
M. Sc. ST 1st SEMESTER

UNIT – IV

Exception Handling:

- An exception is an abnormal condition that arises in a code sequence at run time. In other words, an exception is a run-time error.
- In computer languages that do not support exception handling, errors must be checked and handled manually—typically through the use of error codes, and so on. Which is as cumbersome as it is troublesome.
- Java's exception handling avoids these problems and, in the process, brings run-time error management into the object-oriented world.

Exception-Handling Fundamentals:

- A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code.
- When an exceptional condition arises, an object representing that exception is created and thrown in the method that caused the error.
- That method may choose to handle the exception itself, or pass it on. Either way, at some point, the exception is caught and processed.
- Exceptions can be generated by the Java run-time system, or they can be manually generated by your code.
- Exceptions thrown by Java relate to fundamental errors that violate the rules of the Java language or the constraints of the Java execution environment.
- Manually generated exceptions are typically used to report some error condition to the caller of a method.
- Java exception handling is managed via five keywords: **try**, **catch**, **throw**, **throws**, and **finally**. Briefly, here is how they work.
 - Program statements that you want to monitor for exceptions are contained within a try block.
 - If an exception occurs within the try block, it is thrown.
 - Your code can catch this exception (using catch) and handle it in some rational manner.
 - System-generated exceptions are automatically thrown by the Java run-time system.
 - To manually throw an exception, use the keyword throw.
 - Any exception that is thrown out of a method must be specified as such by a throws clause.
 - Any code that absolutely must be executed after a try block completes is put in a finally block.
- The general form of an exception-handling block is shown in following diagram.....
 - Here, ExceptionType is the type of exception that has occurred.

```

try {
    // block of code to monitor for errors
}
catch (ExceptionType1 exOb) {
    // exception handler for ExceptionType1
}

catch (ExceptionType2 exOb) {
    // exception handler for ExceptionType2
}
// ...
finally {
    // block of code to be executed after try block ends
}

```

Exception Types:

- All exception types are subclasses of the built-in class Throwable. Thus, Throwable is at the top of the exception class hierarchy.
- Immediately below Throwable are two subclasses that partition exceptions into two distinct branches.
- One branch is headed by Exception.
 - This class is used for exceptional conditions that user programs should catch. This is also the class that you will subclass to create your own custom exception types.
 - There is an important subclass of Exception, called RuntimeException.
- Exceptions of this type are automatically defined for the programs that you write and include things such as division by zero and invalid array indexing.
- The other branch is topped by Error, which defines exceptions that are not expected to be caught under normal circumstances by your program.
 - Exceptions of type Error are used by the Java run-time system to indicate errors having to do with the run-time environment, itself. Stack overflow is an example of such an error.

Uncaught Exceptions:

- Before you learn how to handle exceptions in your program, it is useful to see what happens when you don't handle them.
- This small program includes an expression that intentionally causes a divide-by-zero error...

```

class Exc0 {
    public static void main(String args[]) {
        int d = 0;
        int a = 42 / d;
    }
}

```

- When the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and then throws this exception.

- This causes the execution of Exc0 to stop, because once an exception has been thrown, it must be caught by an exception handler and dealt with immediately.
- In this example, we haven't supplied any exception handlers of our own, so the exception is caught by the default handler provided by the Java run-time system.
- Any exception that is not caught by your program will ultimately be processed by the default handler.
- The default handler displays a string describing the exception, prints a stack trace from the point at which the exception occurred, and terminates the program.
- Here is the exception generated when this example is executed.

```
java.lang.ArithmeticException: / by zero
    at Exc0.main(Exc0.java:4)
```

- Notice how the class name, Exc0; the method name, main; the filename, Exc0.java; and the line number, 4, are all included in the simple stack trace.
- Also, notice that the type of exception thrown is a subclass of Exception called ArithmeticException, which more specifically describes what type of error happened.
- Java supplies several built-in exception types that match the various sorts of run-time errors that can be generated.
- The stack trace will always show the sequence of method invocations that led up to the error.
- For e.g., here is another version of the preceding program that introduces the same error but in a method separate from main().

```
class Exc1 {
    static void subroutine() {
        int d = 0;
        int a = 10 / d;
    }
    public static void main(String args[]) {
        Exc1.subroutine();
    }
}
```

```
java.lang.ArithmeticException: / by zero
    at Exc1.subroutine(Exc1.java:4)
    at Exc1.main(Exc1.java:7)
```

