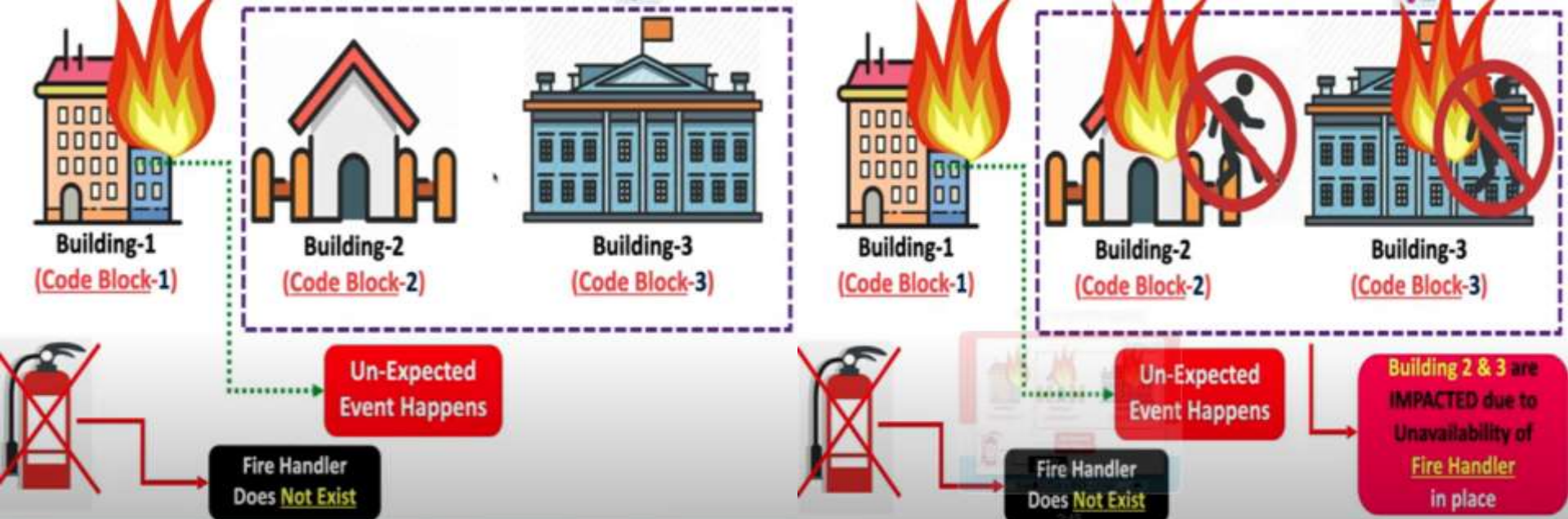


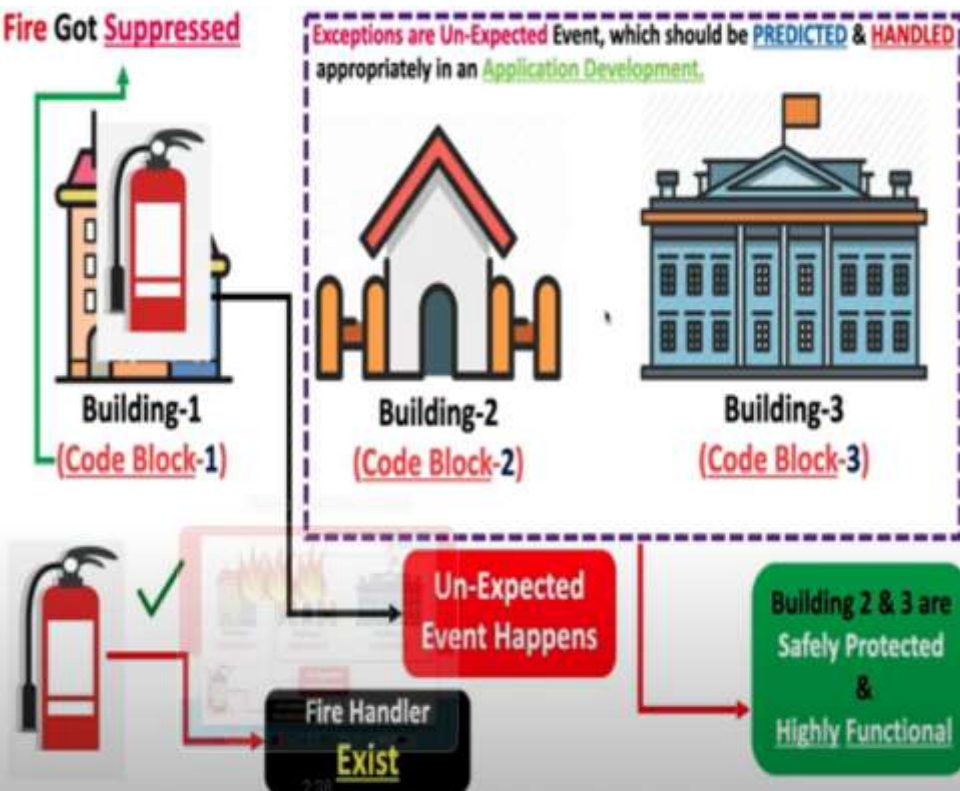
UNIT -II

*Biswajit Senapati,
Faculty,
Department of CSE,
NIT AP,
Tadepalligudem, AP.*

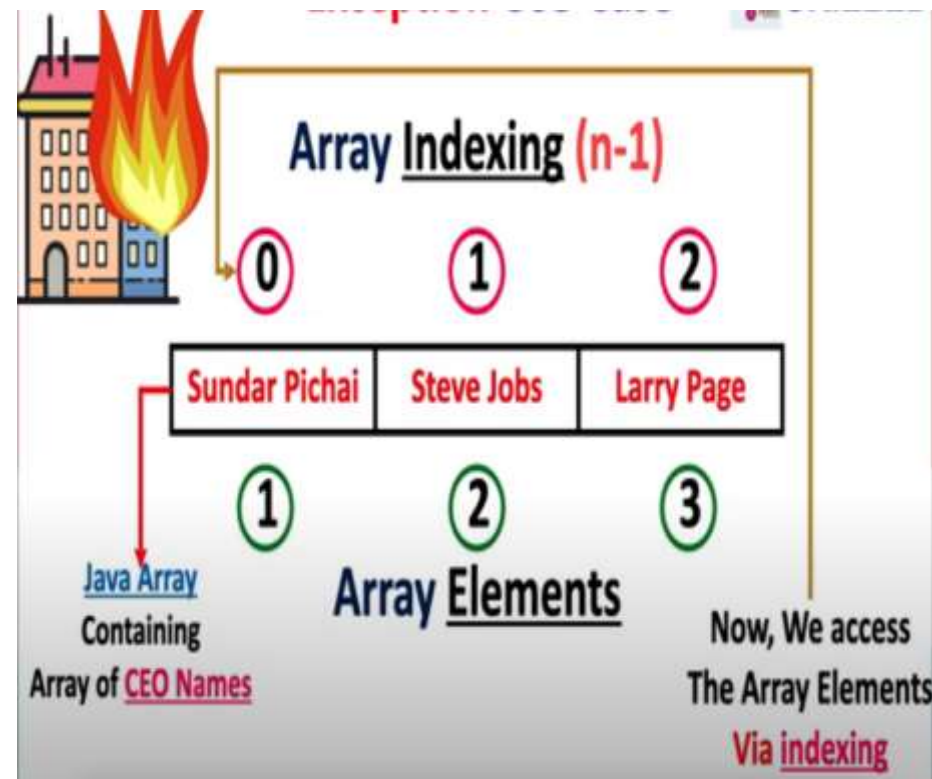
Exception Handling



Fire Got Suppressed



Exceptions are Un-Expected Event, which should be PREDICTED & HANDLED appropriately in an Application Development.



Exception In-Code

```
public class HowExceptionWorks {
```

① Success ✓

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

② Success ✓

```
String[] ceo = new String[] {"Sundar Pichai", "Steve Jobs", "Larry Page"};
```

④ System.out.println(ceo[0]);

Success ✓ O/P SundarPichai

⑤ System.out.println(ceo[2]);

Success ✓ O/P Larry page

⑥ System.out.println(ceo[4]);

Failed X Program Terminates

⑦ System.out.println("Connecting to DB To Store CEO Details");

Cause of Failure

Java.lang.

Line 7

Goes Un-Executed
Due to the Failure of

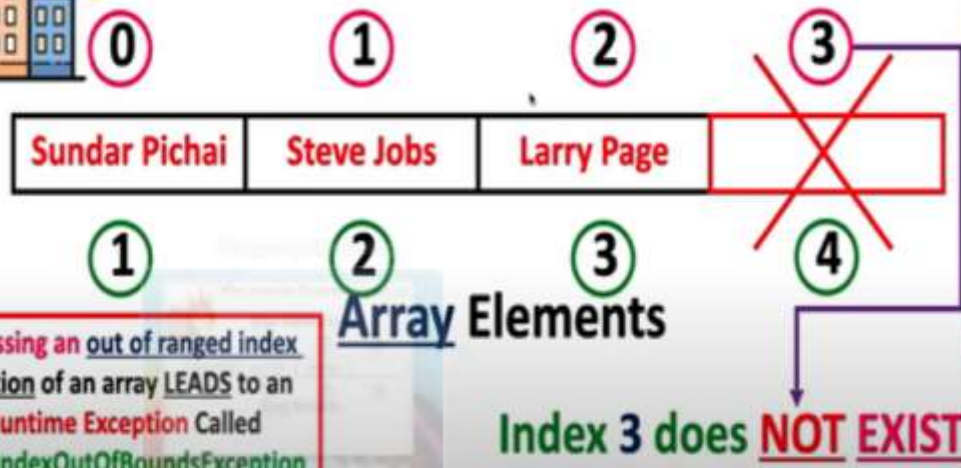
Line 6

③ Success ✓

ArrayIndexOutOfBoundsException



Array Indexing (n-1)



Accessing an out of ranged index Position of an array LEADS to an Runtime Exception Called `ArrayIndexOutOfBoundsException`

Solution To Handle Exception

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

Code that might throw `MyCustom Exception`, `FileNotFoundException` or `SQLException`

Code that Catch all types of checked & Runtime Exceptions.

Exception Basics

When a program violates the Semantic Constraints of the Java programming language, the Java Virtual Machine signals this error to the program as an EXCEPTION.

An example of such a **violation** is an attempt to index outside the bounds of an array.

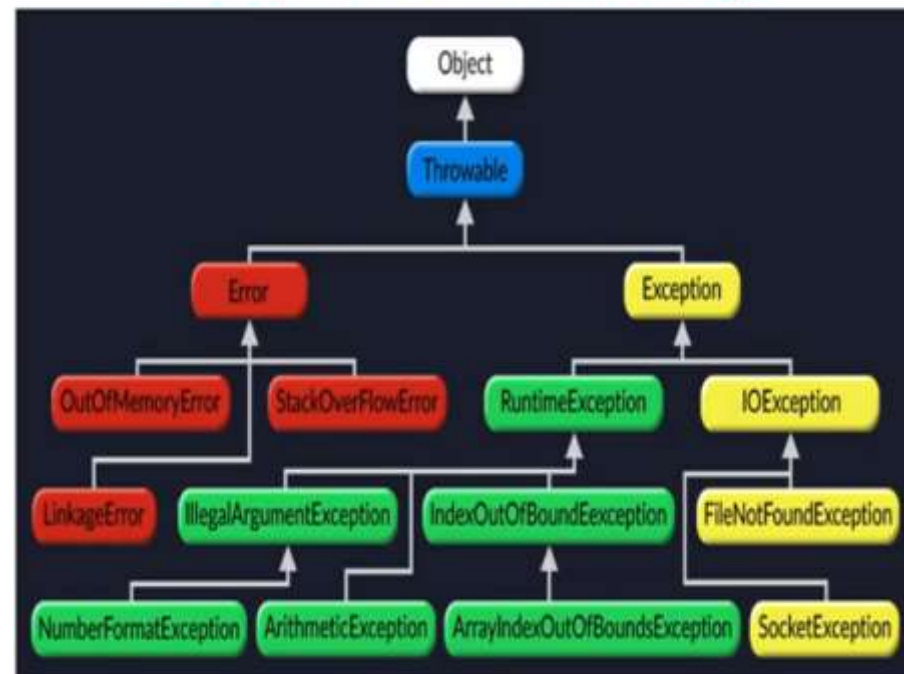
Types of Exceptions:

Checked Exception

&

Unchecked Exception

Exception Class Hierarchy



Exception Handling

An exception in java programming is an abnormal situation that is araised during the program execution. In simple words, an exception is a problem that arises at the time of program execution.

When an exception occurs, it disrupts the program execution flow. When an exception occurs, the program execution gets terminated, and the system generates an error. We use the exception handling mechanism to avoid abnormal termination of program execution.

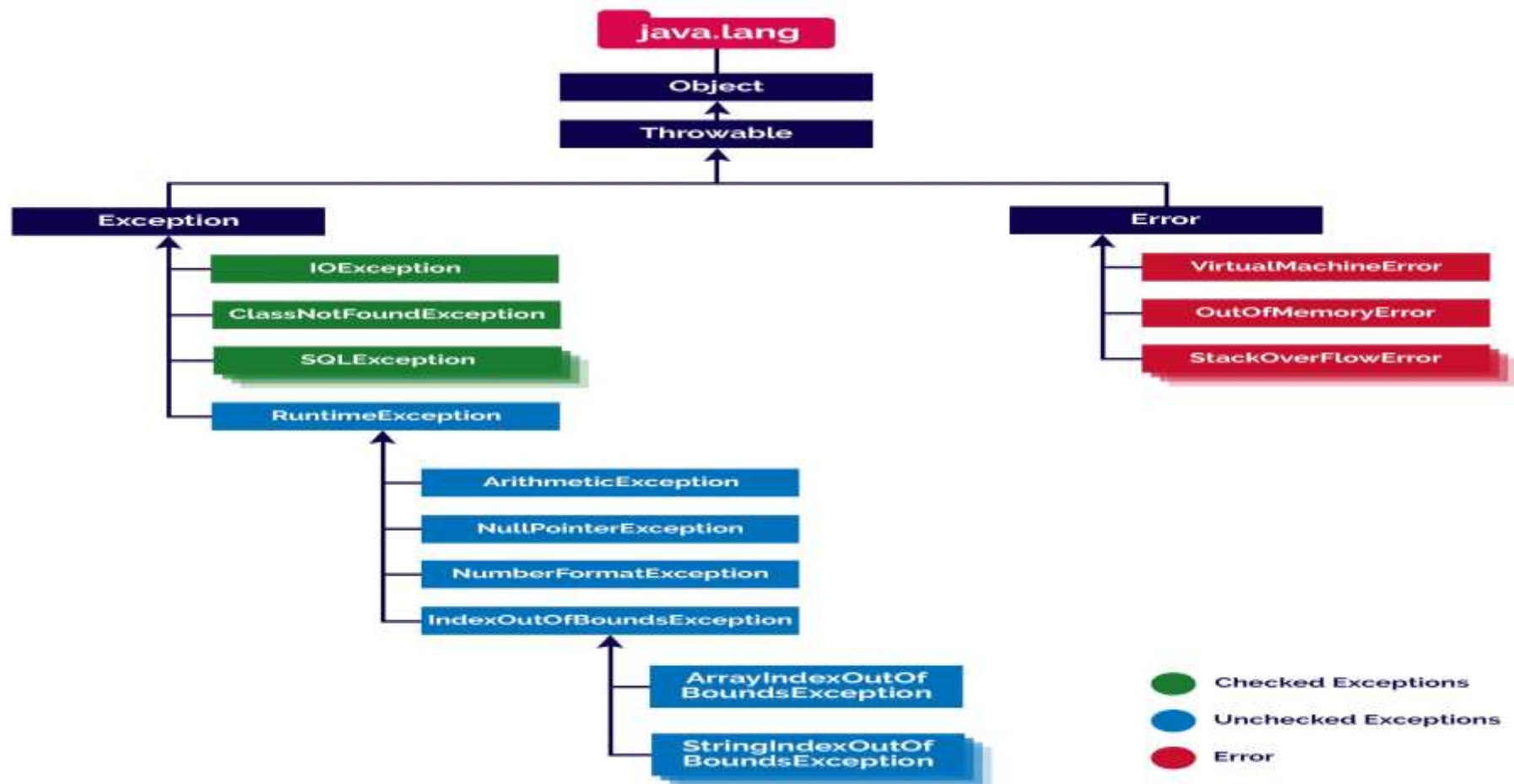
Java programming language has a very powerful and efficient exception handling mechanism with a large number of built-in classes to handle most of the exceptions automatically.

Java programming language has the following class hierarchy to support the exception handling mechanism.

Reasons for Exception Occurrence

Several reasons lead to the occurrence of an exception. A few of them are as follows.

- * When we try to open a file that does not exist may lead to an exception.
- * When the user enters invalid input data, it may lead to an exception.
- * When a network connection has lost during the program execution may lead to an exception.
- * When we try to access the memory beyond the allocated range may lead to an exception.
- * The physical device problems may also lead to an exception.



Types of Exception

In java, exceptions have categorized into two types, and they are as follows.

- * **Checked Exception** - An exception that is checked by the compiler at the time of compilation is called a checked exception.
- * **Unchecked Exception** - An exception that can not be caught by the compiler but occurs at the time of program execution is called an unchecked exception.

How exceptions handled in Java?

In java, the exception handling mechanism uses five keywords namely `try`, `catch`, `finally`, `throw`, and `throws`.

We will learn all these concepts in this series of tutorials.

In java, exceptions are mainly categorized into two types, and they are as follows.

- * Checked Exceptions
- * Unchecked Exceptions

Checked Exceptions

The checked exception is an exception that is checked by the compiler during the compilation process to confirm whether the exception is handled by the programmer or not. If it is not handled, the compiler displays a compilation error using built-in classes.

