Exception Handling in Java

The **Exception Handling in Java** is one of the powerful *mechanism to handle the runtime errors* so that the normal flow of the application can be maintained.

In Java, an exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.

Advantage of Exception Handling

The core advantage of exception handling is **to maintain the normal flow of the application**. An exception normally disrupts the normal flow of the application; that is why we need to handle exceptions.

```
statement 1;
statement 2;
statement 3;
statement 4;
statement 5;//exception occurs
statement 6;
statement 7;
statement 8;
statement 9;
statement 10;
```

Suppose there are 10 statements in a Java program and an exception occurs at statement 5; the rest of the code will not be executed, i.e., statements 6 to 10 will not be executed. However, when we perform exception handling, the rest of the statements will be executed. That is why we use exception handling in Java.

Major reasons why an exception Occurs

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

Types of Exceptions

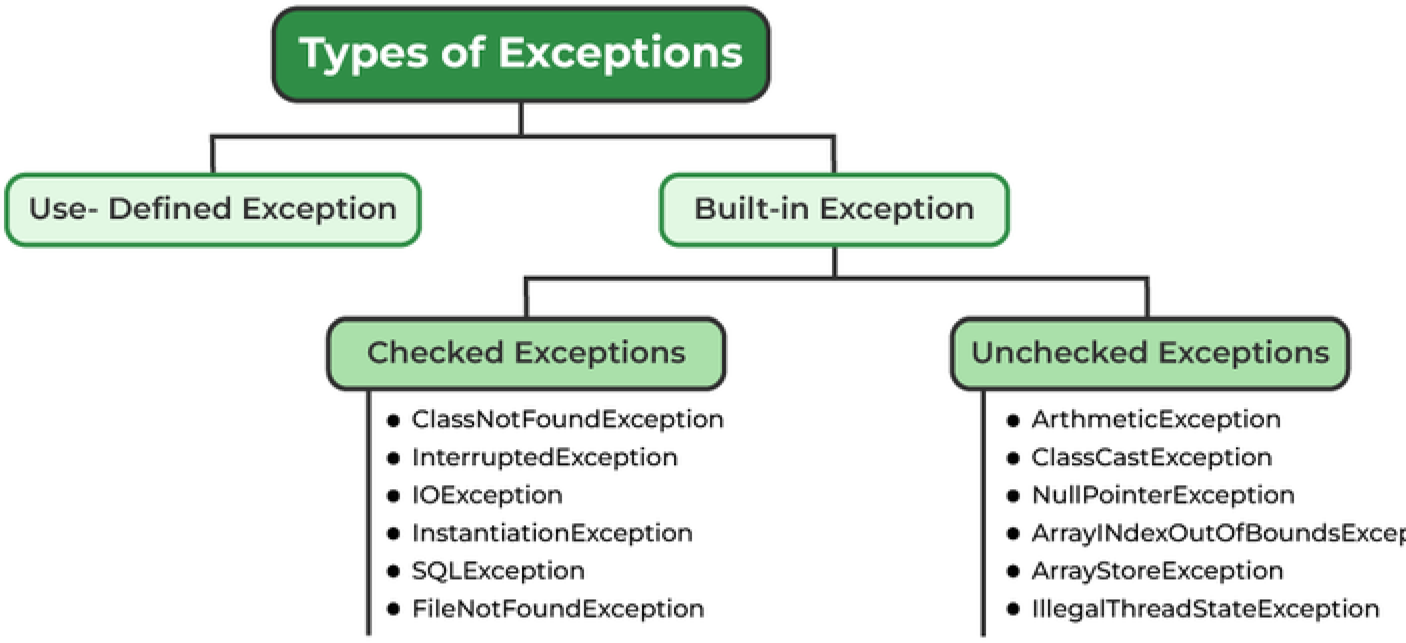
Java defines several types of exceptions that relate to its various class libraries. Java also allows users to define their own exceptions.

1. Built-in Exceptions

Built-in exceptions are the exceptions that are available in Java libraries. These exceptions are suitable to explain certain error situations.

- **Checked Exceptions:** Checked exceptions are called compile-time exceptions because these exceptions are checked at compile-time by the compiler.

- **Unchecked Exceptions:** The unchecked exceptions are just opposite to the checked exceptions. The compiler will not check these exceptions at compile time. In simple words, if a program throws an unchecked exception, and even if we didn't handle or declare it, the program would not give a compilation error.



1. Built-in Exceptions

Built-in exceptions are the exceptions that are available in Java libraries. These exceptions are suitable to explain certain error situations.

- **Checked Exceptions:** Checked exceptions are called compile-time exceptions because these exceptions are checked at compile-time by the

compiler.

- **Unchecked Exceptions:** The unchecked exceptions are just opposite to the checked exceptions. The compiler will not check these exceptions at compile time. In simple words, if a program throws an unchecked exception, and even if we didn't handle or declare it, the program would not give a compilation error.

Java try...catch

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 throws keyword

Syntax of Java throws

```
return_type method_name() throws exception_class_name{
    //method code
    }
```

1. try: The `try` block is used to enclose a segment of code that might throw an exception, ensuring that any exception arising from the enclosed code can be gracefully managed.

2. catch: The `catch` block follows the try block and defines how to handle specific types of exceptions. Multiple catch blocks can be used after a single try to handle different exception types individually.

3. throw: The `throw` keyword is employed to manually trigger or throw an exception from the code. It's often used in conjunction with user-defined or system exceptions to indicate when something abnormal occurs.

4. throws: Used in method signatures, the `throws` keyword indicates that the method might throw specified exceptions.
t's a way of notifying callers that they should be prepared to handle (or further propagate) these exceptions.

5. finally: The `finally` block, used after the try-catch structure, ensures that a particular segment of code runs regardless of whether an exception was thrown in the try block. This is typically used for cleanup operations, like closing resources.

Let's discuss each of the above 5 exception-handling keywords with syntax and examples.

1. try Block

Enclose the code that might throw an exception within a `try` block. If an exception occurs within the try block, that exception is handled by an exception handler associated with it. The try block contains at least one `catch` block or `finally` block.

The syntax of the try-catch block:

```
try{
//code that may throw exception
}catch(Exception_class_Name ref){}
```

The syntax of a try-finally block:

```
try{
//code that may throw exception
}finally{}
```

Example:

```
try {
    int result = 10 / 0;
} catch (ArithmeticException e) {
    System.out.println(e.getMessage());
}
```

In the above code, dividing by zero will cause an *`ArithmeticException`*. The statements inside the try block are where we anticipate this error might occur.

