0;
    } catch (Exception | ArithmeticException | ArrayIndexOutOfBoundsException e) {
      System.out.println(e.getMessage());
    }
  }
}
```

Output

```
Main.java:6: error: Alternatives in a multi-catch statement cannot be related by subclassing
```

In this example, `ArithmeticException` and `ArrayIndexOutOfBoundsException` are both subclasses of the `Exception` class. So, we get a compilation error.

Java try-with-resources

In this tutorial, we will learn about the try-with-resources statement to close resources automatically.

The `try-with-resources` statement automatically closes all the resources at the end of the statement. A resource is an object to be closed at the end of the program.

Its syntax is:

```
try (resource declaration) {
  // use of the resource
} catch (ExceptionType e1) {
  // catch block
}
```

As seen from the above syntax, we declare the `try-with-resources` statement by,

---

---

1. declaring and instantiating the resource within the `try` clause.
2. specifying and handling all exceptions that might be thrown while closing the resource.

Note: The try-with-resources statement closes all the resources that implement the AutoCloseable interface.

Let us take an example that implements the `try-with-resources` statement.

Example 1: try-with-resources

```
import java.io.*;

class Main {

  public static void main(String[] args) {

    String line;

    try(BufferedReader br = new BufferedReader(new FileReader("test.txt"))) {

      while ((line = br.readLine()) != null) {

        System.out.println("Line =>"+line);

      }

    } catch (IOException e) {

      System.out.println("IOException in try block =>" + e.getMessage());

    }

  }

}
```

Output if the test.txt file is not found.

```
IOException in try-with-resources block =>test.txt (No such file or directory)
```

Output if the test.txt file is found.

```
Entering try-with-resources block

Line =>test line
```

In this example, we use an instance of `BufferedReader` to read data from the `test.txt` file.

Declaring and instantiating the `BufferedReader` inside the `try-with-resources` statement ensures that its instance is closed regardless of whether the `try` statement completes normally or throws an exception.

If an exception occurs, it can be handled using the exception handling blocks or the throws keyword.

Suppressed Exceptions

In the above example, exceptions can be thrown from the `try-with-resources` statement when:

- The file `test.txt` is not found.
- Closing the `BufferedReader` object.

An exception can also be thrown from the `try` block as a file read can fail for many reasons at any time.

If exceptions are thrown from both the `try` block and the `try-with-resources` statement, exception from the `try` block is thrown and exception from the `try-with-resources` statement is suppressed.

Retrieving Suppressed Exceptions

In Java 7 and later, the suppressed exceptions can be retrieved by calling the `Throwable.getSuppressed()` method from the exception thrown by the `try` block.

This method returns an array of all suppressed exceptions. We get the suppressed exceptions in the `catch` block.

```
catch(IOException e) {

  System.out.println("Thrown exception=>" + e.getMessage());

  Throwable[] suppressedExceptions = e.getSuppressed();

  for (int i=0; i<suppressedExceptions.length; i++) {

    System.out.println("Suppressed exception=>" + suppressedExceptions[i]);

  }
```

```
}
```

Advantages of using try-with-resources

Here are the advantages of using try-with-resources:

1. finally block not required to close the resource

Before Java 7 introduced this feature, we had to use the `finally` block to ensure that the resource is closed to avoid resource leaks.

Here's a program that is similar to **Example 1**. However, in this program, we have used finally block to close resources.

Example 2: Close resource using finally block

```
import java.io.*;

class Main {

  public static void main(String[] args) {

    BufferedReader br = null;

    String line;


    try {

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

      br = new BufferedReader(new FileReader("test.txt"));

      while ((line = br.readLine()) != null) {

        System.out.println("Line =>"+line);

      }

    } catch (IOException e) {

      System.out.println("IOException in try block =>" + e.getMessage());

    } finally {

      System.out.println("Entering finally block");

      try {

        if (br != null) {

          br.close();
```

```
        }
      } catch (IOException e) {
        System.out.println("IOException in finally block =>"+e.getMessage());
      }

    }
  }
}
```

Output

```
Entering try block
Line =>line from test.txt file
Entering finally block
```

As we can see from the above example, the use of `finally` block to clean up resources makes the code more complex.

Notice the `try...catch` block in the `finally` block as well? This is because an `IOException` can also occur while closing the `BufferedReader` instance inside this `finally` block so it is also caught and handled.

The `try-with-resources` statement does **automatic resource management**. We need not explicitly close the resources as JVM automatically closes them. This makes the code more readable and easier to write.

---

2. try-with-resources with multiple resources

We can declare more than one resource in the `try-with-resources` statement by separating them with a semicolon `;`

Example 3: try with multiple resources

```
import java.io.*;import java.util.*;class Main {
  public static void main(String[] args) throws IOException{
```

```
    try (Scanner scanner = new Scanner(new File("testRead.txt"));
      PrintWriter writer = new PrintWriter(new File("testWrite.txt"))) {
      while (scanner.hasNext()) {
        writer.print(scanner.nextLine());
      }
    }
  }
}
```

If this program executes without generating any exceptions, `Scanner` object reads a line from the `testRead.txt` file and writes it in a new `testWrite.txt` file.

When multiple declarations are made, the `try-with-resources` statement closes these resources in reverse order. In this example, the `PrintWriter` object is closed first and then the `Scanner` object is closed.

Java 9 try-with-resources enhancement

In Java 7, there is a restriction to the `try-with-resources` statement. The resource needs to be declared locally within its block.

```
try (Scanner scanner = new Scanner(new File("testRead.txt"))) {
  // code
}
```

If we declared the resource outside the block in Java 7, it would have generated an error message.

```
Scanner scanner = new Scanner(new File("testRead.txt"));try (scanner) {
  // code
}
```

To deal with this error, Java 9 improved the `try-with-resources` statement so that the reference of the resource can be used even if it is not declared locally. The above code will now execute without any compilation error.