The checked exceptions are generally caused by faults outside of the code itself like missing resources, networking errors, and problems with threads come to mind.

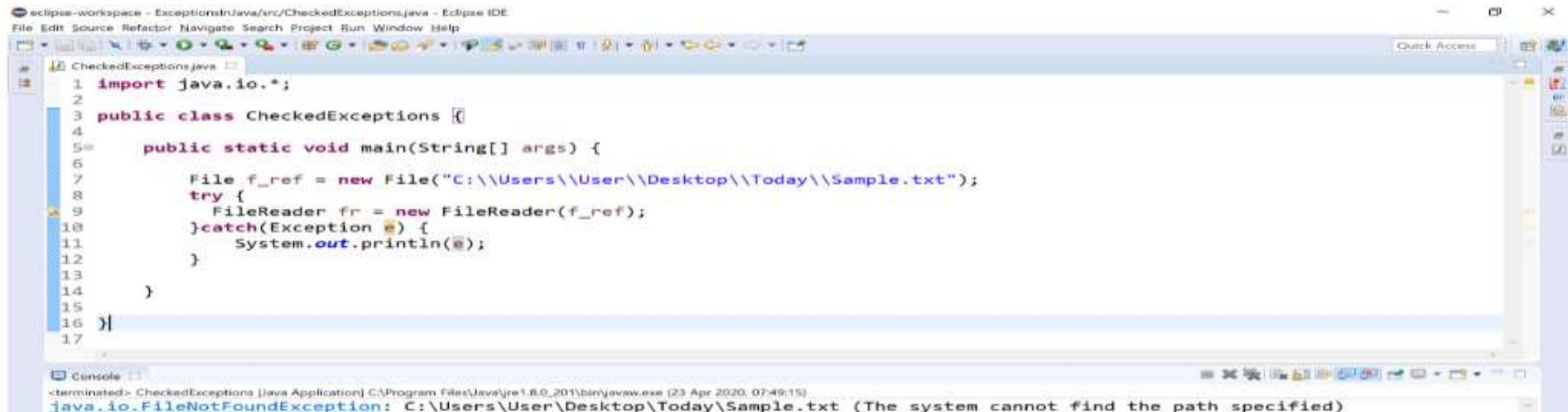
The following are a few built-in classes used to handle checked exceptions in java.

- * IOException
- * FileNotFoundException
- * ClassNotFoundException
- * SQLException
- * DataAccessException
- * InstantiationException
- * UnknownHostException

➡ In the exception class hierarchy, the checked exception classes are the direct children of the Exception class.

The checked exception is also known as a compile-time exception.

When we run the above program, it produces the following output.



The screenshot shows the Eclipse IDE with a Java file named `CheckedExceptions.java`. The code is as follows:

```
1 import java.io.*;
2
3 public class CheckedExceptions {
4
5     public static void main(String[] args) {
6
7         File f_ref = new File("C:\\Users\\User\\Desktop\\Today\\Sample.txt");
8         try {
9             FileReader fr = new FileReader(f_ref);
10        } catch (Exception e) {
11            System.out.println(e);
12        }
13    }
14 }
15
16 }
17 }
```

The console output at the bottom shows the program terminated with the following message:

```
<terminated> CheckedExceptions [Java Application] C:\Program Files\Java\jre1.8.0_201\bin\javaw.exe (23 Apr 2020, 07:49:15)
java.io.FileNotFoundException: C:\Users\User\Desktop\Today\Sample.txt (The system cannot find the path specified)
```


Unchecked Exceptions

The unchecked exception is an exception that occurs at the time of program execution. The unchecked exceptions are not caught by the compiler at the time of compilation.

The unchecked exceptions are generally caused due to bugs such as logic errors, improper use of resources, etc.

The following are a few built-in classes used to handle unchecked exceptions in java.

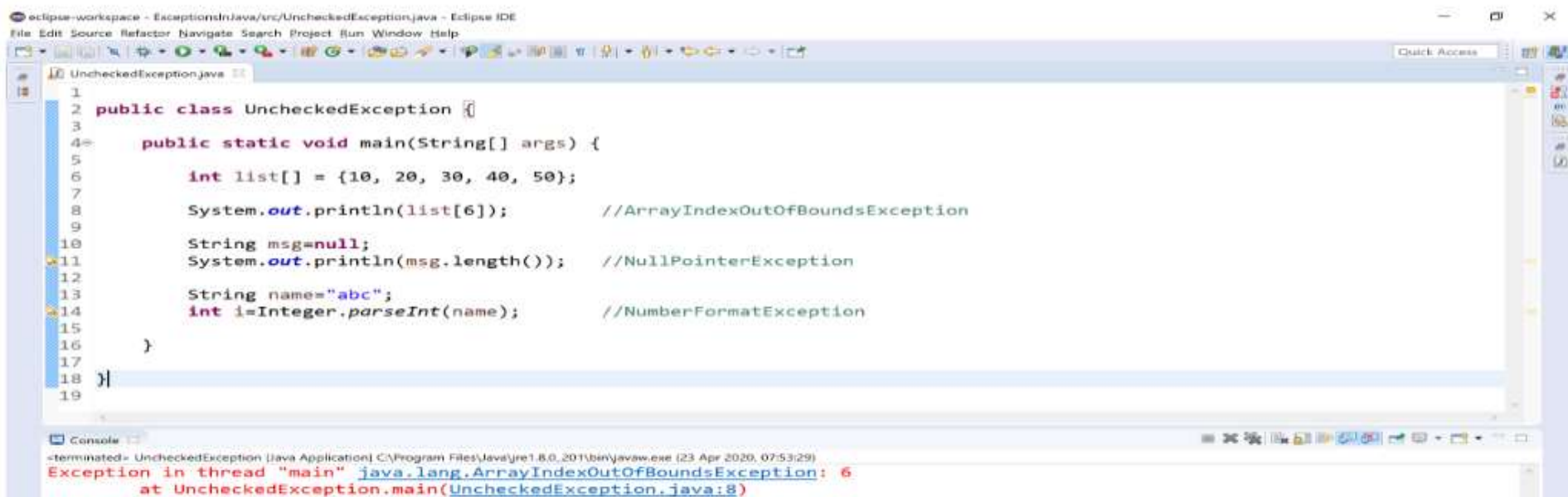
- * ArithmeticException
- * NullPointerException
- * NumberFormatException
- * ArrayIndexOutOfBoundsException
- * StringIndexOutOfBoundsException

🚩 In the exception class hierarchy, the unchecked exception classes are the children of RuntimeException class, which is a child class of Exception class.

The unchecked exception is also known as a runtime exception.

Let's look at the following example program for the unchecked exceptions.

When we run the above program, it produces the following output.



The screenshot shows the Eclipse IDE with a Java file named `UncheckedException.java`. The code defines a public class `UncheckedException` with a `main` method. The `main` method contains four lines of code that will throw unchecked exceptions: an array index out of bounds exception, a null pointer exception, and a number format exception. The console output shows the program terminating with an `ArrayIndexOutOfBoundsException` at line 8.

```
1 public class UncheckedException {
2
3
4     public static void main(String[] args) {
5
6         int list[] = {10, 20, 30, 40, 50};
7
8         System.out.println(list[6]);           //ArrayIndexOutOfBoundsException
9
10        String msg=null;
11        System.out.println(msg.length());      //NullPointerException
12
13        String name="abc";
14        int i=Integer.parseInt(name);          //NumberFormatException
15    }
16 }
17
18 }
19
```

Console Output:

```
<terminated> UncheckedException [Java Application] C:\Program Files\Java\jre1.8.0_201\bin\javaw.exe (23 Apr 2020, 07:53:29)
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 6
    at UncheckedException.main(UncheckedException.java:8)
```

Exception Models

In java, there are two exception models. Java programming language has two models of exception handling. The exception models that java supports are as follows.

- * Termination Model
- * Resumptive Model

Let's look into details of each exception model.

Termination Model

In the termination model, when a method encounters an exception, further processing in that method is terminated and control is transferred to the nearest catch block that can handle the type of exception encountered.

In other words we can say that in termination model the error is so critical there is no way to get back to where the exception occurred.

Resumptive Model

The alternative of termination model is resumptive model. In resumptive model, the exception handler is expected to do something to stabilize the situation, and then the faulting method is retried. In resumptive model we hope to continue the execution after the exception is handled.

In resumptive model we may use a method call that wants resumption like behavior. We may also place the try block in a while loop that keeps re-entering the try block until the result is satisfactory.

Uncaught Exceptions

In java, assume that, if we do not handle the exceptions in a program. In this case, when an exception occurs in a particular function, then Java prints a exception message with the help of uncaught exception handler.

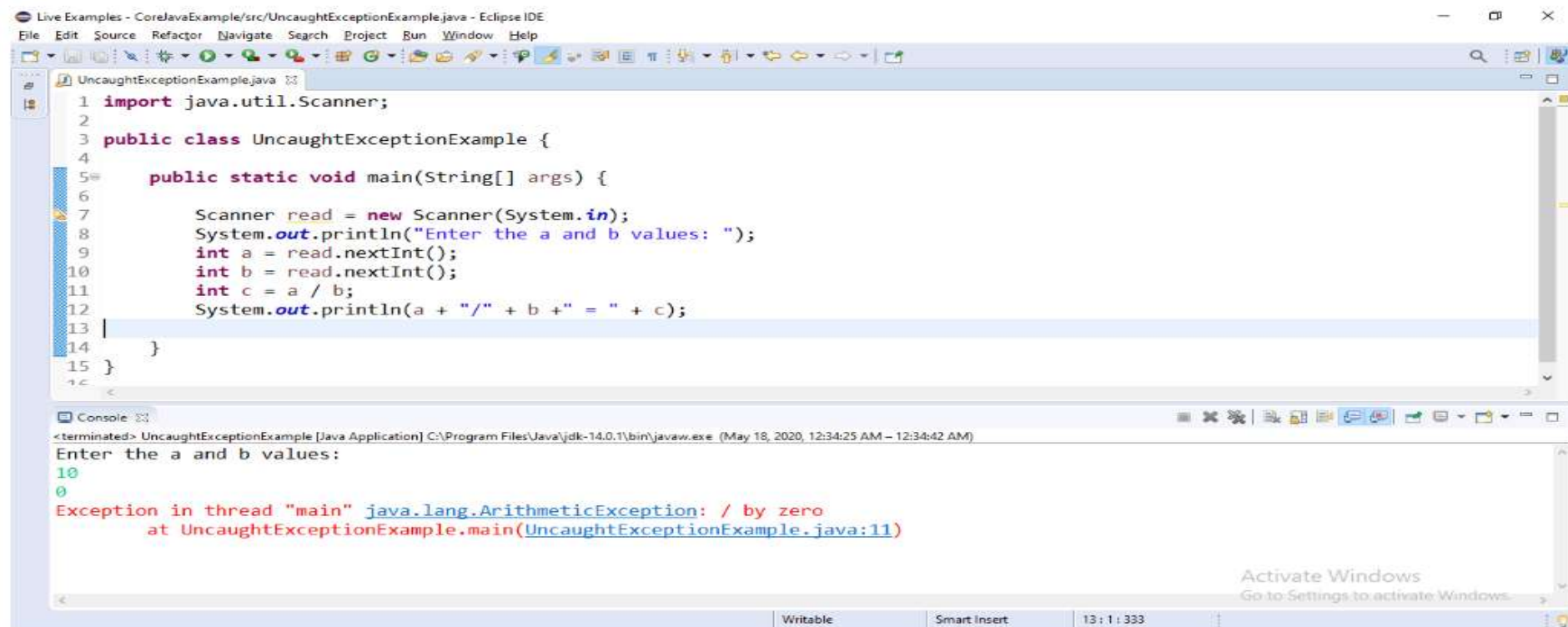
The uncaught exceptions are the exceptions that are not caught by the compiler but automatically caught and handled by the Java built-in exception handler.

Java programming language has a very strong exception handling mechanism. It allow us to handle the exception use the keywords like try, catch, finally, throw, and throws.

When an uncaught exception occurs, the JVM calls a special private method known `dispatchUncaughtException()` on the Thread class in which the exception occurs and terminates the thread.

The Division by zero exception is one of the example for uncaught exceptions. Look at the following code.

When we execute the above code, it produce the following output for the value a = 10 and b = 0.



The screenshot shows the Eclipse IDE with a Java file named `UncaughtExceptionExample.java`. The code is as follows:

```
1 import java.util.Scanner;
2
3 public class UncaughtExceptionExample {
4
5     public static void main(String[] args) {
6
7         Scanner read = new Scanner(System.in);
8         System.out.println("Enter the a and b values: ");
9         int a = read.nextInt();
10        int b = read.nextInt();
11        int c = a / b;
12        System.out.println(a + "/" + b + " = " + c);
13    }
14 }
15 }
```

The console output shows the program execution:

```
<terminated> UncaughtExceptionExample [Java Application] C:\Program Files\Java\jdk-14.0.1\bin\javaw.exe (May 18, 2020, 12:34:25 AM - 12:34:42 AM)
Enter the a and b values:
10
0
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at UncaughtExceptionExample.main(UncaughtExceptionExample.java:11)
```

An "Activate Windows" watermark is visible in the bottom right corner of the console window.

In the above example code, we are not used try and catch blocks, but when the value of b is zero the division by zero exception occurs and it caught by the default exception handler.

try and catch

In java, the `try` and `catch` are the keywords used for exception handling.

The keyword `try` is used to define a block of code that will be tested for the occurrence of an exception. The keyword `catch` is used to define a block of code that handles the exception occurred in the respective `try` block.

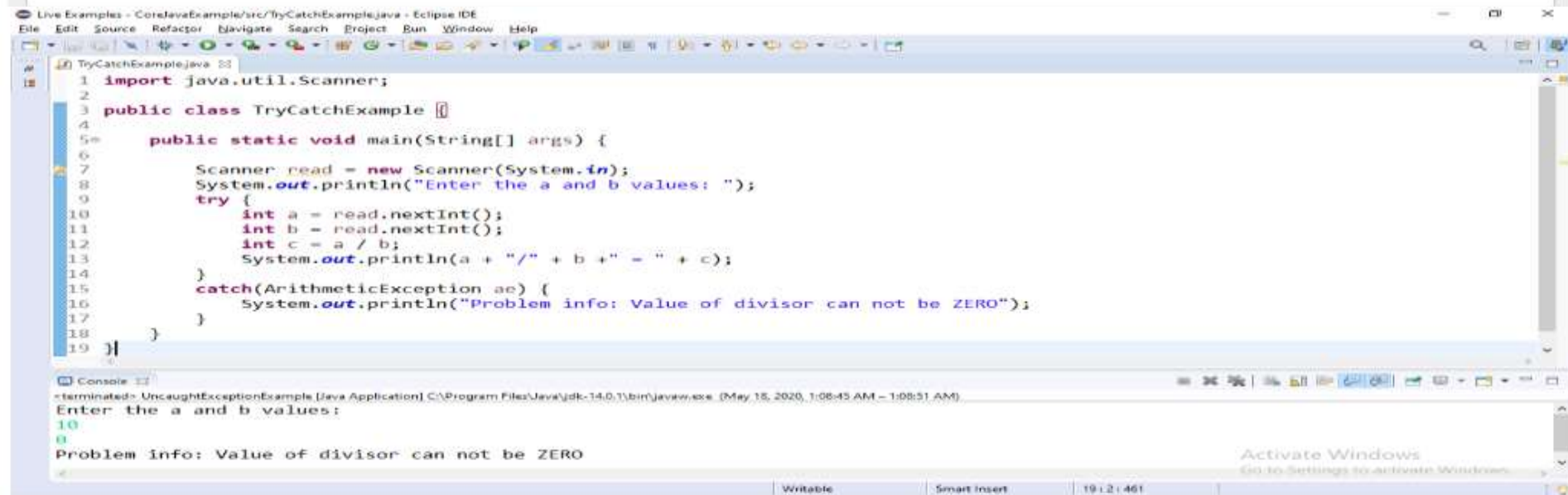
The uncaught exceptions are the exceptions that are not caught by the compiler but automatically caught and handled by the Java built-in exception handler.

Both `try` and `catch` are used as a pair. Every `try` block must have one or more `catch` blocks. We cannot use `try` without at least one `catch`, and `catch` alone cannot be used (`catch` without `try` is not allowed).

The following is the syntax of `try` and `catch` blocks.

Syntax

```
try{
    ...
    code to be tested
    ...
}
catch(ExceptionType object){
    ...
    code for handling the exception
    ...
}
```



The screenshot shows the Eclipse IDE with a Java file named `TryCatchExample.java`. The code is as follows:

```
1 import java.util.Scanner;
2
3 public class TryCatchExample {
4
5     public static void main(String[] args) {
6
7         Scanner read = new Scanner(System.in);
8         System.out.println("Enter the a and b values: ");
9         try {
10             int a = read.nextInt();
11             int b = read.nextInt();
12             int c = a / b;
13             System.out.println(a + "/" + b + " = " + c);
14         }
15         catch(ArithmeticException ae) {
16             System.out.println("Problem info: Value of divisor can not be ZERO");
17         }
18     }
19 }
```

The console output shows the program execution:

```
>terminated> UncaughtExceptionExample [Java Application] C:\Program Files\Java\jdk-14.0.1\bin\javaw.exe (May 15, 2020, 1:08:45 AM - 1:08:51 AM)
Enter the a and b values:
10
0
Problem info: Value of divisor can not be ZERO
```

An "Activate Windows" watermark is visible in the bottom right corner of the IDE window.

In the above example code, when an exception occurs in the `try` block the execution control is transferred to the `catch` block and the `catch` block handles it.