Nested try block

The try block within a try block is known as a nested try block in java.

```
public class NestedTryBlock {

         public static void main(String args[]) {
                 try {
                         try {
                                 System.out.println(" This gives divide by zero 
error");
                                 int b = 39 / 0;
                         } catch (ArithmeticException e) {
                                 System.out.println(e);
                         }

                         try {
                                 System.out.println(" This gives Array index out of 
bound exception");
                                 int a[] = new int[5];
                                 a[5] = 4;
                         } catch (ArrayIndexOutOfBoundsException e) {
                                 System.out.println(e);
                         }

                         System.out.println("other statement");
                 } catch (Exception e) {
                         System.out.println("handeled");
```

```
                }

                System.out.println("normal flow..");
        }
}
```

2. catch Block

Java catch block is used to handle the Exception. It must be used after the *try* block only. You can use multiple catch blocks with a single try.

Syntax:

```
try
{
      //code that cause exception;
}
catch(Exception_type  e)
{
      //exception handling code
}
```

Examples:

Example 1: catch *ArithmeticException* exception:

```
public class Arithmetic {

 public static void main(String[] args) {

  try {
      int result = 30 / 0; // Trying to divide by zero
  } catch (ArithmeticException e) {
       System.out.println("ArithmeticException caught!");
  }
      System.out.println("rest of the code executes");
 }
}
```

Output:

```
ArithmeticException caught!
rest of the code executes
```

Example 2: catch *ArrayIndexOutOfBoundsException* exception:

```
try {
    int[] arr = {1, 2, 3};
    System.out.println(arr[5]);
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("Array index is out of bounds!");
}
```

In this code, we're trying to access an index that doesn't exist in the array. The corresponding catch block catches this exception and handles it by printing a custom error message.

Multi-catch Block

In some cases, more than one exception could be raised by a single piece of code. To handle this type of situation, you can specify two or more catch clauses, each catching a different type of exception.

When an exception is thrown, each catch statement is inspected in order, and the first one whose type matches that of the exception is executed. After one catch statement executes, the others are bypassed, and execution continues after the try/catch block.

Example: Consider a scenario where we want to parse an integer from a string and then use that integer as an array index. Both of these operations can throw exceptions - **NumberFormatException** and **ArrayIndexOutOfBoundsException**.

```
public class MultiCatchExample {

    public static void main(String[] args) {
        String numStr = "10a";  // This will cause NumberFormatException
        int[] numbers = {1, 2, 3, 4, 5};

        try {
            int num = Integer.parseInt(numStr);   // Parsing integer from string
            System.out.println(numbers[num]);     // Accessing array element
        } catch (NumberFormatException | ArrayIndexOutOfBoundsException e) {
            System.out.println("An error occurred: " + e.getMessage());
        }
    }
}
```

Output:

```
An error occurred: For input string: "10a"
```

In the example above, the try block contains two statements, each of which can throw an exception. The catch block can handle both exceptions because of the multi-catch feature.

3. throw Keyword

The *throw* keyword is used to explicitly throw an exception from a method or any block of code. We can throw either checked or unchecked exceptions using the *throw* keyword. The throw keyword is followed by an instance of the exception.

Syntax:

```
throw exception_instance;
```

Example:

Let's consider a simple example where we have a method *setAge()* that sets the age of a person. If someone tries to set a negative age, it's clearly an incorrect value. In such a case, we can throw an **IllegalArgumentException**.

```
public class ThrowExample {

    private int age;

    public void setAge(int age) {
        if (age < 0) {
            throw new IllegalArgumentException("Age cannot be negative!");
        }
        this.age = age;
    }

    public static void main(String[] args) {
        ThrowExample person = new ThrowExample();

        try {
            person.setAge(-5);   // This will cause an exception
        } catch (IllegalArgumentException e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
}
```

Output:

```
Error: Age cannot be negative!
```

In the *setAge* method, if the age provided is negative, we throw an **IllegalArgumentException** with a relevant message.

4. throws Keyword

The *throws* keyword is used to declare exceptions. It doesn't throw an exception but specifies that a method might throw exceptions. It's typically used to inform callers of the exceptions they might encounter.

Syntax:

```
return_type method_name() throws exception_class_name{
//method code
}
```

Example:

```
public class ExceptionHandlingWorks {

    public static void main(String[] args) {
        exceptionHandler();
    }

    private static void exceptionWithoutHandler() throws IOException {
        try (BufferedReader reader = new BufferedReader(new FileReader(new
File("/invalid/file/location")))) {
            int c;
            // Read and display the file.
            while ((c = reader.read()) != -1) {
                System.out.println((char) c);
            }
        }
    }

    private static void exceptionWithoutHandler1() throws IOException {
        exceptionWithoutHandler();
    }

    private static void exceptionWithoutHandler2() throws IOException {
        exceptionWithoutHandler1();
    }

    private static void exceptionHandler() {
        try {
            exceptionWithoutHandler2();
        } catch (IOException e) {
            System.out.println("IOException caught!");
        }
    }
}
```

5. finally Block

- Java *finally* block is a block that is used to execute important code such as closing connection, stream, etc.
- Java *finally* block is always executed whether an exception is handled or not.
- Java *finally* block follows try or catch block.
- For each try block, there can be zero or more catch blocks, but only one *finally* block.
- The *finally* block will not be executed if the program exits(either by calling *System.exit()* or by causing a fatal error that causes the process to abort).

Syntax:

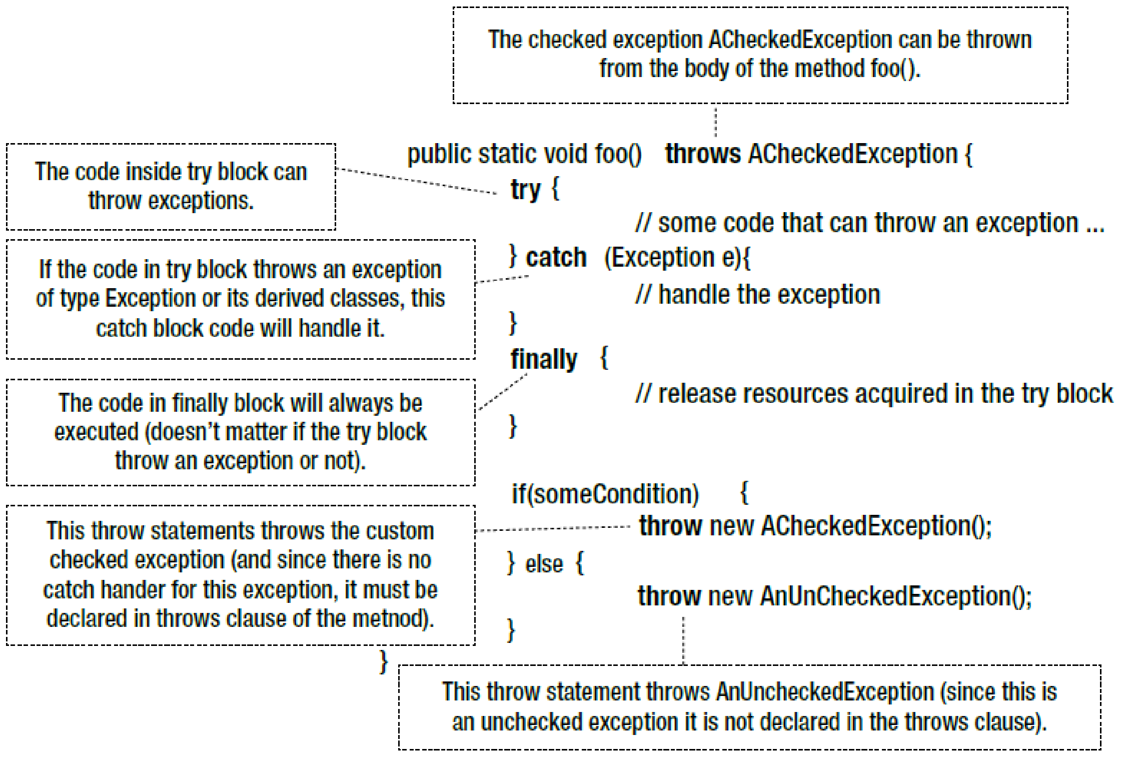
```
try {
    // Code that might throw an exception
} catch (ExceptionType1 e1) {
    // Code to handle ExceptionType1
} catch (ExceptionType2 e2) {
    // Code to handle ExceptionType2
}
// ... more catch blocks if necessary ...
finally {
    // Code to be executed always, whether an exception occurred or not
}
```

Example 1:

In this example, we have used **FileInputStream** to read the *simple.txt* file. After reading a file the resource **FileInputStream** should be closed by using *finally* block.

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

        FileInputStream fis = null;
        try {
            File file = new File("sample.txt");
            fis = new FileInputStream(file);
            int content;
            while ((content = fis.read()) != -1) {
                // convert to char and display it
                System.out.print((char) content);
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if (fis != null) {
                try {
                    fis.close();
                } catch (IOException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
            }
            }
        }
    }
```

This diagram summarizes the usage of try, catch, throw, throws, and finally keywords:



The checked exception ACheckedException can be thrown from the body of the method foo().

The code inside try block can throw exceptions.

If the code in try block throws an exception of type Exception or its derived classes, this catch block code will handle it.

The code in finally block will always be executed (doesn't matter if the try block throw an exception or not).

This throw statements throws the custom checked exception (and since there is no catch hander for this exception, it must be declared in throws clause of the metnod).

This throw statement throws AnUncheckedException (since this is an unchecked exception it is not declared in the throws clause).

```
public static void foo()   throws ACheckedException {
    try {
            // some code that can throw an exception ...
    } catch  (Exception e){
            // handle the exception
    }
    finally  {
            // release resources acquired in the try block
    }

    if(someCondition)    {
            throw new ACheckedException();
    } else  {
            throw new AnUnCheckedException();
    }
}
```

Sr. no.	Basis of Differences	throw	throws
1.	Definition	Java throw keyword is used throw an exception explicitly in the code, inside the function or the block of code.	Java throws keyword is used the method signature to decla an exception which might thrown by the function while th execution of the code.

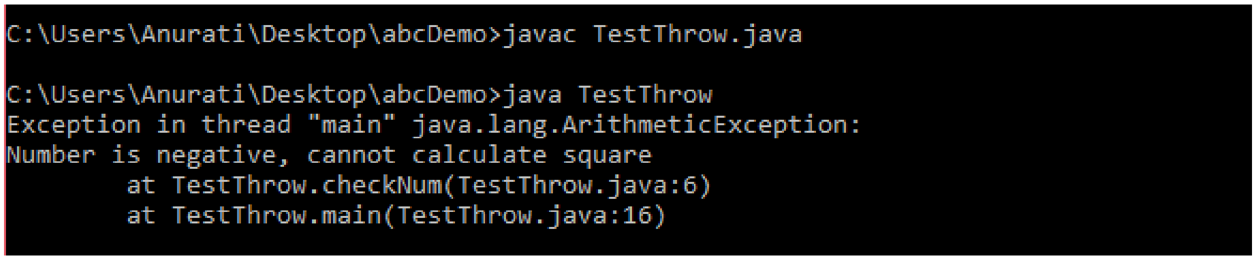
2.	Type of exception Using throw keyword, we can only propagate unchecked exception i.e., the checked exception cannot be propagated using throw only.	Using throws keyword, we can declare both checked and unchecked exceptions. However, the throws keyword can be used to propagate checked exceptions only.	
3.	Syntax	The throw keyword is followed by an instance of Exception to be thrown.	The throws keyword is followe by class names of Exceptions t be thrown.
4.	Declaration	throw is used within the method.	throws is used with the metho signature.
5.	Internal implementation	We are allowed to throw only one exception at a time i.e. we cannot throw multiple exceptions.	We can declare multip exceptions using thro keyword that can be thrown b the method. For example, main throws IOExceptio SQLException.

Java throw Example

TestThrow.java

```
public class TestThrow {
  //defining a method
  public static void checkNum(int num) {
    if (num < 1) {
        throw new ArithmeticException("\nNumber is negative, cannot calculate squ
are");
    }
    else {
      System.out.println("Square of " + num + " is " + (num*num));
    }
  }
  //main method
```

```
    public static void main(String[] args) {
        TestThrow obj = new TestThrow();
        obj.checkNum(-3);
        System.out.println("Rest of the code..");
    }
}
```



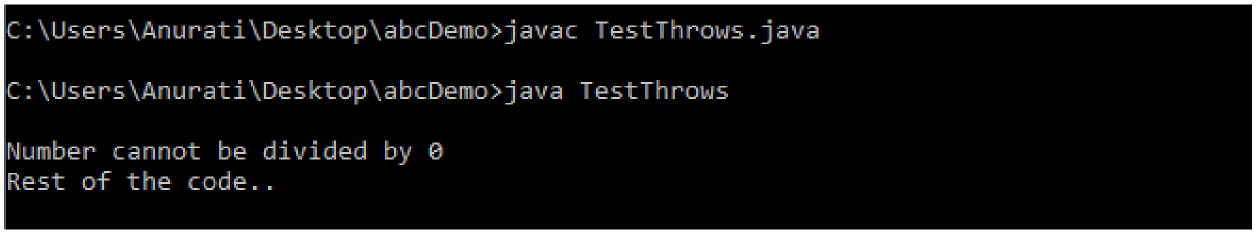
```
C:\Users\Anurati\Desktop\abcDemo>javac TestThrow.java

C:\Users\Anurati\Desktop\abcDemo>java TestThrow
Exception in thread "main" java.lang.ArithmeticException:
Number is negative, cannot calculate square
        at TestThrow.checkNum(TestThrow.java:6)
        at TestThrow.main(TestThrow.java:16)
```

Java throws Example

TestThrows.java