Multiple catch clauses

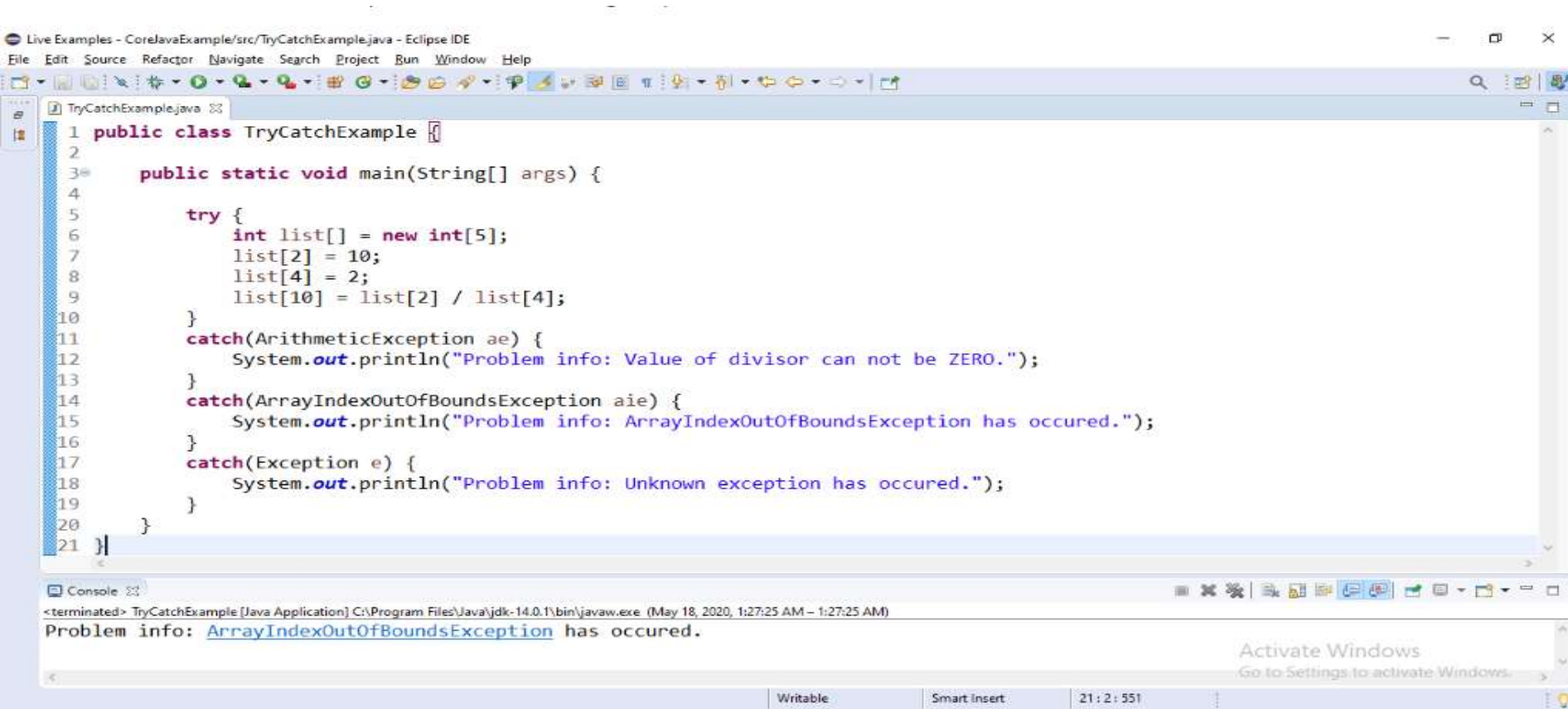
In java programming language, a try block may has one or more number of catch blocks. That means a single try statement can have multiple catch clauses.

When a try block has more than one catch block, each catch block must contain a different exception type to be handled.

The multiple catch clauses are defined when the try block contains the code that may lead to different type of exceptions.

- ▶ The try block generates only one exception at a time, and at a time only one catch block is executed.
- ▶ When there are multiple catch blocks, the order of catch blocks must be from the most specific exception handler to most general.
- ▶ The catch block with Exception class handler must be defined at the last.

Let's look at the following example Java code to illustrate multiple catch clauses.



The screenshot shows the Eclipse IDE with a Java file named `TryCatchExample.java`. The code defines a `public class TryCatchExample` with a `main` method. Inside the `try` block, an array `list` of type `int` is created with size 5. Elements are assigned at indices 2 and 4. An attempt is made to access index 10, which triggers an `ArrayIndexOutOfBoundsException`. The `catch` blocks handle `ArithmeticException`, `ArrayIndexOutOfBoundsException`, and a general `Exception` in that order. The console output shows the message for `ArrayIndexOutOfBoundsException`.

```
1 public class TryCatchExample {
2
3     public static void main(String[] args) {
4
5         try {
6             int list[] = new int[5];
7             list[2] = 10;
8             list[4] = 2;
9             list[10] = list[2] / list[4];
10        }
11        catch(ArithmeticException ae) {
12            System.out.println("Problem info: Value of divisor can not be ZERO.");
13        }
14        catch(ArrayIndexOutOfBoundsException aie) {
15            System.out.println("Problem info: ArrayIndexOutOfBoundsException has occurred.");
16        }
17        catch(Exception e) {
18            System.out.println("Problem info: Unknown exception has occurred.");
19        }
20    }
21 }
```

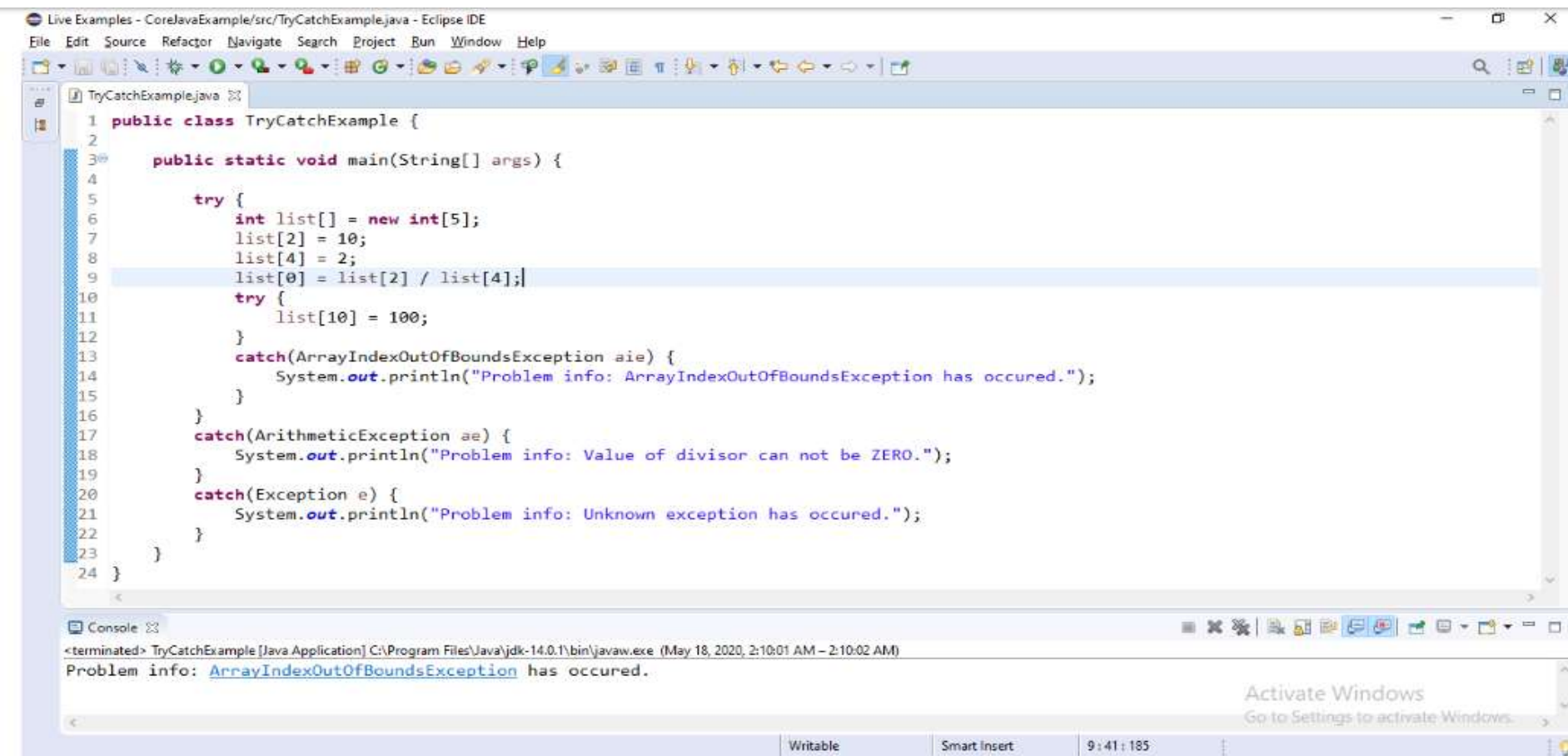
Console Output:
<terminated> TryCatchExample [Java Application] C:\Program Files\Java\jdk-14.0.1\bin\javaw.exe (May 18, 2020, 1:27:25 AM – 1:27:25 AM)
Problem info: [ArrayIndexOutOfBoundsException](#) has occurred.

Nested try statements

The java allows to write a try statement inside another try statement. A try block within another try block is known as nested try block.

When there are nested try blocks, each try block must have one or more separate catch blocks.

Let's look at the following example Java code to illustrate nested try statements.



The screenshot shows the Eclipse IDE with a Java file named `TryCatchExample.java`. The code defines a `public class TryCatchExample` with a `main` method. Inside the `main` method, there is a `try` block. Within this `try` block, there is an `int` array `list` of size 5, and several assignments: `list[2] = 10;`, `list[4] = 2;`, and `list[0] = list[2] / list[4];`. The last line is highlighted. Inside this `try` block, there is another `try` block with `list[10] = 100;`. This inner `try` block has a `catch` block for `ArrayIndexOutOfBoundsException` that prints "Problem info: ArrayIndexOutOfBoundsException has occurred.". The outer `try` block has two more `catch` blocks: one for `ArithmeticException` that prints "Problem info: Value of divisor can not be ZERO.", and one for `Exception` that prints "Problem info: Unknown exception has occurred.". The console output shows the message "Problem info: ArrayIndexOutOfBoundsException has occurred.".

```
1 public class TryCatchExample {
2
3     public static void main(String[] args) {
4
5         try {
6             int list[] = new int[5];
7             list[2] = 10;
8             list[4] = 2;
9             list[0] = list[2] / list[4];
10            try {
11                list[10] = 100;
12            }
13            catch(ArrayIndexOutOfBoundsException aie) {
14                System.out.println("Problem info: ArrayIndexOutOfBoundsException has occurred.");
15            }
16        }
17        catch(ArithmeticException ae) {
18            System.out.println("Problem info: Value of divisor can not be ZERO.");
19        }
20        catch(Exception e) {
21            System.out.println("Problem info: Unknown exception has occurred.");
22        }
23    }
24 }
```

Console Output:

```
<terminated> TryCatchExample [Java Application] C:\Program Files\Java\jdk-14.0.1\bin\javaw.exe (May 18, 2020, 2:10:01 AM - 2:10:02 AM)
Problem info: ArrayIndexOutOfBoundsException has occurred.
```

🔥 In case of nested try blocks, if an exception occurred in the inner try block and it's catch blocks are unable to handle it then it transfers the control to the outer try's catch block to handle it.

throw, throws, and finally keywords

In java, the keywords throw, throws, and finally are used in the exception handling concept. Let's look at each of these keywords.

throw keyword in Java

The throw keyword is used to throw an exception instance explicitly from a try block to corresponding catch block. That means it is used to transfer the control from try block to corresponding catch block.

The throw keyword must be used inside the try block. When JVM encounters the throw keyword, it stops the execution of try block and jump to the corresponding catch block.

- ⚠ Using throw keyword only object of Throwable class or its sub classes can be thrown.
- ⚠ Using throw keyword only one exception can be thrown.
- ⚠ The throw keyword must followed by an throwable instance.

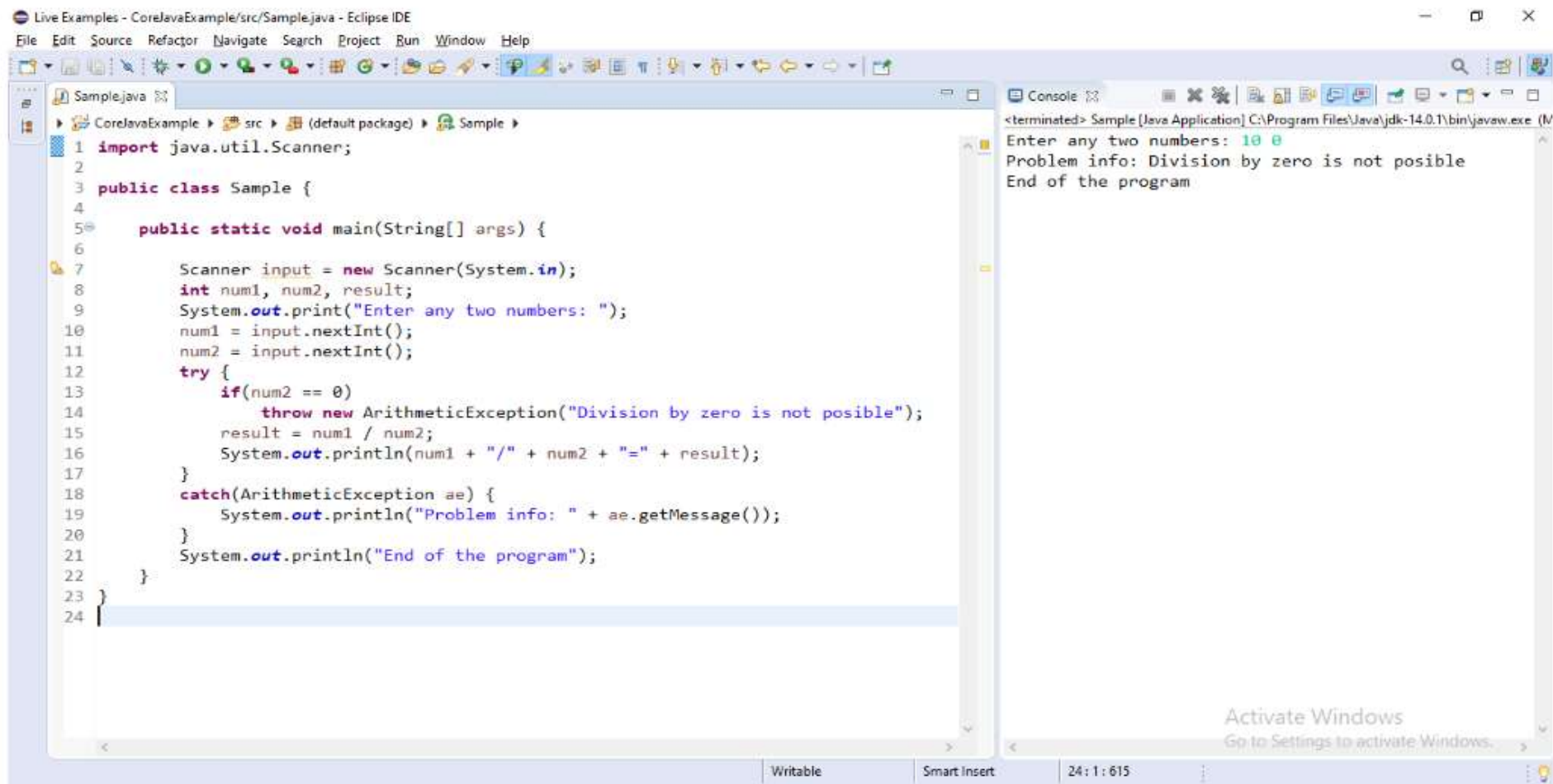
The following is the general syntax for using throw keyword in a try block.

Syntax

```
throw instance;
```

Here the instace must be throwable instance and it can be created dynamically using new operator.

When we run the above code, it produce the following output.