```
public class TestThrows {
    //defining a method
    public static int divideNum(int m, int n) throws ArithmeticException {
        int div = m / n;
        return div;
    }
    //main method
    public static void main(String[] args) {
        TestThrows obj = new TestThrows();
        try {
            System.out.println(obj.divideNum(45, 0));
        }
        catch (ArithmeticException e){
            System.out.println("\nNumber cannot be divided by 0");
        }

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

Output:



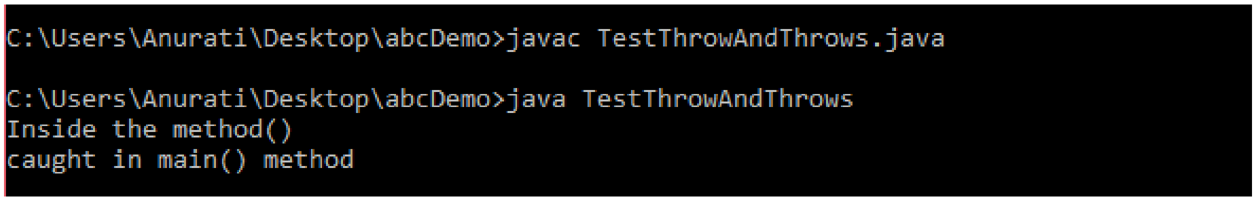
```
C:\Users\Anurati\Desktop\abcDemo>javac TestThrows.java

C:\Users\Anurati\Desktop\abcDemo>java TestThrows

Number cannot be divided by 0
Rest of the code..
```

Java throw and throws Example

TestThrowAndThrows.java

```
public class TestThrowAndThrows
{
    // defining a user-defined method
    // which throws ArithmeticException
    static void method() throws ArithmeticException
    {
        System.out.println("Inside the method()");
        throw new ArithmeticException("throwing ArithmeticException");
    }
    //main method
    public static void main(String args[])
    {
        try
        {
            method();
        }
        catch(ArithmeticException e)
        {
            System.out.println("caught in main() method");
            }
        }
    }
```

Output:



```
C:\Users\Anurati\Desktop\abcDemo>javac TestThrowAndThrows.java

C:\Users\Anurati\Desktop\abcDemo>java TestThrowAndThrows
Inside the method()
caught in main() method
```

you can use the Throw keyword to throw an exception explicitly in the code. In contrast, you can use the Throws keyword to declare that a method might throw an exception in the code.

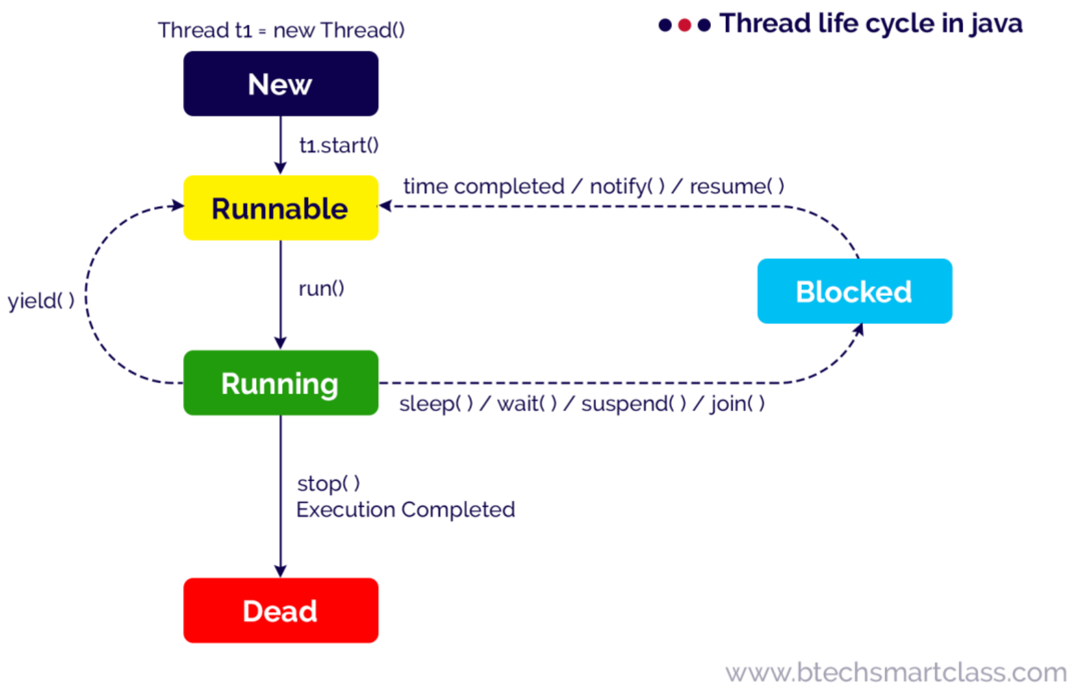
S. No.	**Key Difference**	**throw**	**throws**
1.	Point of Usage	The **throw** keyword is used inside a function. It is used when it is required to throw an Exception logically.	The **throws** keyword is used in the method signature. It is used when the method has some statements that can lead to exceptions.
2.	Exceptions Thrown	The **throw** keyword is used to throw an exception explicitly. It can throw only one exception at a time.	The **throws** keyword can be used to declare multiple exceptions, separated by a comma. Whichever exception occurs, if matched with the declared ones, is thrown automatically then.
3.	Syntax	Syntax of **throw** keyword includes the instance of the Exception to be thrown. Syntax wise throw keyword is followed by the instance variable.	Syntax of **throws** keyword includes the class names of the Exceptions to be thrown. Syntax wise throws keyword is followed by exception class names.

S. No.	Key Difference	throw	throws
4.	Propagation of Exceptions	**throw** keyword cannot propagate checked exceptions. It is only used to propagate the unchecked Exceptions that are not checked using the throws keyword.	**throws** keyword is used to propagate the checked Exceptions only.

Java 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 a light wieght process. 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();
suspened();
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.