The screenshot shows the Eclipse IDE with a Java file named `Sample.java` open. The code is as follows:

```
1 import java.util.Scanner;
2
3 public class Sample {
4
5     public static void main(String[] args) {
6
7         Scanner input = new Scanner(System.in);
8         int num1, num2, result;
9         System.out.print("Enter any two numbers: ");
10        num1 = input.nextInt();
11        num2 = input.nextInt();
12        try {
13            if(num2 == 0)
14                throw new ArithmeticException("Division by zero is not posible");
15            result = num1 / num2;
16            System.out.println(num1 + "/" + num2 + "=" + result);
17        }
18        catch(ArithmeticException ae) {
19            System.out.println("Problem info: " + ae.getMessage());
20        }
21        System.out.println("End of the program");
22    }
23 }
24 }
```

The console output on the right shows the execution results:

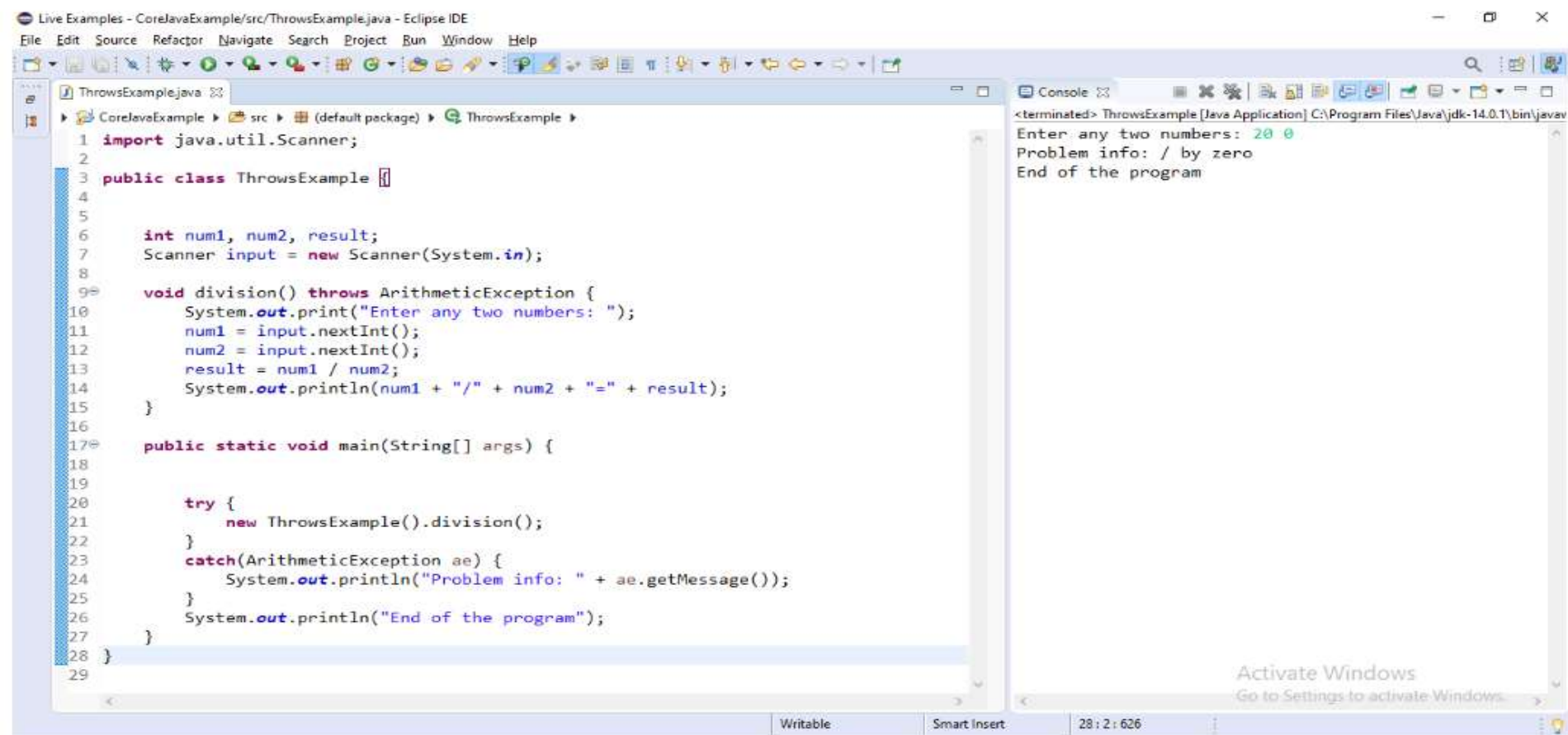
```
<terminated> Sample [Java Application] C:\Program Files\Java\jdk-14.0.1\bin\javaw.exe (N
Enter any two numbers: 10 0
Problem info: Division by zero is not posible
End of the program
```

throws keyword in Java

The throws keyword specifies the exceptions that a method can throw to the default handler and does not handle itself. That means when we need a method to throw an exception automatically, we use throws keyword followed by method declaration

⚠ When a method throws an exception, we must put the calling statement of method in try-catch block.

When we run the above code, it produce the following output.



```
Live Examples - CoreJavaExample/src/ThrowsExample.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help

ThrowsExample.java
CoreJavaExample > src > (default package) > ThrowsExample
1 import java.util.Scanner;
2
3 public class ThrowsExample {
4
5
6     int num1, num2, result;
7     Scanner input = new Scanner(System.in);
8
9     void division() throws ArithmeticException {
10         System.out.print("Enter any two numbers: ");
11         num1 = input.nextInt();
12         num2 = input.nextInt();
13         result = num1 / num2;
14         System.out.println(num1 + "/" + num2 + "=" + result);
15     }
16
17     public static void main(String[] args) {
18
19         try {
20             new ThrowsExample().division();
21         }
22         catch(ArithmeticException ae) {
23             System.out.println("Problem info: " + ae.getMessage());
24         }
25         System.out.println("End of the program");
26     }
27 }
28
29

Console
<terminated> ThrowsExample [Java Application] C:\Program Files\Java\jdk-14.0.1\bin\javaw
Enter any two numbers: 20 0
Problem info: / by zero
End of the program

Activate Windows
Go to Settings to activate Windows.
```

finally keyword in Java

The finally keyword used to define a block that must be executed irrespective of exception occurrence.

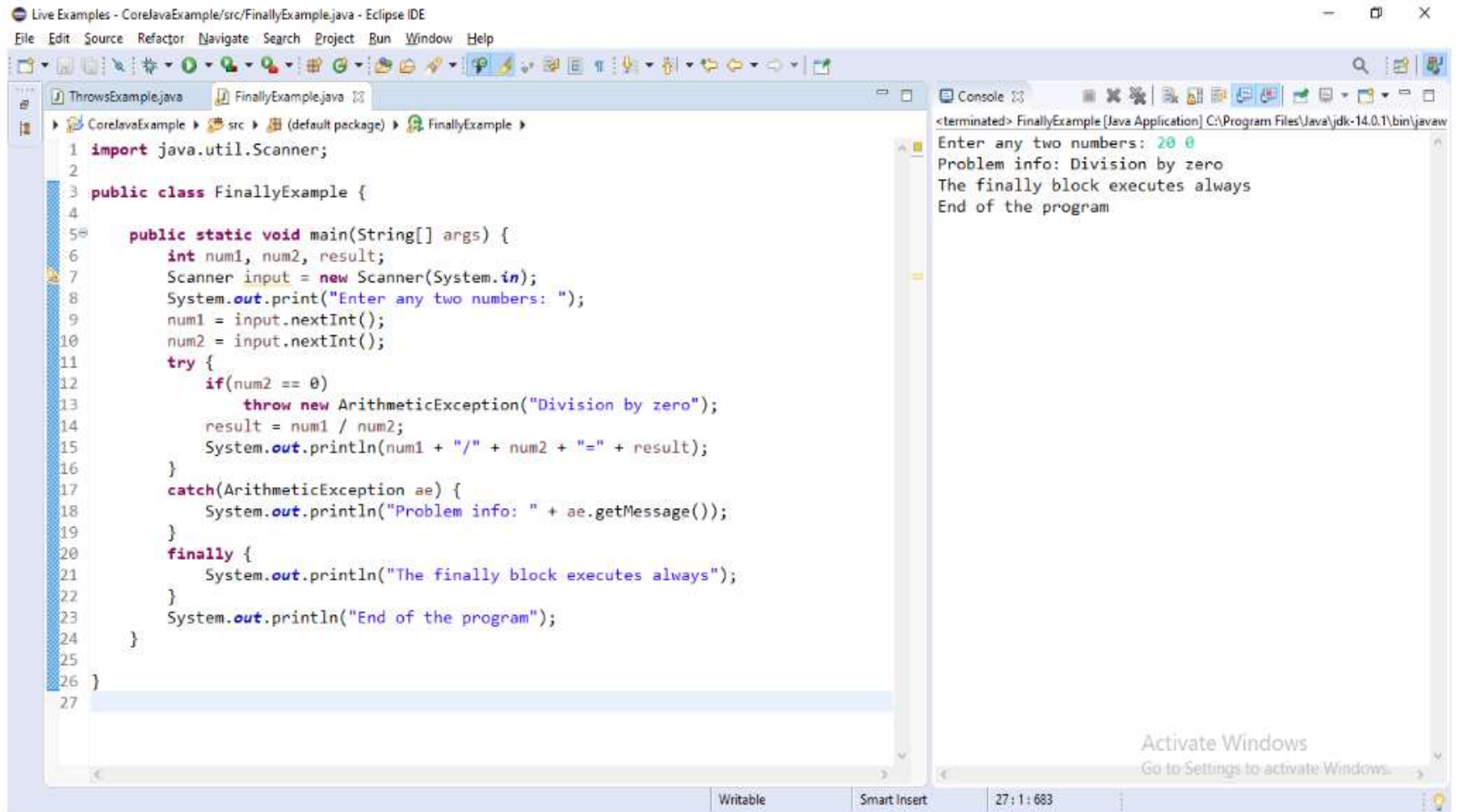
The basic purpose of finally keyword is to cleanup resources allocated by try block, such as closing file, closing database connection, etc.

▲ Only one finally block is allowed for each try block.

▲ Use of finally block is optional.

Let's look at the following example Java code to illustrate throws keyword.

When we run the above code, it produce the following output.



The screenshot shows the Eclipse IDE interface. The main editor window displays the file `FinallyExample.java` with the following code:

```
1 import java.util.Scanner;
2
3 public class FinallyExample {
4
5     public static void main(String[] args) {
6         int num1, num2, result;
7         Scanner input = new Scanner(System.in);
8         System.out.print("Enter any two numbers: ");
9         num1 = input.nextInt();
10        num2 = input.nextInt();
11        try {
12            if(num2 == 0)
13                throw new ArithmeticException("Division by zero");
14            result = num1 / num2;
15            System.out.println(num1 + "/" + num2 + "=" + result);
16        }
17        catch(ArithmeticException ae) {
18            System.out.println("Problem info: " + ae.getMessage());
19        }
20        finally {
21            System.out.println("The finally block executes always");
22        }
23        System.out.println("End of the program");
24    }
25 }
26
27
```

The console window on the right shows the output of the program:

```
<terminated> FinallyExample [Java Application] C:\Program Files\Java\jdk-14.0.1\bin\javaw
Enter any two numbers: 20 0
Problem info: Division by zero
The finally block executes always
End of the program
```

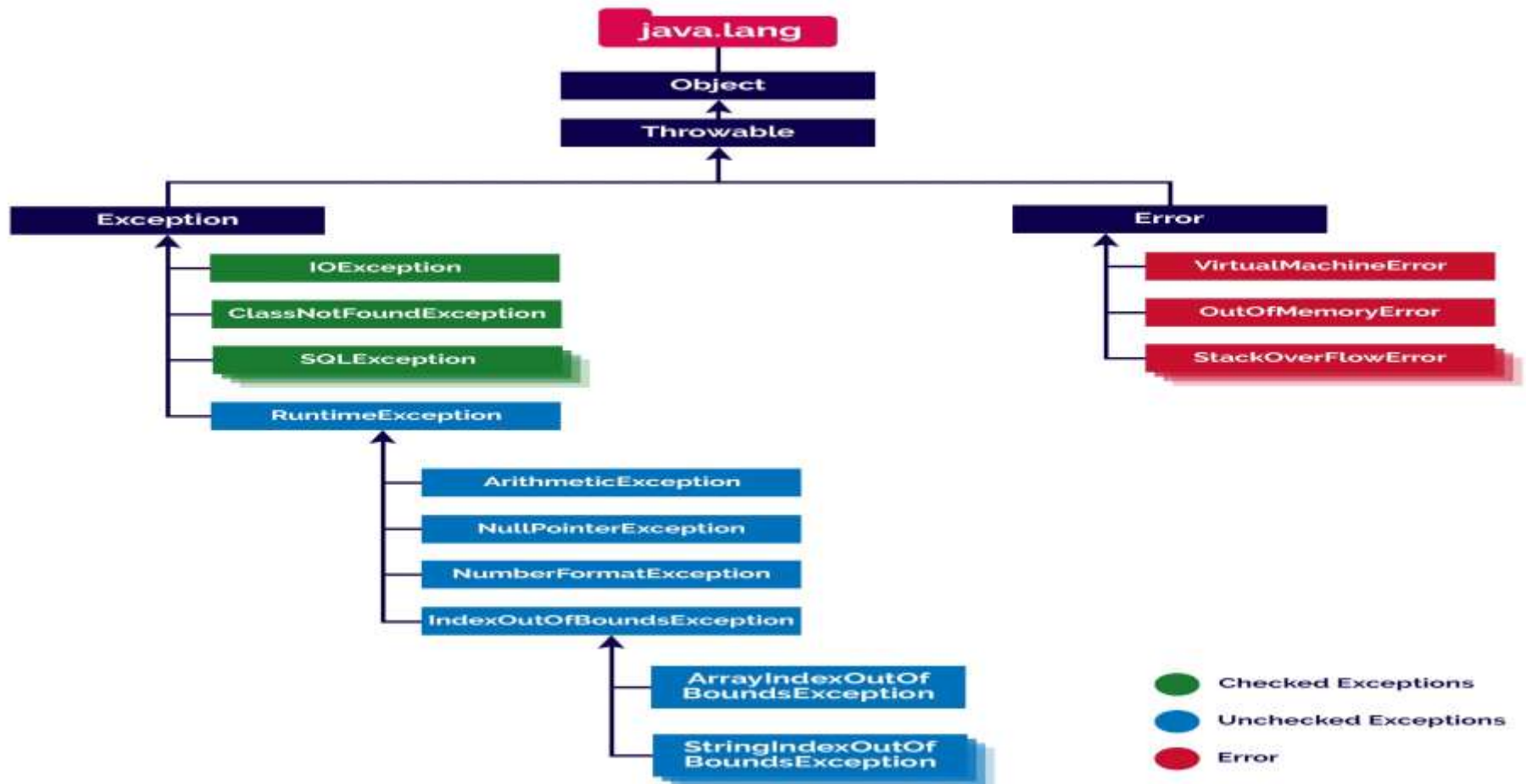
At the bottom of the IDE, there is a status bar with the text "Activate Windows Go to Settings to activate Windows." and a small lightbulb icon.

Built-in Exceptions

The Java programming language has several built-in exception class that support exception handling. Every exception class is suitable to explain certain error situations at run time.

All the built-in exception classes in Java were defined a package `java.lang`.

Few built-in exceptions in Java are shown in the following image.



List of checked exceptions in Java

S. No.	Exception Class with Description
1	ClassNotFoundException It is thrown when the Java Virtual Machine (JVM) tries to load a particular class and the specified class cannot be found in the classpath.
2	CloneNotSupportedException Used to indicate that the clone method in class Object has been called to clone an object, but that the object's class does not implement the Cloneable interface.
3	IllegalAccessException It is thrown when one attempts to access a method or member that visibility qualifiers do not allow.
4	InstantiationException It is thrown when an application tries to create an instance of a class using the newInstance method in class Class , but the specified class object cannot be instantiated because it is an interface or is an abstract class.
5	InterruptedException It is thrown when a thread that is sleeping, waiting, or is occupied is interrupted.
6	NoSuchFieldException It indicates that the class doesn't have a field of a specified name.
7	NoSuchMethodException It is thrown when some JAR file has a different version at runtime that it had at compile time, a NoSuchMethodException occurs during reflection when we try to access a method that does not exist.

List of unchecked exceptions in Java

S. No.	Exception Class with Description
1	ArithmeticException It handles the arithmetic exceptions like division by zero
2	ArrayIndexOutOfBoundsException It handles the situations like an array has been accessed with an illegal index. The index is either negative or greater than or equal to the size of the array.
3	ArrayStoreException It handles the situations like when an attempt has been made to store the wrong type of object into an array of objects
4	AssertionError It is used to indicate that an assertion has failed
5	ClassCastException It handles the situation when we try to improperly cast a class from one type to another.
6	IllegalArgumentException This exception is thrown in order to indicate that a method has been passed an illegal or inappropriate argument.
7	IllegalMonitorStateException This indicates that the calling thread has attempted to wait on an object's monitor, or has attempted to notify other threads that wait on an object's monitor, without owning the specified monitor.
8	IllegalStateException It signals that a method has been invoked at an illegal or inappropriate time.
9	IllegalThreadStateException It is thrown by the Java runtime environment, when the programmer is trying to modify the state of the thread when it is illegal.
10	IndexOutOfBoundsException It is thrown when attempting to access an invalid index within a collection, such as an array , vector , string , and so forth.
11	NegativeArraySizeException It is thrown if an applet tries to create an array with negative size.
12	NullPointerException it is thrown when program attempts to use an object reference that has the null value.
13	NumberFormatException It is thrown when we try to convert a string into a numeric value such as float or integer, but the format of the input string is not appropriate or illegal.
14	SecurityException It is thrown by the Java Card Virtual Machine to indicate a security violation.