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

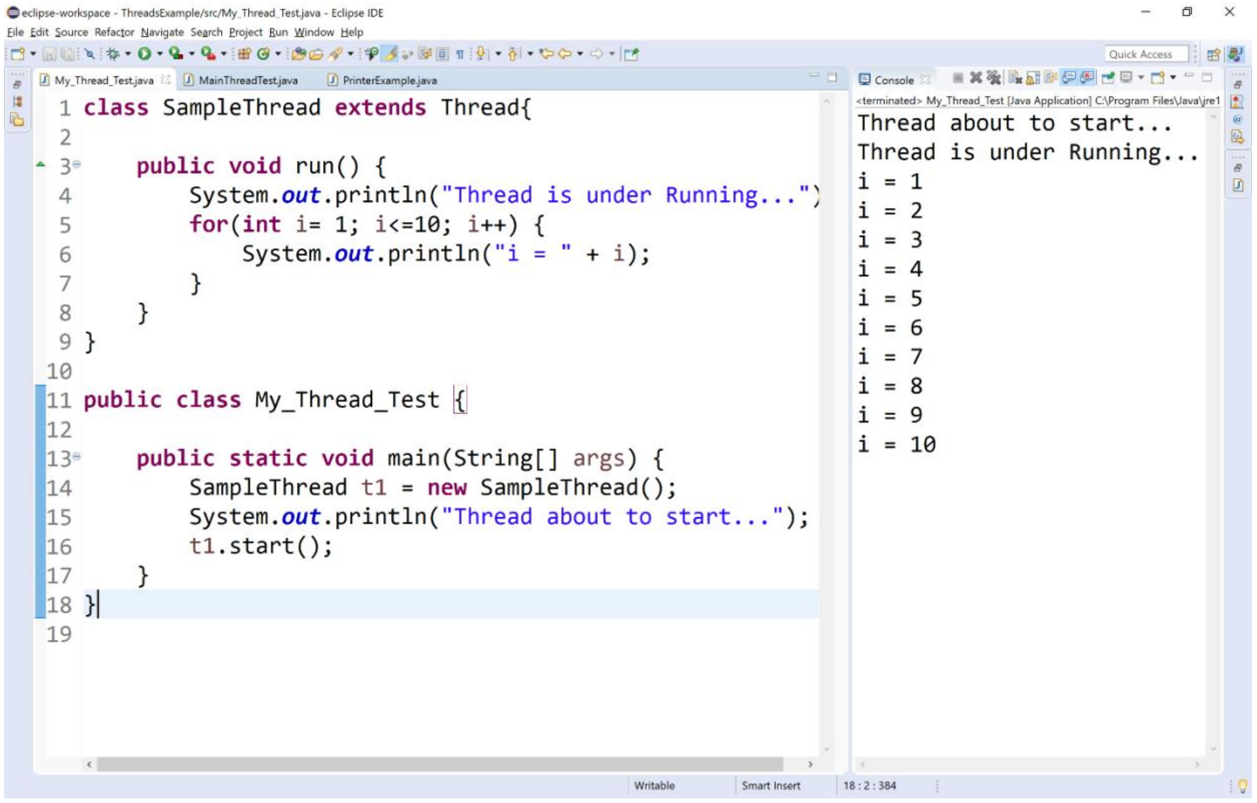
- **Using Thread class (by extending Thread class)**
- **Uisng Runnable interface (by implementing Runnable interface)**

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.



eclipse-workspace - ThreadsExample/src/My_Thread_Test.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
My_Thread_Test.java
MainThreadTest.java
PrinterExample.java
Console
<terminated> My_Thread_Test [Java Application] C:\Program Files\Java\jre1
Writable
Smart Insert
18 : 2 : 384

```java
class SampleThread extends Thread{

    public void run() {
        System.out.println("Thread is under Running...")
        for(int i= 1; i<=10; i++) {
            System.out.println("i = " + i);
        }
    }
}

public class My_Thread_Test {

    public static void main(String[] args) {
        SampleThread t1 = new SampleThread();
        System.out.println("Thread about to start...");
        t1.start();
    }
}

```

```
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
```

Implementng Runnable interface

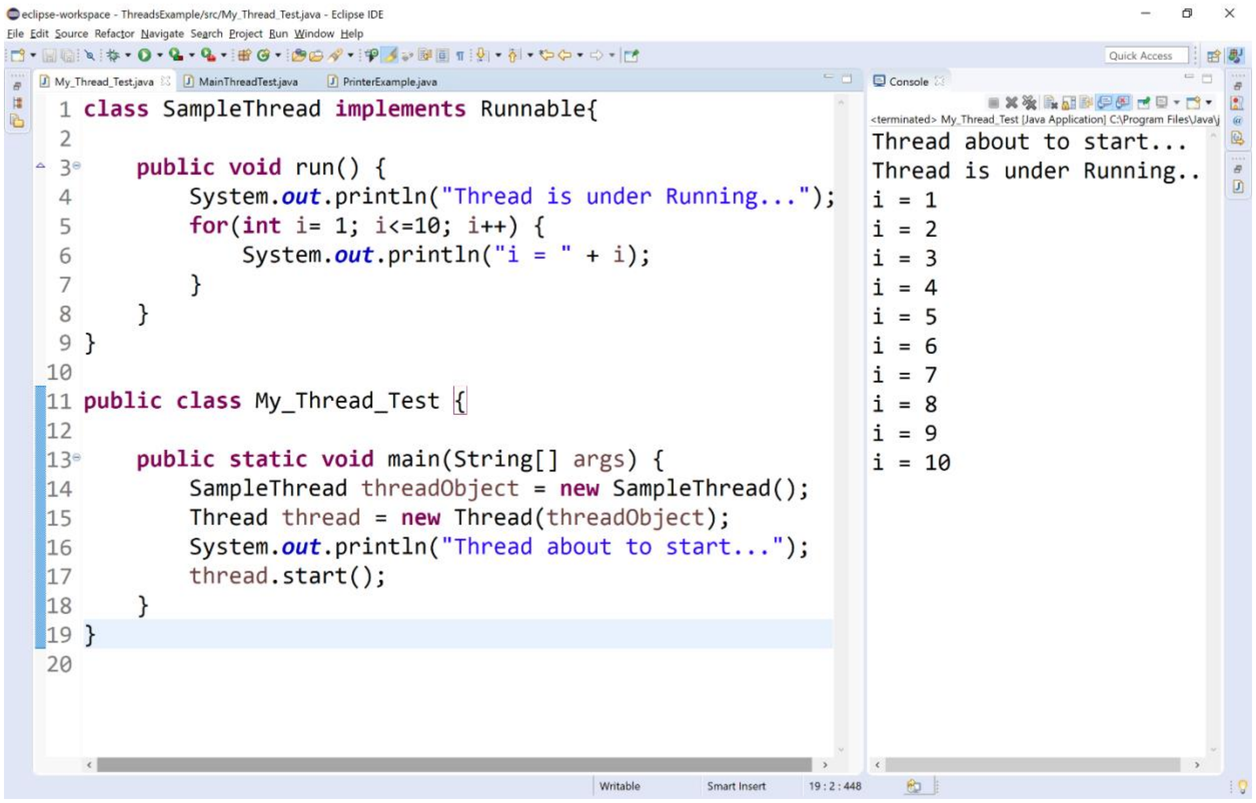
The java contains a built-in interface Runnable inside the java.lang package. The Runnable interface implemented by the Thread class that contains all the methods that are related to the threads.

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.

Look at the following example program.


eclipse-workspace - ThreadsExample/src/My_Thread_Test.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
My_Thread_Test.java
MainThreadTest.java
PrinterExample.java
Console
<terminated> My_Thread_Test [Java Application] C:\Program Files\Java\j
Writable
Smart Insert
19 : 2 : 448

```java
class SampleThread implements Runnable{

    public void run() {
        System.out.println("Thread is under Running...");
        for(int i= 1; i<=10; i++) {
            System.out.println("i = " + i);
        }
    }
}

public class My_Thread_Test {

    public static void main(String[] args) {
        SampleThread threadObject = new SampleThread();
        Thread thread = new Thread(threadObject);
        System.out.println("Thread about to start...");
        thread.start();
    }
}
```

```
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
```

The Thread classs contains the following methods.