Exception Handling with Method Overriding

If the superclass method does not declare an exception:

If the superclass method does not declare an exception, subclass overridden method **cannot** declare the **checked exception** but it **can** declare **unchecked exception**.

```
import java.io.IOException;

class Animal
{
    void eat()
    {
        System.out.println("Animal is eating.");
    }
}

class Dog extends Animal
{
    void eat() throws IOException
    {
        System.out.println("Dog is eating meat.");
    }
}
```



```
class Animal
{
    void eat()
    {
        System.out.println("Animal is eating.");
    }
}

class Dog extends Animal
{
    void eat() throws ArithmeticException
    {
        System.out.println("Dog is eating meat.");
    }
}
```




Exception Handling with Method Overriding

If the superclass method declares an exception:

If the superclass method declares an exception, subclass overridden method **can** declare same, subclass exception or no exception but **cannot** declare parent exception.


```
class Animal
{
    void eat() throws Exception
    {
        System.out.println("Animal is eating.");
    }
}

class Dog extends Animal
{
    void eat() throws Exception
    {
        System.out.println("Dog is eating meat.");
    }
}
```




```
class Animal
{
    void eat() throws Exception
    {
        System.out.println("Animal is eating.");
    }
}

class Dog extends Animal
{
    void eat() throws ArithmeticException
    {
        System.out.println("Dog is eating meat.");
    }
}
```




```
class Animal
{
    void eat() throws Exception
    {
        System.out.println("Animal is eating.");
    }
}

class Dog extends Animal
{
    void eat()
    {
        System.out.println("Dog is eating meat.");
    }
}
```



```
class Animal
{
    void eat() throws ArithmeticException
    {
        System.out.println("Animal is eating.");
    }
}

class Dog extends Animal
{
    void eat() throws Exception
    {
        System.out.println("Dog is eating meat.");
    }
}
```



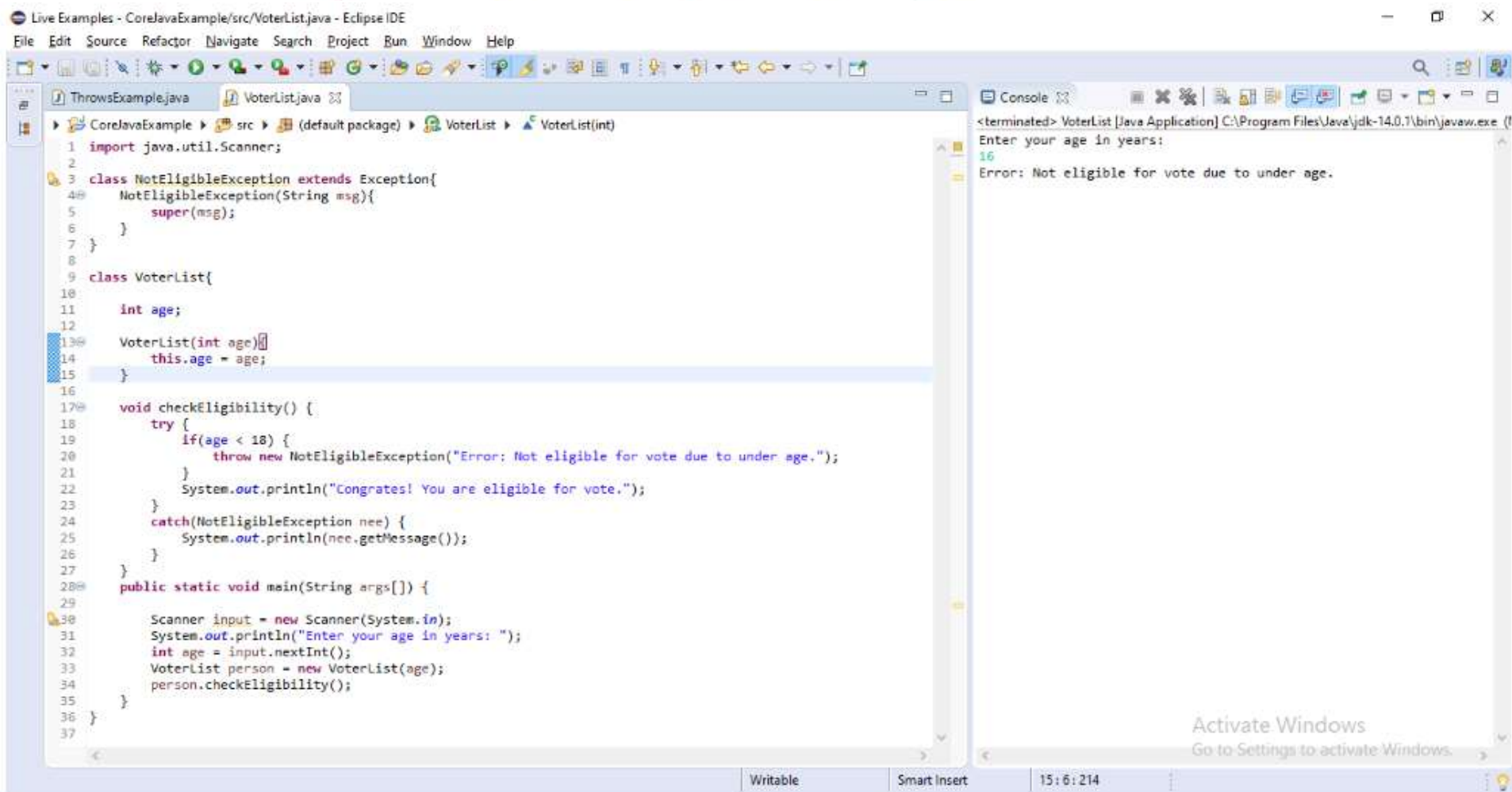
Creating Own Exceptions/Custom Exceptions

The Java programming language allow us to create our own exception classes which are basically subclasses built-in class `Exception`.

To create our own exception class simply create a class as a subclass of built-in `Exception` class.

We may create constructor in the user-defined exception class and pass a string to `Exception` class constructor using `super()`. We can use `getMessage()` method to access the string.

Let's look at the following Java code that illustrates the creation of user-defined exception.



The screenshot displays the Eclipse IDE with two main components: a Java editor on the left and a console on the right.

Java Editor (VoterList.java):

```
1 import java.util.Scanner;
2
3 class NotEligibleException extends Exception{
4     NotEligibleException(String msg){
5         super(msg);
6     }
7 }
8
9 class VoterList{
10
11     int age;
12
13     VoterList(int age){
14         this.age = age;
15     }
16
17     void checkEligibility() {
18         try {
19             if(age < 18) {
20                 throw new NotEligibleException("Error: Not eligible for vote due to under age.");
21             }
22             System.out.println("Congrates! You are eligible for vote.");
23         }
24         catch(NotEligibleException nee) {
25             System.out.println(nee.getMessage());
26         }
27     }
28     public static void main(String args[]) {
29
30         Scanner input = new Scanner(System.in);
31         System.out.println("Enter your age in years: ");
32         int age = input.nextInt();
33         VoterList person = new VoterList(age);
34         person.checkEligibility();
35     }
36 }
37 }
```

Console:

```
<terminated> VoterList [Java Application] C:\Program Files\Java\jdk-14.0.1\bin\javaw.exe (!
Enter your age in years:
16
Error: Not eligible for vote due to under age.
```

The console output shows the user entered '16', which triggered the custom exception, resulting in the message: "Error: Not eligible for vote due to under age."

Thread



Multithreading

The java programming language allows us to create a program that contains one or more parts that can run simultaneously at the same time. This type of program is known as a multithreading program. Each part of this program is called a thread. Every thread defines a separate path of execution in java. A thread is explained in different ways, and a few of them are as specified below.

A thread is a light weight process.

A thread may also be defined as follows.

A thread is a subpart of a process that can run individually.

In java, multiple threads can run at a time, which enables the java to write multitasking programs. The multithreading is a specialized form of multitasking. All modern operating systems support multitasking. There are two types of multitasking, and they are as follows.

- * Process-based multitasking
- * Thread-based multitasking

It is important to know the difference between process-based and thread-based multitasking. Let's distinguish both.

Process-based multitasking	Thread-based multitasking
It allows the computer to run two or more programs concurrently	It allows the computer to run two or more threads concurrently
In this process is the smallest unit.	In this thread is the smallest unit.
Process is a larger unit.	Thread is a part of process.
Process is heavy weight.	Thread is light weight.
Process requires separate address space for each.	Threads share same address space.
Process never gain access over idle time of CPU.	Thread gain access over idle time of CPU.
Inter process communication is expensive.	Inter thread communication is not expensive.

Thread Model

The java programming language allows us to create a program that contains one or more parts that can run simultaneously at the same time. This type of program is known as a multithreading program. Each part of this program is called a thread. Every thread defines a separate path of execution in java. A thread is explained in different ways, and a few of them are as specified below.

A thread is a light weight process.

A thread may also be defined as follows.

A thread is a subpart of a process that can run individually.

In java, a thread goes through different states throughout its execution. These stages are called thread life cycle states or phases. A thread may in any of the states like new, ready or runnable, running, blocked or wait, and dead or terminated state. The life cycle of a thread in java is shown in the following figure.



New

When a thread object is created using `new`, then the thread is said to be in the New state. This state is also known as Born state.

Example

```
Thread t1 = new Thread();
```

Runnable / Ready

When a thread calls `start()` method, then the thread is said to be in the Runnable state. This state is also known as a Ready state.

Example

```
t1.start();
```

Running

When a thread calls `run()` method, then the thread is said to be Running. The `run()` method of a thread called automatically by the `start()` method.

Blocked / Waiting

A thread in the Running state may move into the blocked state due to various reasons like `sleep()` method called, `wait()` method called, `suspend()` method called, and `join()` method called, etc.

When a thread is in the blocked or waiting state, it may move to Runnable state due to reasons like sleep time completed, waiting time completed, `notify()` or `notifyAll()` method called, `resume()` method called, etc.

Example

```
Thread.sleep(1000);  
wait(1000);  
wait();  
suspend();  
notify();  
notifyAll();  
resume();
```

Dead / Terminated

A thread in the Running state may move into the dead state due to either its execution completed or the `stop()` method called. The dead state is also known as the terminated state.

Creating threads

In java, a thread is a lightweight process. Every java program executes by a thread called the main thread. When a java program gets executed, the main thread created automatically. All other threads called from the main thread.

The java programming language provides two methods to create threads, and they are listed below.

- * Using Thread class (by extending Thread class)
- * Using Runnable interface (by implementing Runnable interface)

Let's see how to create threads using each of the above.

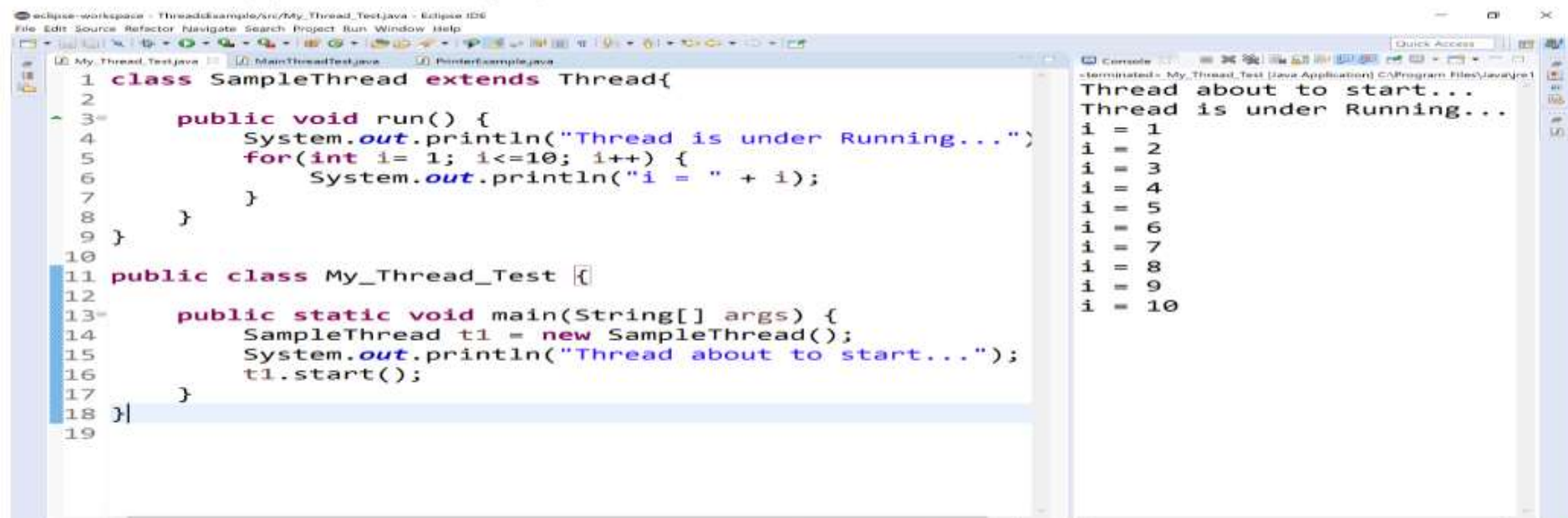
Extending Thread class

The java contains a built-in class Thread inside the java.lang package. The Thread class contains all the methods that are related to the threads.

To create a thread using Thread class, follow the step given below.

- * **Step-1:** Create a class as a child of Thread class. That means, create a class that extends Thread class.
- * **Step-2:** Override the run() method with the code that is to be executed by the thread. The run() method must be public while overriding.
- * **Step-3:** Create the object of the newly created class in the main() method.
- * **Step-4:** Call the start() method on the object created in the above step.

When we run this code, it produce the following output.



The screenshot shows the Eclipse IDE with two windows. The left window displays the source code for `My_Thread_Test.java`, and the right window shows the console output.

```
1 class SampleThread extends Thread{
2
3     public void run() {
4         System.out.println("Thread is under Running...");
5         for(int i= 1; i<=10; i++) {
6             System.out.println("i = " + i);
7         }
8     }
9 }
10
11 public class My_Thread_Test {
12
13     public static void main(String[] args) {
14         SampleThread t1 = new SampleThread();
15         System.out.println("Thread about to start...");
16         t1.start();
17     }
18 }
19
```

The console output on the right shows the execution results:

```
terminated: My_Thread_Test [Java Application] C:\Program Files\Java\jre1
Thread about to start...
Thread is under Running...
i = 1
i = 2
i = 3
i = 4
i = 5
i = 6
i = 7
i = 8
i = 9
i = 10
```

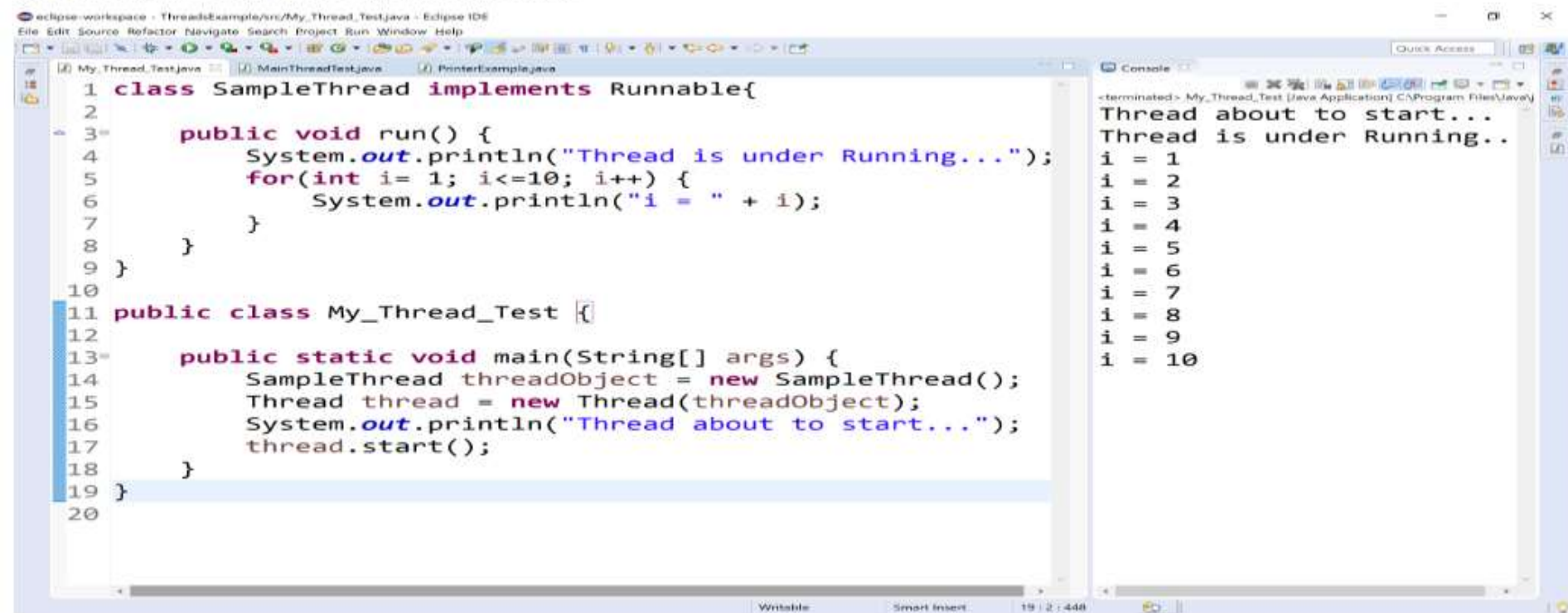
Implementing Runnable interface

The java contains a built-in interface Runnable inside the java.lang package. The Runnable interface implemented by the Thread class that contains all the methods that are related to the threads.

To create a thread using Runnable interface, follow the step given below.

- * **Step-1:** Create a class that implements Runnable interface.
- * **Step-2:** Override the run() method with the code that is to be executed by the thread. The run() method must be public while overriding.
- * **Step-3:** Create the object of the newly created class in the main() method.
- * **Step-4:** Create the Thread class object by passing above created object as parameter to the Thread class constructor.
- * **Step-5:** Call the start() method on the Thread class object created in the above step.

When we run this code, it produce the following output.



The screenshot shows the Eclipse IDE with two windows. The left window displays the source code for `My_Thread_Test.java`, and the right window shows the console output.

```
1 class SampleThread implements Runnable{
2
3     public void run() {
4         System.out.println("Thread is under Running...");
5         for(int i= 1; i<=10; i++) {
6             System.out.println("i = " + i);
7         }
8     }
9 }
10
11 public class My_Thread_Test {
12
13     public static void main(String[] args) {
14         SampleThread threadObject = new SampleThread();
15         Thread thread = new Thread(threadObject);
16         System.out.println("Thread about to start...");
17         thread.start();
18     }
19 }
20
```

The console output shows the following sequence of messages:

```
<terminated> My_Thread_Test [Java Application] C:\Program Files\Java\
Thread about to start...
Thread is under Running..
i = 1
i = 2
i = 3
i = 4
i = 5
i = 6
i = 7
i = 8
i = 9
i = 10
```

More about Thread class

The Thread class in java is a subclass of Object class and it implements Runnable interface. The Thread class is available inside the java.lang package. The Thread class has the following syntax.

```
class Thread extends Object implements Runnable{
    ...
}
```

The Thread class has the following constructors.

- * Thread()
- * Thread(String threadName)
- * Thread(Runnable objectName)
- * Thread(Runnable objectName, String threadName)

The Thread classs contains the following methods.