Method	**Description**
run( )	Defines actual task of the thread.
start( )	It moves the thread from Ready state to Running state by calling run( ) metho
setName(String)	Assigns a name to the thread.

Method	Description
getName( )	Returns the name of the thread.
setPriority(int)	Assigns priority to the thread.
getPriority( )	Returns the priority of the thread.
getId( )	Returns the ID of the thread.
activeCount()	Returns total number of thread under active.
currentThread( )	Returns the reference of the thread that currently in running state.
sleep(long)	moves the thread to blocked state till the specified number of milliseconds.
isAlive( )	Tests if the thread is alive.
yield( )	Tells to the scheduler that the current thread is willing to yield its current use o processor.
join( )	Waits for the thread to end.

Java 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);
```

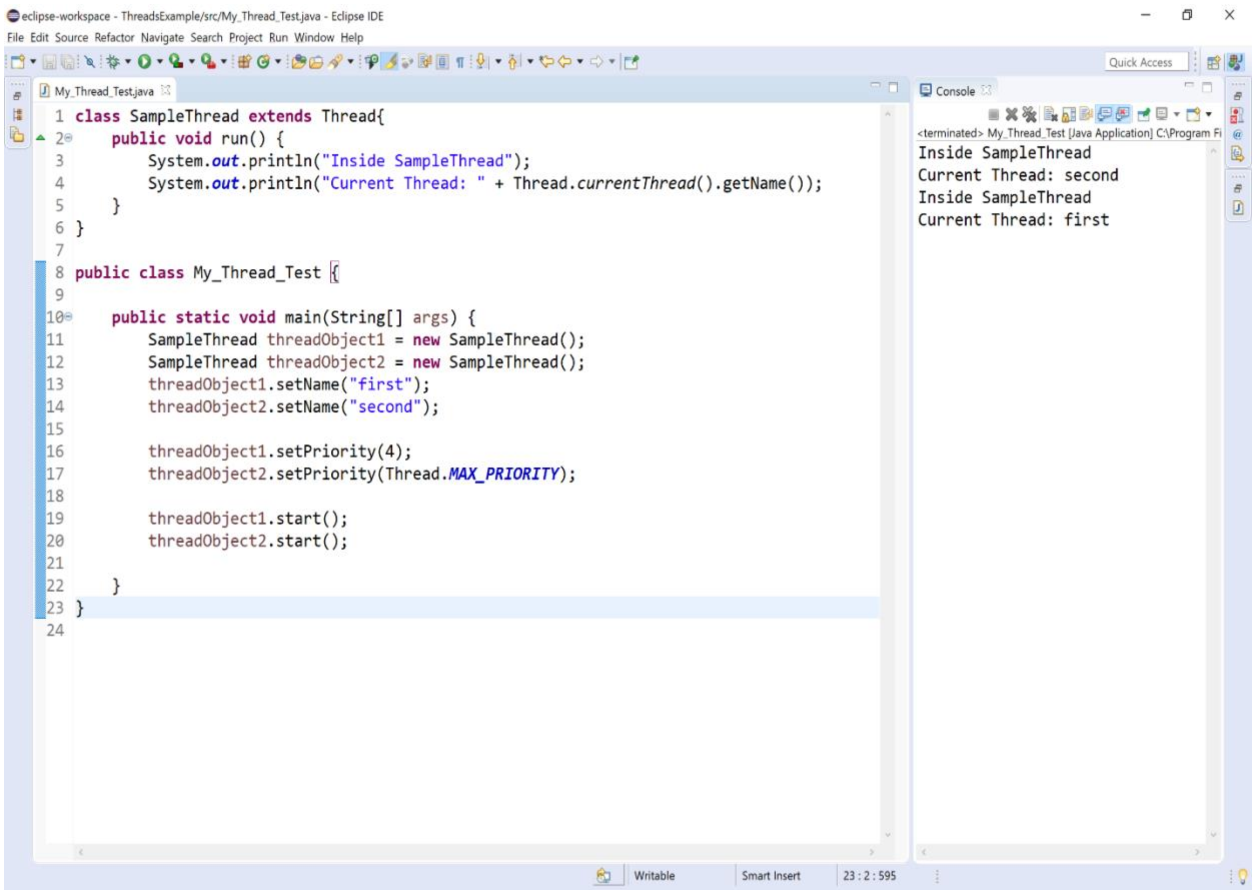
getPriority( ) method

The getPriority( ) method of Thread class used to access the priority of a thread. It does not takes anyargument 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
Quick Access
My_Thread_Test.java
Console
<terminated> My_Thread_Test [Java Application] C:\Program Fi
Writable
Smart Insert
23 : 2 : 595

```java
class SampleThread extends Thread{
    public void run() {
        System.out.println("Inside SampleThread");
        System.out.println("Current Thread: " + Thread.currentThread().getName());
    }
}

public class My_Thread_Test {

    public static void main(String[] args) {
        SampleThread threadObject1 = new SampleThread();
        SampleThread threadObject2 = new SampleThread();
        threadObject1.setName("first");
        threadObject2.setName("second");

        threadObject1.setPriority(4);
        threadObject2.setPriority(Thread.MAX_PRIORITY);

        threadObject1.start();
        threadObject2.start();