Method	Description	Return Value
run()	Defines actual task of the thread.	void
start()	It moves thre thread from Ready state to Running state by calling run() method.	void
setName(String)	Assigns a name to the thread.	void
getName()	Returns the name of the thread.	String
setPriority(int)	Assigns priority to the thread.	void
getPriority()	Returns the priority of the thread.	int
getId()	Returns the ID of the thread.	long
activeCount()	Returns total number of thread under active.	int
currentThread()	Returns the reference of the thread that currently in running state.	void
sleep(long)	moves the thread to blocked state till the specified number of milliseconds.	void
isAlive()	Tests if the thread is alive.	boolean
yield()	Tells to the scheduler that the current thread is willing to yield its current use of a processor.	void
join()	Waits for the thread to end.	void

⚠ The Thread class in java also contains methods like `stop()`, `destroy()`, `suspend()` and `resume()`. But they are deprecated.

Thread Scheduler

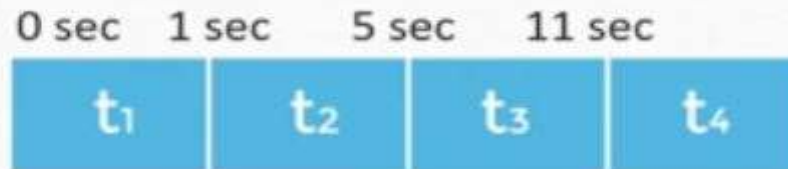
- A component of Java that decides which thread to run or execute and which thread to wait is called a **thread scheduler in Java**.
- In Java, a thread is only chosen by a thread scheduler if it is in the runnable state.
- **Each thread starts in a separate call stack.**
- There are two factors for scheduling a thread i.e. **Priority** and **Time of arrival**.
- **Priority:** Priority of each thread lies between 1 to 10. If a thread has a higher priority, it means that thread has got a better chance of getting picked up by the thread scheduler.
- **Time of Arrival:** Suppose two threads of the same priority enter the runnable state, then priority cannot be the factor to pick a thread from these two threads. In such a case, **arrival time** of thread is considered by the thread scheduler. A thread that arrived first gets the preference over the other threads.

Thread Scheduler Algorithms

- First Come First Serve Scheduling
- Time-slicing scheduling:
- Preemptive-Priority Scheduling:

FIRST COME FIRST SERVE SCHEDULING

Scheduler picks the threads that arrive first in the runnable queue. Observe the following table:



First Come First Serve Scheduling

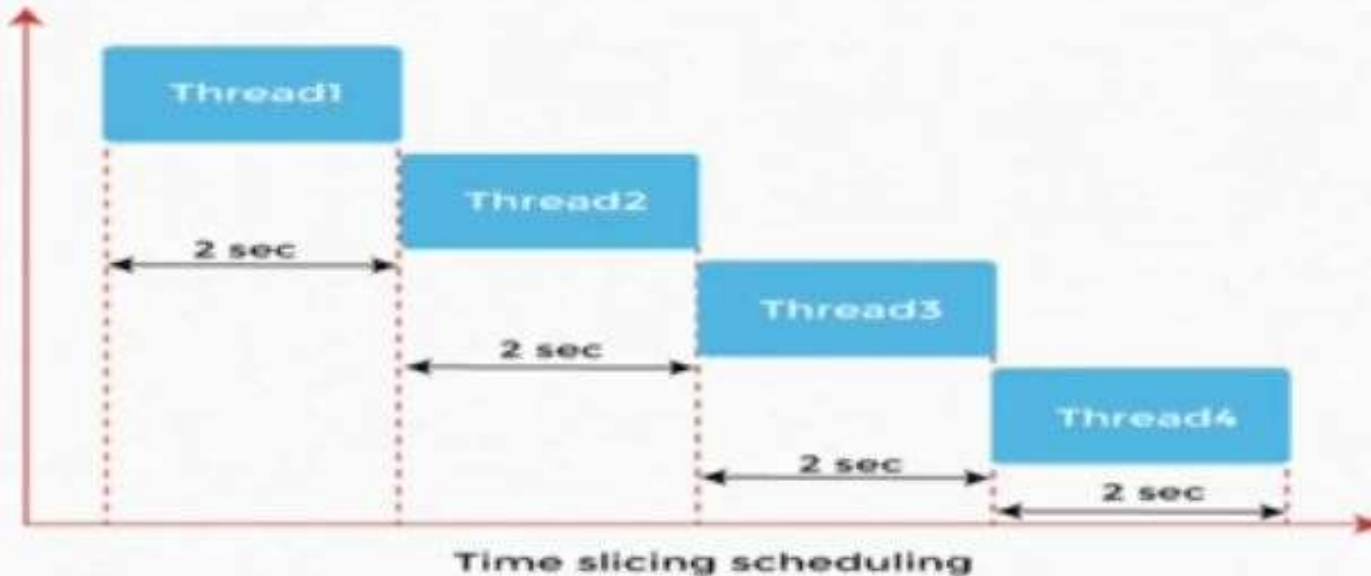
- Thread t1 will be processed first, and Thread t4 will be processed last.

DISADVANTAGE

- First Come First Serve algorithm is non-preemptive, which is bad as it may lead to infinite blocking (also known as starvation).
- Time-slices are provided to the threads so that after some time, the running thread has to give up the CPU. Thus, the other waiting threads also get time to run their job.

Time Slicing scheduling

- Each thread is given a time slice of 2 seconds.
- After every 2 seconds, the first thread leaves the CPU, and the CPU is then captured by Thread2.



DISADVANTAGE

- If the slicing time is short, processor output will be delayed.
- Performance depends heavily on time quantum.
- Priorities can't be fixed for processes.
- No priority to more important tasks.
- Finding an appropriate time quantum is quite difficult.

Thread Priority

In a java programming language, every thread has a property called priority. Most of the scheduling algorithms use the thread priority to schedule the execution sequence. In java, the thread priority range from 1 to 10. Priority 1 is considered as the lowest priority, and priority 10 is considered as the highest priority. The thread with more priority allocates the processor first.

The java programming language Thread class provides two methods `setPriority(int)`, and `getPriority()` to handle thread priorities.

The Thread class also contains three constants that are used to set the thread priority, and they are listed below.

- * `MAX_PRIORITY` - It has the value 10 and indicates highest priority.
- * `NORM_PRIORITY` - It has the value 5 and indicates normal priority.
- * `MIN_PRIORITY` - It has the value 1 and indicates lowest priority.

👉 The default priority of any thread is 5 (i.e. `NORM_PRIORITY`).

setPriority() method

The `setPriority()` method of Thread class used to set the priority of a thread. It takes an integer range from 1 to 10 as an argument and returns nothing (void).

The regular use of the `setPriority()` method is as follows.

Example

```
threadObject.setPriority(4);  
or  
threadObject.setPriority(MAX_PRIORITY);
```

getPriority() method

The `getPriority()` method of Thread class used to access the priority of a thread. It does not takes any argument and returns name of the thread as String.

The regular use of the `getPriority()` method is as follows.

Example

```
String threadName = threadObject.getPriority();
```


eclipse-workspace - ThreadsExample/src/My_Thread_Test.java - Eclipse IDE

File Edit Source Refactor Navigate Search Project Run Window Help

```
1 class SampleThread extends Thread{
2     public void run() {
3         System.out.println("Inside SampleThread");
4         System.out.println("Current Thread: " + Thread.currentThread().getName());
5     }
6 }
7
8 public class My_Thread_Test {
9
10    public static void main(String[] args) {
11        SampleThread threadObject1 = new SampleThread();
12        SampleThread threadObject2 = new SampleThread();
13        threadObject1.setName("first");
14        threadObject2.setName("second");
15
16        threadObject1.setPriority(4);
17        threadObject2.setPriority(Thread.MAX_PRIORITY);
18
19        threadObject1.start();
20        threadObject2.start();
21    }
22 }
23
24
```

Console

```
<terminated> My_Thread_Test [Java Application] C:\Program Fi
Inside SampleThread
Current Thread: second
Inside SampleThread
Current Thread: first
```

Writable Smart Insert 23 : 2 : 595

✶ In java, it is not guaranteed that threads execute according to their priority because it depends on JVM specification that which scheduling it chooses.

Thread Synchronization

The java programming language supports multithreading. The problem of shared resources occurs when two or more threads get execute at the same time. In such a situation, we need some way to ensure that the shared resource will be accessed by only one thread at a time, and this is performed by using the concept called synchronization.

➤ The synchronization is the process of allowing only one thread to access a shared resource at a time.

In java, the synchronization is achieved using the following concepts.

- ✳ Mutual Exclusion
- ✳ Inter thread communication

In this tutorial, we discuss mutual exclusion only, and the interthread communication will be discussed in the next tutorial.

Mutual Exclusion

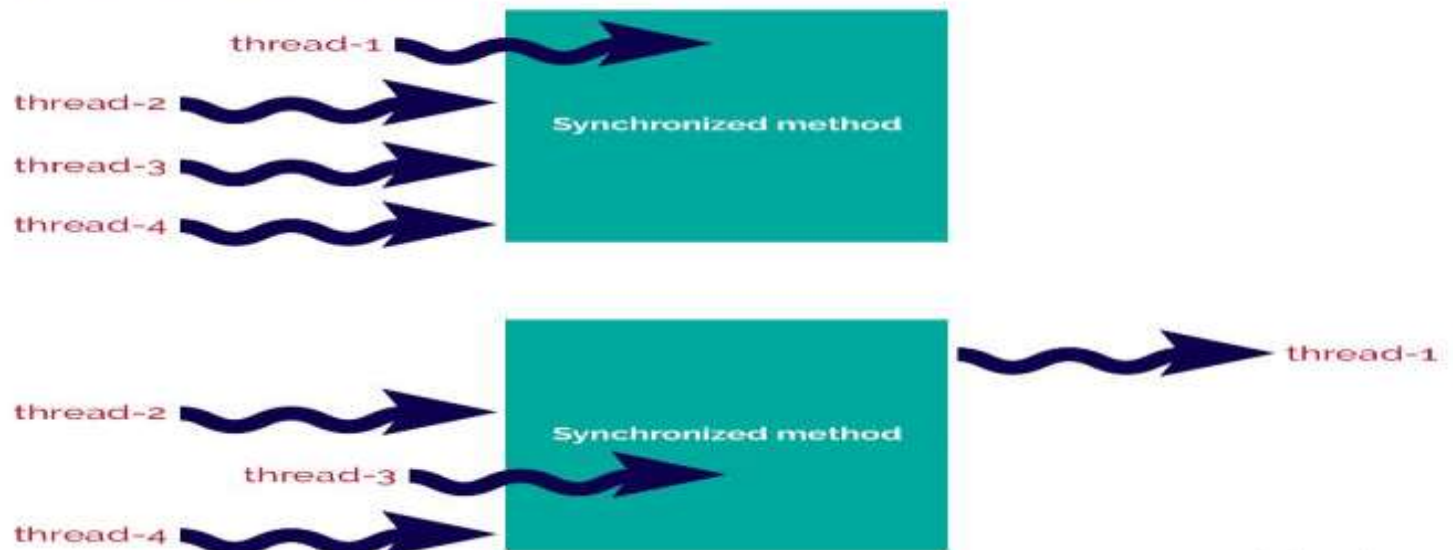
Using the mutual exclusion process, we keep threads from interfering with one another while they accessing the shared resource. In java, mutual exclusion is achieved using the following concepts.

- ✳ Synchronized method
- ✳ Synchronized block

Synchronized method

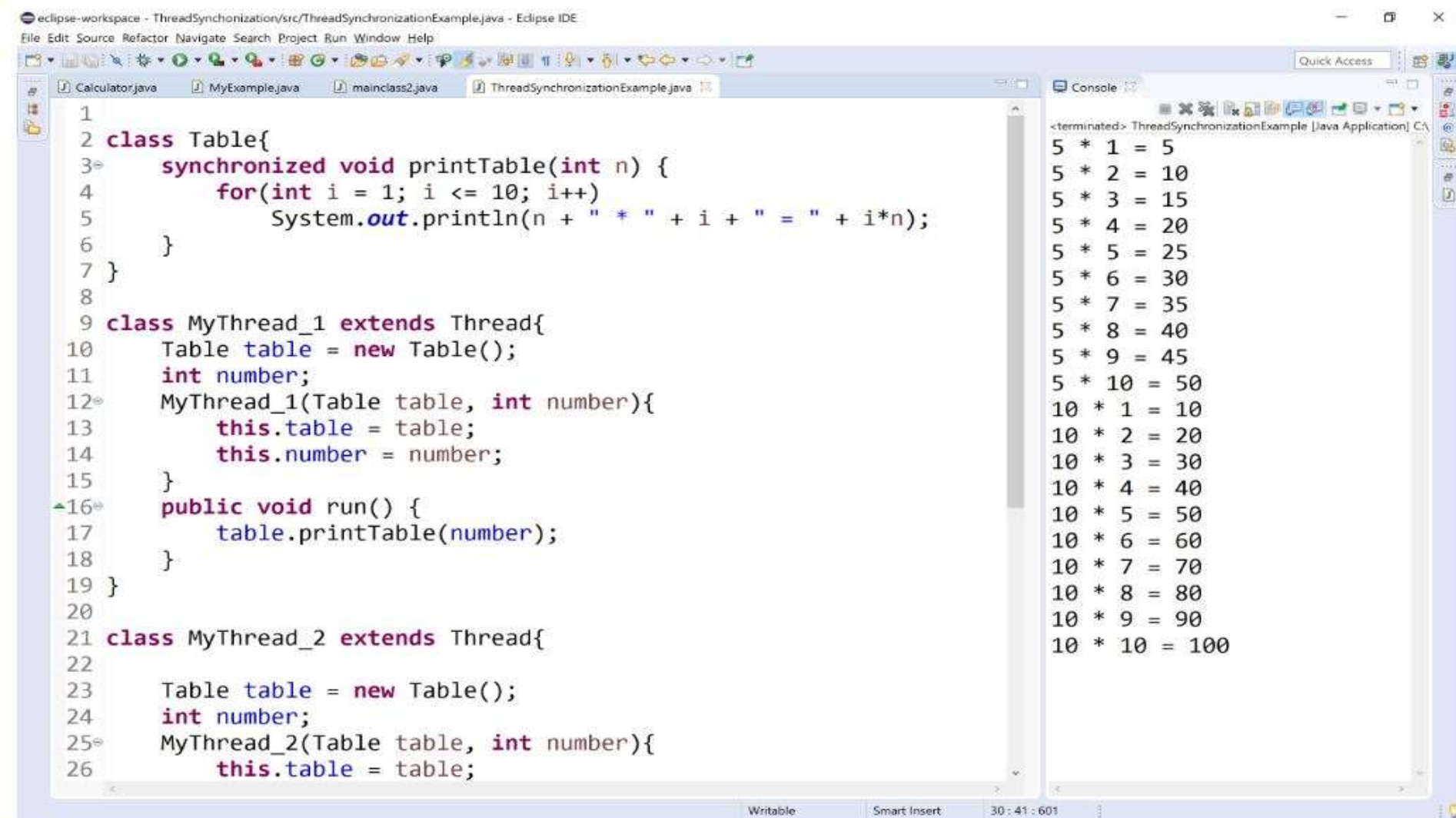
When a method created using a synchronized keyword, it allows only one object to access it at a time. When an object calls a synchronized method, it put a lock on that method so that other objects or thread that are trying to call the same method must wait, until the lock is released. Once the lock is released on the shared resource, one of the threads among the waiting threads will be allocated to the shared resource.

● ● ● Java thread execution with synchronized method



In the above image, initially the thread-1 is accessing the synchronized method and other threads (thread-2, thread-3, and thread-4) are waiting for the resource (synchronized method). When thread-1 completes its task, then one of the threads that are waiting is allocated with the synchronized method, in the above it is thread-3.

When we run this code, it produces the following output.