    }
}
```

```
Inside SampleThread
Current Thread: second
Inside SampleThread
Current Thread: first
```

Java Thread Synchronisation

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

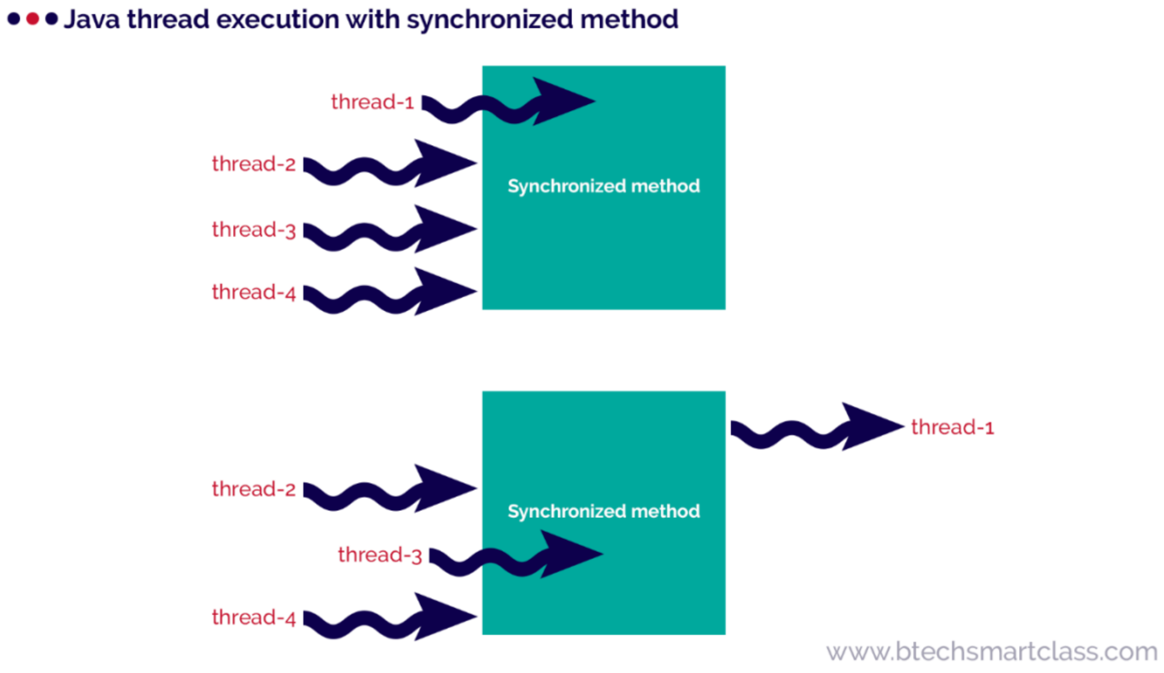
Mutual Exclusion

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

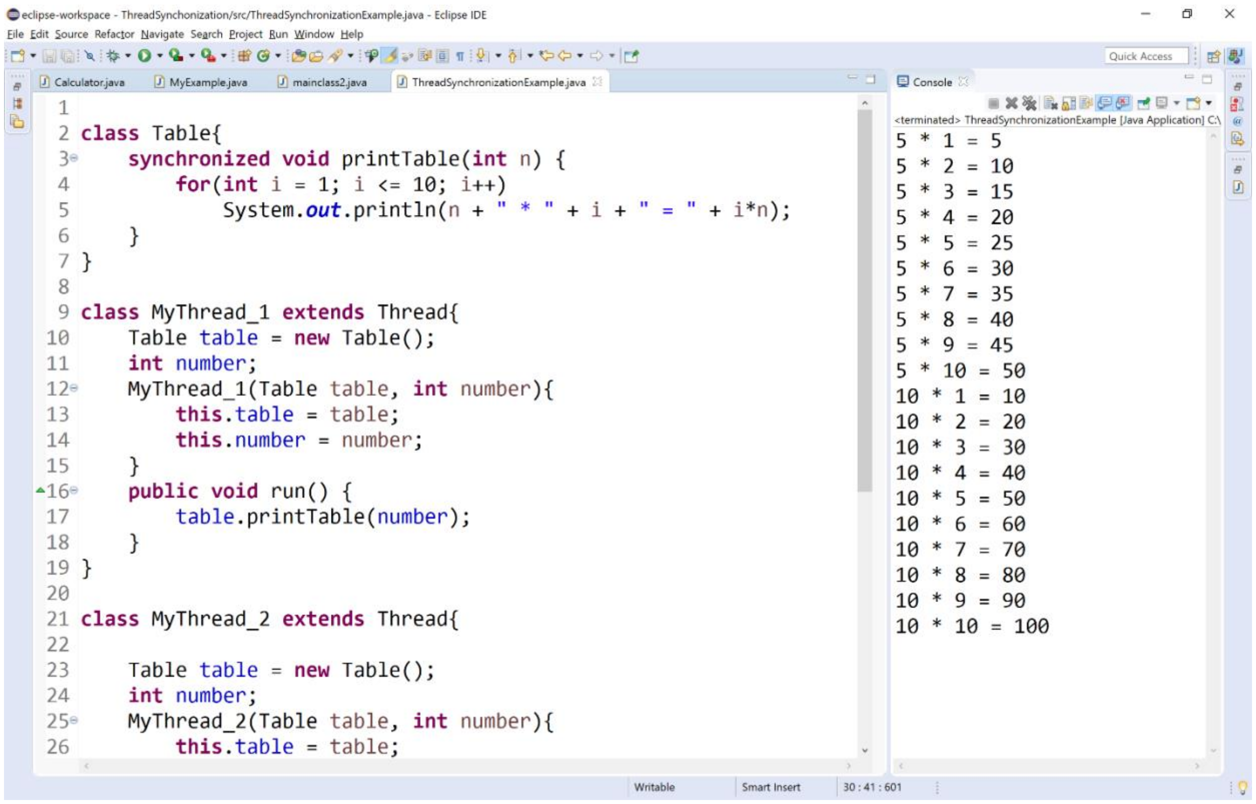
When a method created using a synchronized keyword, it allows only one object to access it at a time. When an object calls a synchronized method, it put a lock on that method so that other objects or thread that are trying to call the same method must wait, until the lock is released. Once the lock is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



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 it task, then one of the threads that are waiting is allocated with the synchronized method, in the above it is thread-3.

Example

```
class Table{
        synchronized void printTable(int n) {
                for(int i = 1; i <= 10; i++)
                        System.out.println(n + " * " + i + " = " + i*n);
        }
}

class MyThread_1 extends Thread{
        Table table = new Table();
        int number;
        MyThread_1(Table table, int number){
                this.table = table;
                this.number = number;
        }
        public void run() {
                table.printTable(number);
        }
}

class MyThread_2 extends Thread{

        Table table = new Table();
        int number;
        MyThread_2(Table table, int number){
                this.table = table;
                this.number = number;
        }
        public void run() {
                table.printTable(number);
        }
}
public class ThreadSynchronizationExample {
        public static void main(String[] args) {
                Table table = new Table();
                MyThread_1 thread_1 = new MyThread_1(table, 5);
                MyThread_2 thread_2 = new MyThread_2(table, 10);
                thread_1.start();
                thread_2.start();
        }
}
```



```
class Table{
    synchronized void printTable(int n) {
        for(int i = 1; i <= 10; i++)
            System.out.println(n + " * " + i + " = " + i*n);
    }
}

class MyThread_1 extends Thread{
    Table table = new Table();
    int number;
    MyThread_1(Table table, int number){
        this.table = table;
        this.number = number;
    }
    public void run() {
        table.printTable(number);
    }
}

class MyThread_2 extends Thread{

    Table table = new Table();
    int number;
    MyThread_2(Table table, int number){
        this.table = table;
```

```
<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
```

Java 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 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.

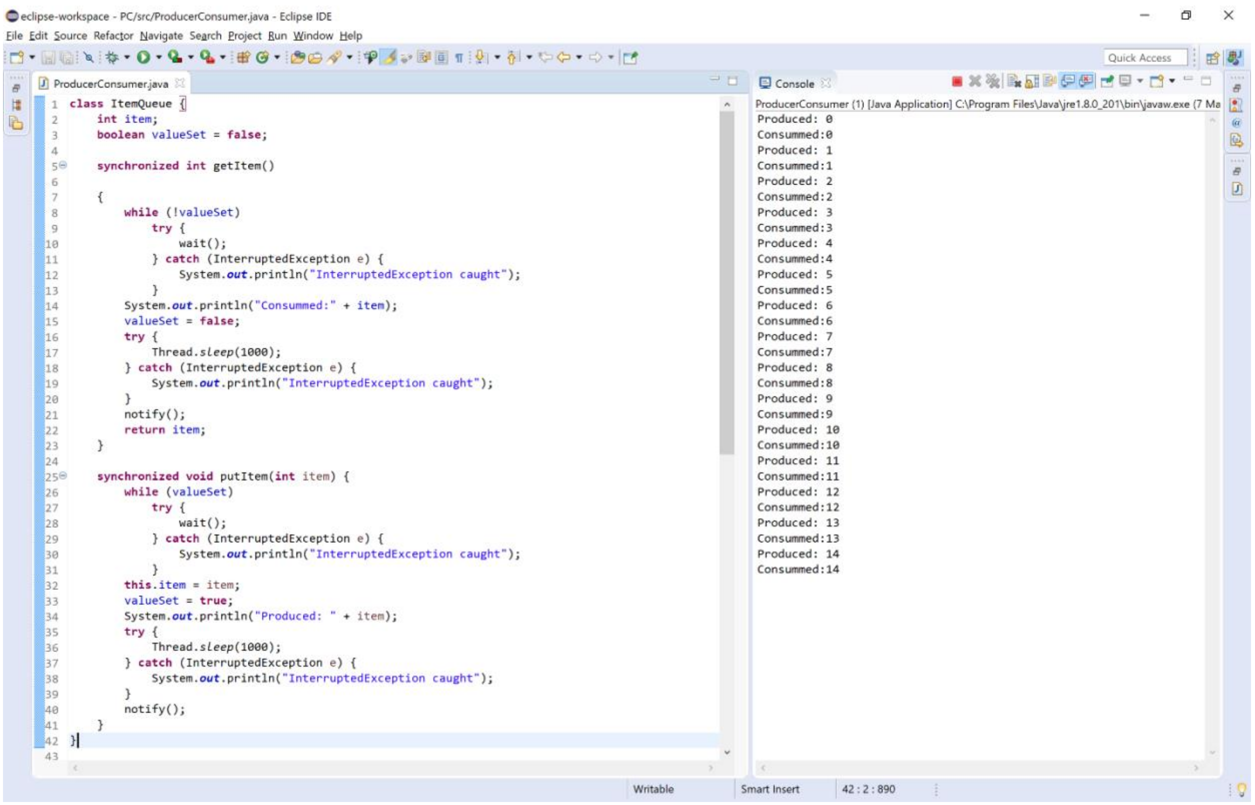
Let's look at an example problem of producer and consumer. The producer produces the item and the consumer consumes the same. But here, the consumer can not consume until the producer produces the item, and producer can not produce until the consumer consumes the item that already been produced. So here, 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.

```
class ItemQueue {
        int item;
        boolean valueSet = false;

        synchronized int getItem()

        {
                while (!valueSet)
                        try {
                                wait();
                        } catch (InterruptedException e) {
```

```
                    System.out.println("InterruptedException caught");
                }
            System.out.println("Consummed:" + item);
            valueSet = false;
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                System.out.println("InterruptedException caught");
            }
            notify();
            return item;
        }

        synchronized void putItem(int item) {
            while (valueSet)
                try {
                    wait();
                } catch (InterruptedException e) {
                    System.out.println("InterruptedException caught");
                }
            this.item = item;
            valueSet = true;
            System.out.println("Produced: " + item);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                System.out.println("InterruptedException caught");
            }
            notify();
        }
    }
```

```
class Producer implements Runnable{
        ItemQueue itemQueue;
        Producer(ItemQueue itemQueue){
                this.itemQueue = itemQueue;
                new Thread(this, "Producer").start();
        }
        public void run() {
                int i = 0;
                while(true) {
                        itemQueue.putItem(i++);
                }
        }
}
class Consumer implements Runnable{

        ItemQueue itemQueue;
        Consumer(ItemQueue itemQueue){
                this.itemQueue = itemQueue;
                new Thread(this, "Consumer").start();
        }
        public void run() {
                while(true) {
                        itemQueue.getItem();
                }
        }
}

class ProducerConsumer{
        public static void main(String args[]) {
                ItemQueue itemQueue = new ItemQueue();
                new Producer(itemQueue);
                new Consumer(itemQueue);
```

```
            }
}
```


eclipse-workspace - PC/src/ProducerConsumer.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
ProducerConsumer.java
Console
Quick Access
Writable
Smart Insert
42 : 2 : 890

```java
class ItemQueue {
    int item;
    boolean valueSet = false;

    synchronized int getItem()

    {
        while (!valueSet)
            try {
                wait();
            } catch (InterruptedException e) {
                System.out.println("InterruptedException caught");
            }
        System.out.println("Consummed:" + item);
        valueSet = false;
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            System.out.println("InterruptedException caught");
        }
        notify();
        return item;
    }

    synchronized void putItem(int item) {
        while (valueSet)
            try {
                wait();
            } catch (InterruptedException e) {
                System.out.println("InterruptedException caught");
            }
        this.item = item;
        valueSet = true;
        System.out.println("Produced: " + item);
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            System.out.println("InterruptedException caught");
        }
        notify();
    }
}
```

```
ProducerConsumer (1) [Java Application] C:\Program Files\Java\jre1.8.0_201\bin\javaw.exe (7 Ma
Produced: 0
Consummed:0
Produced: 1
Consummed:1
Produced: 2
Consummed:2
Produced: 3
Consummed:3
Produced: 4
Consummed:4
Produced: 5
Consummed:5
Produced: 6
Consummed:6
Produced: 7
Consummed:7
Produced: 8
Consummed:8
Produced: 9
Consummed:9
Produced: 10
Consummed:10
Produced: 11
Consummed:11
Produced: 12
Consummed:12
Produced: 13
Consummed:13
Produced: 14
Consummed:14
```

Multithreading in java

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 wieght 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 r 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
Inter process communication is expensive.	Inter thread communication is not ex

Java Wrapper Classes

A Wrapper class in Java is a class that wraps around a primitive data type and converts it into an object

Primitive Data Type	Wrapper Class

byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean
char	Character

Sometimes you must use wrapper classes, for example when working with Collection objects, such as `ArrayList`, where primitive types cannot be used (the list can only store objects):

Example

```
ArrayList<int> myNumbers = new ArrayList<int>(); // Invalid

ArrayList<Integer> myNumbers = new ArrayList<Integer>(); // Valid
```

Autoboxing

Autoboxing is the automatic conversion that the Java compiler makes between the primitive types and their corresponding object wrapper classes. For example, converting an int to an Integer, a double to a Double, and so on.