The screenshot shows the Eclipse IDE with a Java project named 'ThreadSynchronizationExample'. The code defines a 'Table' class with a synchronized 'printTable' method and two threads, 'MyThread_1' and 'MyThread_2', that use this method. The console output shows the results of these threads, demonstrating thread synchronization.

```
1
2 class Table{
3     synchronized void printTable(int n) {
4         for(int i = 1; i <= 10; i++)
5             System.out.println(n + " * " + i + " = " + i*n);
6     }
7 }
8
9 class MyThread_1 extends Thread{
10     Table table = new Table();
11     int number;
12     MyThread_1(Table table, int number){
13         this.table = table;
14         this.number = number;
15     }
16     public void run() {
17         table.printTable(number);
18     }
19 }
20
21 class MyThread_2 extends Thread{
22
23     Table table = new Table();
24     int number;
25     MyThread_2(Table table, int number){
26         this.table = table;
```

Console Output:

```
<terminated> ThreadSynchronizationExample [Java Application] C:\
5 * 1 = 5
5 * 2 = 10
5 * 3 = 15
5 * 4 = 20
5 * 5 = 25
5 * 6 = 30
5 * 7 = 35
5 * 8 = 40
5 * 9 = 45
5 * 10 = 50
10 * 1 = 10
10 * 2 = 20
10 * 3 = 30
10 * 4 = 40
10 * 5 = 50
10 * 6 = 60
10 * 7 = 70
10 * 8 = 80
10 * 9 = 90
10 * 10 = 100
```

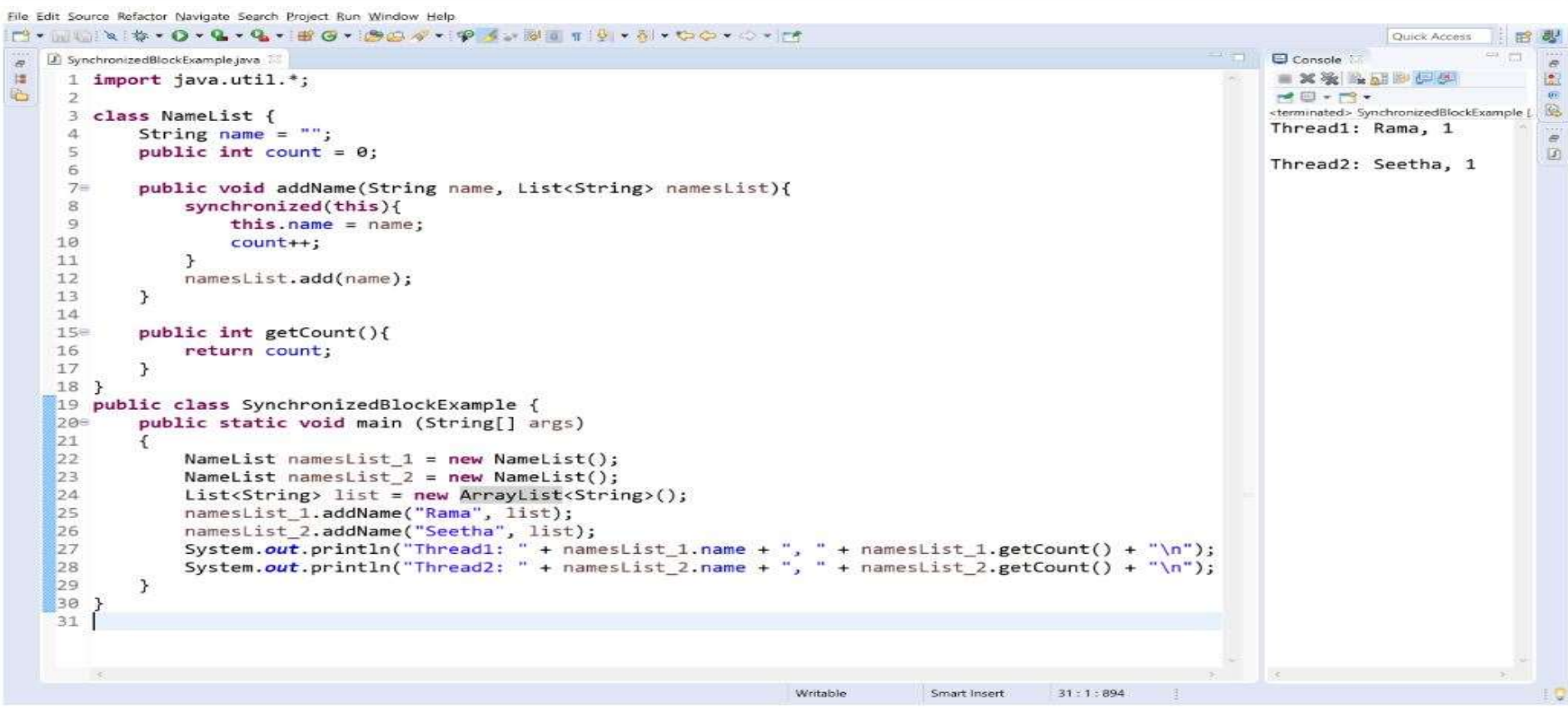
Synchronized block

The synchronized block is used when we want to synchronize only a specific sequence of lines in a method. For example, let's consider a method with 20 lines of code where we want to synchronize only a sequence of 5 lines code, we use the synchronized block.

The following syntax is used to define a synchronized block.

Syntax

```
synchronized(object){  
  
    ...  
    block code  
    ...  
}
```



The complete code of a method may be written inside the synchronized block, where it works similarly to the synchronized method.

Inter Thread Communication

Inter thread communication is the concept where two or more threads communicate to solve the problem of **polling**. In java, polling is the situation to check some condition repeatedly, to take appropriate action, once the condition is true. That means, in inter-thread communication, a thread waits until a condition becomes true such that other threads can execute its task. The inter-thread communication allows the synchronized threads to communicate with each other.

Java provides the following methods to achieve inter thread communication.

- * wait()
- * notify()
- * notifyAll()

The following table gives detailed description about the above methods.

Method	Description
void wait()	It makes the current thread to pause its execution until other thread in the same monitor calls notify()
void notify()	It wakes up the thread that called wait() on the same object.
void notifyAll()	It wakes up all the threads that called wait() on the same object.

👉 Calling notify() or notifyAll() does not actually give up a lock on a resource.

Let's look at an example problem of producer and consumer. The producer produces the item and the consumer consumes the same. But here, the consumer can not consume until the producer produces the item, and producer can not produce until the consumer consumes the item that already been produced. So here, the consumer has to wait until the producer produces the item, and the producer also needs to wait until the consumer consumes the same. Here we use the inter-thread communication to implement the producer and consumer problem.

The sample implementation of producer and consumer problem is as follows.



ProducerConsumer.java

```
1 class ItemQueue {
2     int item;
3     boolean valueSet = false;
4
5     synchronized int getItem()
6     {
7         while (!valueSet)
8             try {
9                 wait();
10            } catch (InterruptedException e) {
11                System.out.println("InterruptedException caught");
12            }
13        System.out.println("Consumed:" + item);
14        valueSet = false;
15        try {
16            Thread.sleep(1000);
17        } catch (InterruptedException e) {
18            System.out.println("InterruptedException caught");
19        }
20        notify();
21        return item;
22    }
23
24    synchronized void putItem(int item) {
25        while (valueSet)
26            try {
27                wait();
28            } catch (InterruptedException e) {
29                System.out.println("InterruptedException caught");
30            }
31        this.item = item;
32        valueSet = true;
33        System.out.println("Produced: " + item);
34        try {
35            Thread.sleep(1000);
36        } catch (InterruptedException e) {
37            System.out.println("InterruptedException caught");
38        }
39        notify();
40    }
41 }
42 }
43 }
```

Console

ProducerConsumer (1) [Java Application] C:\Program Files\Java\jre1.8.0_201\bin\javaw.exe (7 Ma

```
Produced: 0
Consumed:0
Produced: 1
Consumed:1
Produced: 2
Consumed:2
Produced: 3
Consumed:3
Produced: 4
Consumed:4
Produced: 5
Consumed:5
Produced: 6
Consumed:6
Produced: 7
Consumed:7
Produced: 8
Consumed:8
Produced: 9
Consumed:9
Produced: 10
Consumed:10
Produced: 11
Consumed:11
Produced: 12
Consumed:12
Produced: 13
Consumed:13
Produced: 14
Consumed:14
```

Daemon Threads

- Daemon thread is a low priority thread
- runs in background to perform tasks such as garbage collection.
- **Properties:**
 - JVM does not care whether Daemon thread is running or not.
 - It is an utmost low priority thread.
 - JVM terminates itself when all user threads finish their execution


METHODS

- **void setDaemon(boolean status)**
 - used to mark the current thread as daemon thread or user thread
 - Example:
 - `tD.setDaemon(false)` : tD is a user thread
 - `tD.setDaemon(True)` : tD becomes a Daemon Thread
- **boolean isDaemon():**
 - used to check that current is daemon.
 - It returns true if the thread is Daemon else it returns false.

```
// Java program to demonstrate the usage of
// setDaemon() and isDaemon() method.
public class Daemon1 extends Thread
{
    public Daemon1(String name){
        super(name);
    }

    public void run()
    {
        // Checking whether the thread is Daemon or not
        if(Thread.currentThread().isDaemon())
        {
            System.out.println(getName() + " is Daemon
thread");
        }

        else
        {
            System.out.println(getName() + " is User thread");
        }
    }
}
```




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

    Daemon1 t1 = new Daemon1("t1");
    Daemon1 t2 = new Daemon1("t2");
    Daemon1 t3 = new Daemon1("t3");

    // Setting user thread t1 to Daemon
    t1.setDaemon(true);

    // starting first 2 threads
    t1.start();
    t2.start();

    // Setting user thread t3 to Daemon
    t3.setDaemon(true);
    t3.start();

}
```



```
Administrator: C:\Windows\system32\cmd.exe

C:\Java\jdk1.8.0_181\bin>java Daemon1
t1 is Daemon thread
t3 is Daemon thread
t2 is User thread

C:\Java\jdk1.8.0_181\bin>
```

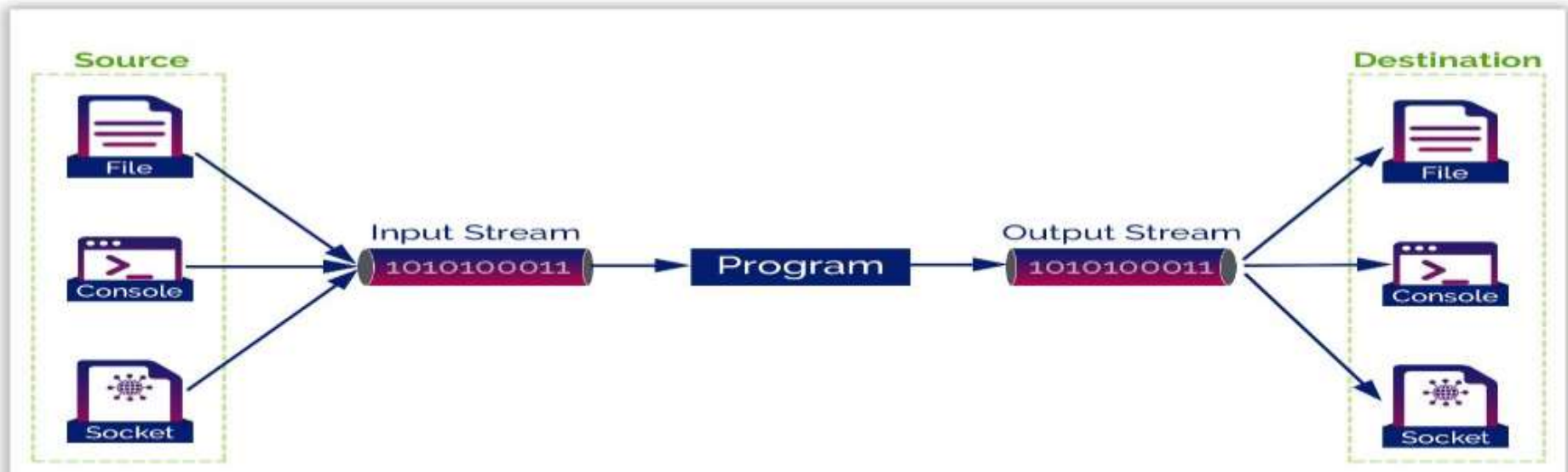
FILE I/O

Stream

In java, the IO operations are performed using the concept of streams. Generally, a stream means a continuous flow of data. In java, a stream is a logical container of data that allows us to read from and write to it. A stream can be linked to a data source, or data destination, like a console, file or network connection by java IO system. The stream-based IO operations are faster than normal IO operations.

The Stream is defined in the java.io package.

To understand the functionality of java streams, look at the following picture.



In java, the stream-based IO operations are performed using two separate streams input stream and output stream. The input stream is used for input operations, and the output stream is used for output operations. The java stream is composed of bytes.

In Java, every program creates 3 streams automatically, and these streams are attached to the console.

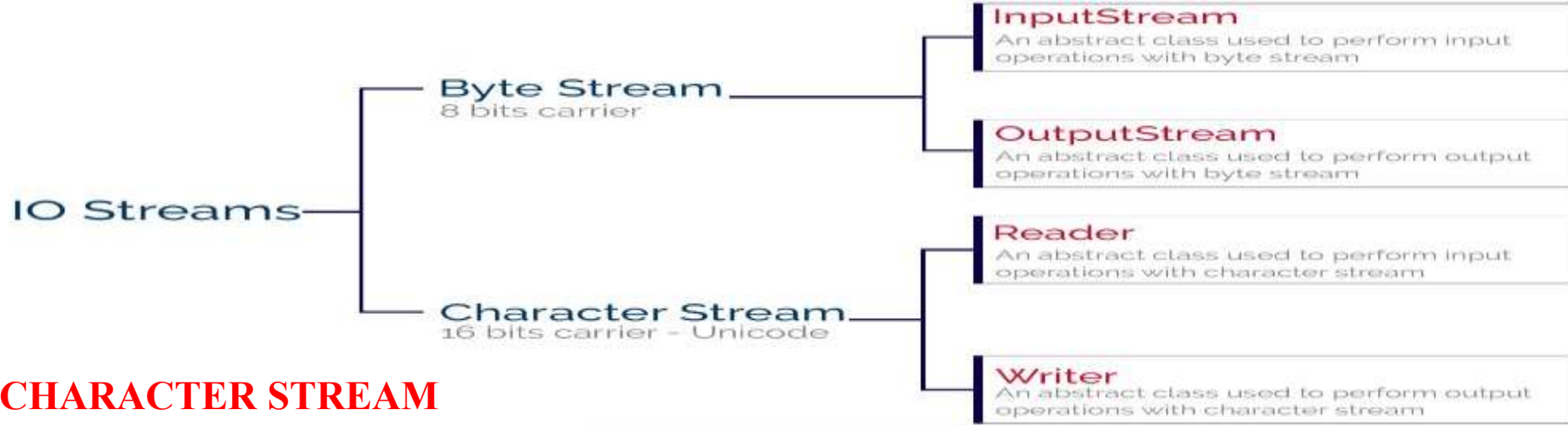
- * **System.out** : standard output stream for console output operations.
- * **System.in** : standard input stream for console input operations.
- * **System.err** : standard error stream for console error output operations.

The Java streams support many different kinds of data, including simple bytes, primitive data types, localized characters, and objects.

Java provides two types of streams, and they are as follows.

- * **Byte Stream**
- * **Character Stream**

The following picture shows how streams are categorized, and various built-in classes used by the java IO system.



CHARACTER STREAM

In java, when the IO stream manages 16-bit Unicode characters, it is called a character stream. The unicode set is basically a type of character set where each character corresponds to a specific numeric value within the given character set, and every programming language has a character set.

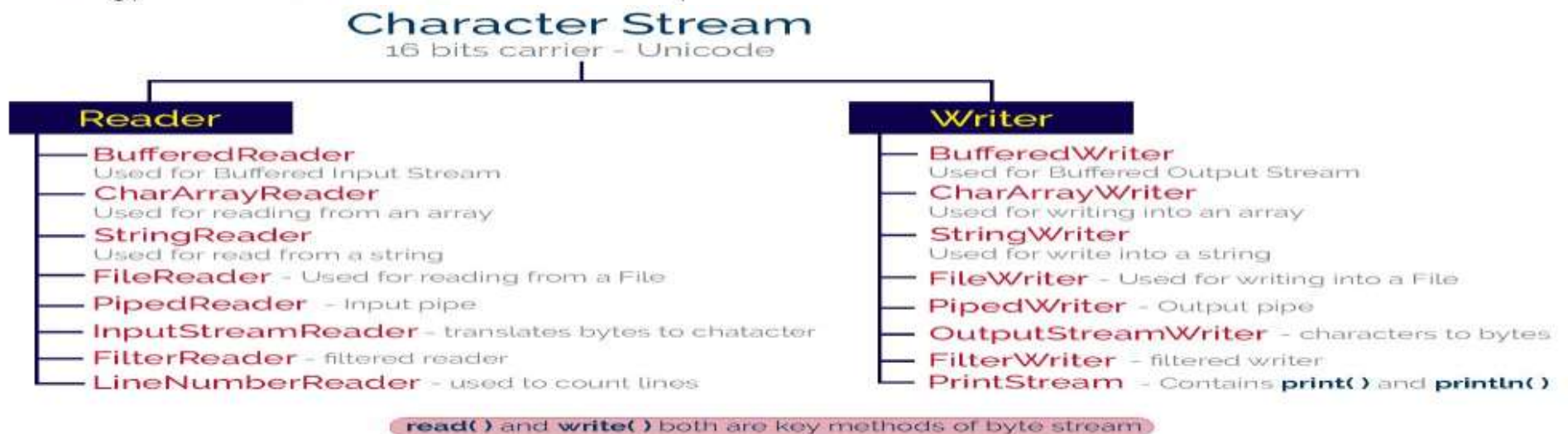
In java, the character stream is a 16 bits carrier. The character stream in java allows us to transmit 16 bits of data.

The character stream was introduced in Java 1.1 version. The character stream

The java character stream is defined by two abstract classes, **Reader** and **Writer**. The Reader class used for character stream based input operations, and the Writer class used for character stream based output operations.

The Reader and Writer classes have several concrete classes to perform various IO operations based on the character stream.

The following picture shows the classes used for character stream operations.



Reader class

The Reader class has defined as an abstract class, and it has the following methods which have implemented by its concrete classes.

S.No.	Method with Description
1	int read() It reads the next character from the input stream.
2	int read(char[] cbuffer) It reads a chunk of charaters from the input stream and store them in its byte array, cbuffer.
3	int read(char[] cbuf, int off, int len) It reads charaters into a portion of an array.
4	int read(CharBuffer target) It reads charaters into into the specified character buffer.
5	String readLine() It reads a line of text. A line is considered to be terminated by any oneof a line feed ('\n'), a carriage return ('\r'), or a carriage returnfollowed immediately by a linefeed.
6	boolean ready() It tells whether the stream is ready to be read.
7	void close() It closes the input stream and also frees any resources connected with this input stream.

Writer class

The Writer class has defined as an abstract class, and it has the following methods which have implemented by its concrete classes.

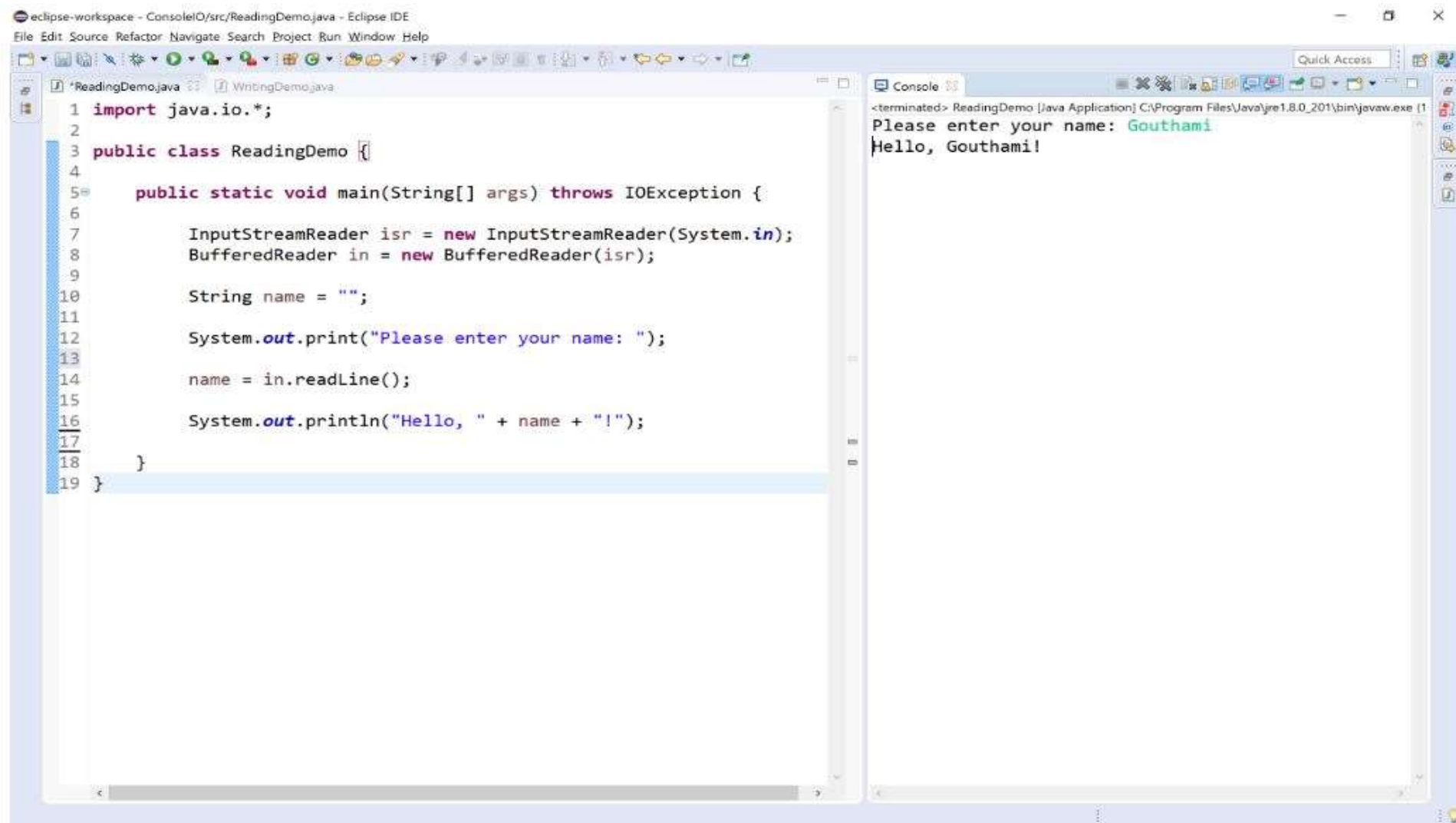
S.No.	Method with Description
1	void flush() It flushes the output steam by forcing out buffered bytes to be written out.
2	void write(char[] cbuf) It writes a whole array(cbuf) to the output stream.
3	void write(char[] cbuf, int off, int len) It writes a portion of an array of characters.
4	void write(int c) It writes single character.
5	void write(String str) It writes a string.
6	void write(String str, int off, int len) It writes a portion of a string.
7	Writer append(char c) It appends the specified character to the writer.
8	Writer append(CharSequence csq) It appends the specified character sequence to the writer.
9	Writer append(CharSequence csq, int start, int end) It appends a subsequence of the specified character sequence to the writer.
10	void close() It closes the output stream and also frees any resources connected with this output stream.

Reading data using BufferedReader

We can use the `BufferedReader` class to read data from the console. The `BufferedReader` class needs `InputStreamReader` class. The `BufferedReader` use a method `read()` to read a value from the console, or file, or socket.

Let's look at an example code to illustrate reading data using `BufferedReader`.

Example 1 - Reading from console



The screenshot shows the Eclipse IDE with a Java project named 'ConsoleIO'. The main editor displays the source code for 'ReadingDemo.java'. The code imports `java.io.*`, defines a `ReadingDemo` class with a `main` method that uses `InputStreamReader` and `BufferedReader` to read a name from the console and prints a greeting. The console window on the right shows the program's execution, displaying the prompt 'Please enter your name: ' followed by the input 'Gouthami' and the output 'Hello, Gouthami!'.

```
1 import java.io.*;
2
3 public class ReadingDemo {
4
5     public static void main(String[] args) throws IOException {
6
7         InputStreamReader isr = new InputStreamReader(System.in);
8         BufferedReader in = new BufferedReader(isr);
9
10        String name = "";
11
12        System.out.print("Please enter your name: ");
13
14        name = in.readLine();
15
16        System.out.println("Hello, " + name + "!");
17    }
18 }
19 }
```

Console Output:

```
<terminated> ReadingDemo [Java Application] C:\Program Files\Java\jre1.8.0_201\bin\javaw.exe [1
Please enter your name: Gouthami
Hello, Gouthami!
```

Example 2 - Reading from a file

eclipse-workspace - Console/O/src/ReadingDemo.java - Eclipse IDE

File Edit Source Refactor Navigate Search Project Run Window Help

When we run the above program, it produce the following output.

eclipse-workspace - Console/O/src/ReadingDemo.java - Eclipse IDE

File Edit Source Refactor Navigate Search Project Run Window Help

```
1 import java.io.*;
2
3 public class ReadingDemo {
4
5     public static void main(String[] args) throws IOException {
6
7         Reader in = new FileReader("C:\\\\Raja\\\\dataFile.txt");
8
9         try {
10             char c = (char)in.read();
11             System.out.println("Data read from a file - '" + c + "'");
12         }
13         catch(Exception e) {
14             System.out.println(e);
15         }
16         finally {
17             in.close();
18         }
19     }
20 }
```

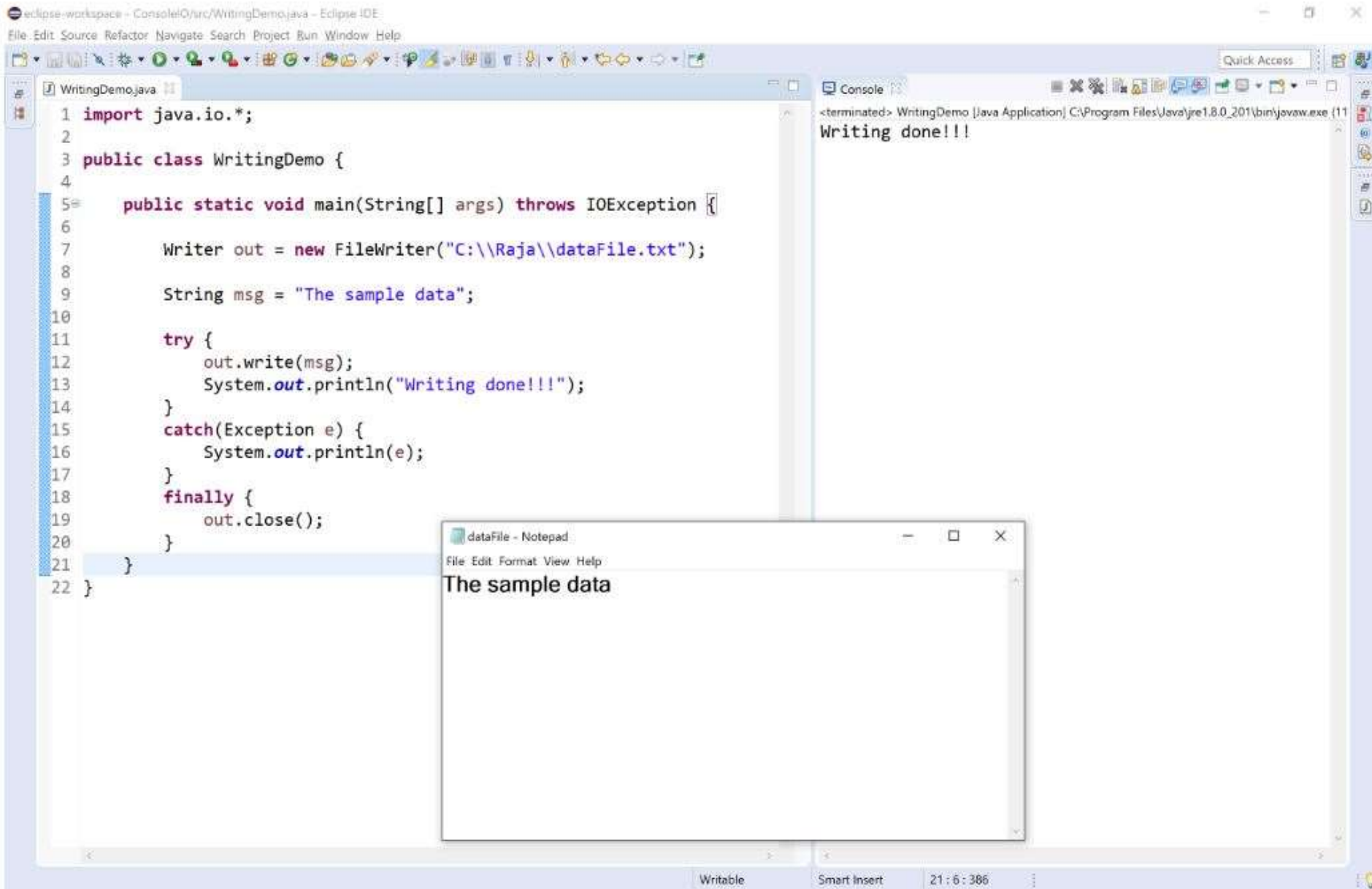
Console
<terminated> ReadingDemo [Java Application] C:\Program Files\Java\jre1.8.0_201\bin\javaw.exe (1)
Data read from a file - 'J'

dataFile - Notepad

File Edit Format View Help

Java tutorials by BTech Smart Class

Writing data using File Writer



File Handling using Character Stream

In java, we can use a character stream to handle files. The character stream has the following built-in classes to perform various operations on a file.

- * **FileReader** - It is a built-in class in java that allows reading data from a file. This class has implemented based on the character stream. The `FileReader` class provides a method `read()` to read data from a file character by character.
- * **FileWriter** - It is a built-in class in java that allows writing data to a file. This class has implemented based on the character stream. The `FileWriter` class provides a method `write()` to write data to a file character by character.

Let's look at the following example program that reads data from a file and writes the same to another file using `FileReader` and `FileWriter` classes.

When we run the above program, it produce the following output.

The screenshot displays the Eclipse IDE environment. The main editor shows the source code for `FileIO.java`, which implements a program to read from `Input-File.txt` and write to `Output-File.txt`. The code includes imports for `java.io.*`, class declarations, and a `main` method that uses `FileReader` and `FileWriter` within a try-catch-finally block to handle exceptions and ensure resource closure.

```
1 import java.io.*;
2 public class FileIO {
3
4     public static void main(String args[]) throws IOException {
5         FileReader in = null;
6         FileWriter out = null;
7
8         try {
9             in = new FileReader("C:\\Raja\\Input-File.txt");
10            out = new FileWriter("C:\\Raja\\Output-File.txt");
11
12            int c;
13            while ((c = in.read()) != -1) {
14                out.write(c);
15            }
16            System.out.println("Reading and Writing in a file is done!!!");
17        }
18        catch(Exception e) {
19            System.out.println(e);
20        }
21        finally {
22            if (in != null) {
23                in.close();
24            }
25            if (out != null) {
26                out.close();
27            }
28        }
29    }
30 }
```

The **Console** window shows the execution output: `<terminated> FileIO [Java Application] C:\Program Files\Java\jre1.8.0_201\bin\javaw.exe (12 A) Reading and Writing in a file is done!!!`

Two Notepad windows are open: `Input-File - Notepad` and `Output-File - Notepad`. Both files contain the text: `Java Tutorials - BTech Smart Class`, demonstrating that the data was successfully read from the input file and written to the output file.

Random Access File

In java, the `java.io` package has a built-in class `RandomAccessFile` that enables a file to be accessed randomly. The `RandomAccessFile` class has several methods used to move the cursor position in a file.

A random access file behaves like a large array of bytes stored in a file.

RandomAccessFile Constructors

The `RandomAccessFile` class in java has the following constructors.

S.No.	Constructor with Description
1	<code>RandomAccessFile(File fileName, String mode)</code> It creates a random access file stream to read from, and optionally to write to, the file specified by the <code>File</code> argument.
2	<code>RandomAccessFile(String fileName, String mode)</code> It creates a random access file stream to read from, and optionally to write to, a file with the specified <code>fileName</code> .

Access Modes

Using the `RandomAccessFile`, a file may be created in the following modes.

- * `r` - Creates the file with read mode; Calling write methods will result in an `IOException`.
- * `rw` - Creates the file with read and write mode.
- * `rwd` - Creates the file with read and write mode - synchronously. All updates to file content are written to the disk synchronously.
- * `rws` - Creates the file with read and write mode - synchronously. All updates to file content or meta data are written to the disk synchronously.

RandomAccessFile methods

The `RandomAccessFile` class in java has the following methods.

RandomAccessFile methods

The RandomAccessFile class in java has the following methods.

S.No.	Methods with Description
1	int read() It reads byte of data from a file. The byte is returned as an integer in the range 0-255.
2	int read(byte[] b) It reads byte of data from file upto b.length, -1 if end of file is reached.
3	int read(byte[] b, int offset, int len) It reads bytes initialising from offset position upto b.length from the buffer.
4	boolean readBoolean() It reads a boolean value from from the file.
5	byte readByte() It reads signed eight-bit value from file.
6	char readChar() It reads a character value from file.
7	double readDouble() It reads a double value from file.
8	float readFloat() It reads a float value from file.
9	long readLong() It reads a long value from file.
10	int readInt() It reads a integer value from file.
11	void readFully(byte[] b) It reads bytes initialising from offset position upto b.length from the buffer.
12	void readFully(byte[] b, int offset, int len) It reads bytes initialising from offset position upto b.length from the buffer.
13	String readUTF() t reads in a string from the file.
14	void seek(long pos) It sets the file-pointer(cursor) measured from the beginning of the file, at which the next read or write occurs.
15	long length() It returns the length of the file.
16	void write(int b) It writes the specified byte to the file from the current cursor position.
17	void writeFloat(float v) It converts the float argument to an int using the floatToIntBits method in class Float, and then writes that int value to the file as a four-byte quantity, high byte first.
18	void writeDouble(double v) It converts the double argument to a long using the doubleToLongBits method in class Double, and then writes that long value to the file as an eight-byte quantity, high byte first.

100

Scanner class

The **Scanner** is a built-in class in java used for read the input from the user in java programming. The Scanner class is defined inside the **java.util** package.

The Scanner class implements **Iterator** interface.

The Scanner class provides the easiest way to read input in a Java program.

🔹 The Scanner object breaks its input into tokens using a delimiter pattern, the default delimiter is whitespace.

The Scanner class in java has the following constructors.

S.No.	Constructor with Description
1	Scanner(InputStream source) It creates a new Scanner that produces values read from the specified input stream.
2	Scanner(InputStream source, String charsetName) It creates a new Scanner that produces values read from the specified input stream.
3	Scanner(File source) It creates a new Scanner that produces values scanned from the specified file.
4	Scanner(File source, String charsetName) It creates a new Scanner that produces values scanned from the specified file.
5	Scanner(String source) It creates a new Scanner that produces values scanned from the specified string.
6	Scanner(Readable source) It creates a new Scanner that produces values scanned from the specified source.
7	Scanner(ReadableByteChannel source) It creates a new Scanner that produces values scanned from the specified channel.
8	Scanner(ReadableByteChannel source, String charsetName) It creates a new Scanner that produces values scanned from the specified channel.

The Scanner class in java has the following methods:

S.No.	Methods with Description
1	String next() It reads the next complete token from the invoking scanner.
2	String next(Pattern pattern) It reads the next token if it matches the specified pattern.
3	String next(String pattern) It reads the next token if it matches the pattern constructed from the specified string.
4	boolean nextBoolean() It reads a boolean value from the user.
5	byte nextByte() It reads a byte value from the user.
6	double nextDouble() It reads a double value from the user.
7	float nextFloat() It reads a floating-point value from the user.
8	int nextInt() It reads an integer value from the user.
9	long nextLong() It reads a long value from the user.
10	short nextShort() It reads a short value from the user.
11	String nextLine() It reads a string value from the user.
12	boolean hasNext() It returns true if the invoking scanner has another token in its input.

When we execute the above code, it produce the following output.

eclipse-workspace - JavaCollectionFramework/src/ScannerClassExample.java - Eclipse IDE

File Edit Source Refactor Navigate Search Project Run Window Help



```
1 import java.util.Scanner;
2
3 public class ScannerClassExample {
4
5     public static void main(String[] args) {
6
7         Scanner read = new Scanner(System.in); // Input stream is used
8
9         System.out.print("Enter any name: ");
10        String name = read.next();
11
12        System.out.print("Enter your age in years: ");
13        int age = read.nextInt();
14
15        System.out.print("Enter your salary: ");
16        double salary = read.nextDouble();
17
18        System.out.print("Enter any message: ");
19        read = new Scanner(System.in);
20        String msg = read.nextLine();
21
22        System.out.println("\n-----");
23        System.out.println("Hello, " + name);
24        System.out.println("You are " + age + " years old.");
25        System.out.println("You are earning Rs." + salary + " per month.");
26        System.out.println("Words from " + name + " - " + msg);
27    }
28 }
29
```

Console

<terminated> ScannerClassExample [Java Application] C:\Program Files\Java\jre1.8.0

```
Enter any name: Raja
Enter your age in years: 32
Enter your salary: 30000
Enter any message: Good luck
```

```
-----
Hello, Raja
You are 32 years old.
You are earning Rs.30000.0 per month.
Words from Raja - Good luck
```