Using try and catch:

- Although the default exception handler provided by the Java run-time system is useful for debugging, we will usually want to handle an exception yourself. Doing so provides two benefits.
 - First, it allows you to fix the error.
 - Second, it prevents the program from automatically terminating.
- Most users would be confused (to say the least) if your program stopped running and printed a stack trace whenever an error occurred!
- Fortunately, it is quite easy to prevent this.
 - To guard against and handle a run-time error, simply enclose the code that you want to monitor inside a try block.
 - Immediately following the try block, include a catch clause that specifies the exception type that you wish to catch.
- To illustrate how easily this can be done.....
- The following program includes a try block and a catch clause that processes the `ArithmeticException` generated by the division-by-zero error.
 - Notice that the call to `println()` inside the try block is never executed. Once an exception is thrown, program control transfers out of the try block into the catch block.
 - Put differently, catch is not “called,” so execution never “returns” to the try block from a catch. Thus, the line “This will not be printed.” is not displayed.

```
class Exc2 {
    public static void main(String args[]) {
        int d, a;

        try { // monitor a block of code.
            d = 0;
            a = 42 / d;
            System.out.println("This will not be printed.");
        } catch (ArithmeticException e) { // catch divide-by-zero error
            System.out.println("Division by zero.");
        }
        System.out.println("After catch statement.");
    }
}
```

This program generates the following output:

```
Division by zero.
After catch statement.
```

- Once the catch statement has executed, program control continues with the next line in the program following the entire try/catch mechanism.

- A try and its catch statement form a unit. The scope of the catch clause is restricted to those statements specified by the immediately preceding try statement.
- A catch statement cannot catch an exception thrown by another try statement (except in the case of nested try statements, described shortly).
- The statements that are protected by try must be surrounded by curly braces (within a block.) You cannot use try on a single statement.
- The goal of most well-constructed catch clauses should be to resolve the exceptional condition and then continue on as if the error had never happened.
- For example...
 - In the next program each iteration of the for loop obtains two random integers.
 - Those two integers are divided by each other, and the result is used to divide the value 12345.
 - The final result is put into a.
 - If either division operation causes a divide-by-zero error, it is caught, the value of a is set to zero, and the program continues.

```
// Handle an exception and move on.
import java.util.Random;

class HandleError {
    public static void main(String args[]) {
        int a=0, b=0, c=0;
        Random r = new Random();

        for(int i=0; i<32000; i++) {
            try {
                b = r.nextInt();
                c = r.nextInt();
                a = 12345 / (b/c);
            } catch (ArithmeticException e) {
                System.out.println("Division by zero.");
                a = 0; // set a to zero and continue
            }
            System.out.println("a: " + a);
        }
    }
}
```

Displaying a Description of an Exception:

- Throwable overrides the toString() method (defined by Object) so that it returns a string containing a description of the exception.
- You can display this description in a println() statement by simply passing the exception as an argument.
- For example, the catch block in the preceding program can be rewritten like this...

```

catch (ArithmeticException e) {
    System.out.println("Exception: " + e);
    a = 0; // set a to zero and continue
}

```

- When this version is substituted in the program, and the program is run, each divide-by-zero error displays the message.
- While it is of no particular value in this context, the ability to display a description of an exception is valuable in other circumstances—particularly when you are experimenting with exceptions or when you are debugging.

Multiple catch Clauses:

- 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.
- The following example traps two different exception types.

```

// Demonstrate multiple catch statements.
class MultiCatch {
    public static void main(String args[]) {
        try {
            int a = args.length;
            System.out.println("a = " + a);
            int b = 42 / a;
            int c[] = { 1 };
            c[42] = 99;
        } catch (ArithmeticException e) {
            System.out.println("Divide by 0: " + e);
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Array index oob: " + e);
        }
        System.out.println("After try/catch blocks.");
    }
}

C:\>java MultiCatch
a = 0
Divide by 0: java.lang.ArithmeticException: / by zero
After try/catch blocks.

C:\>java MultiCatch TestArg
a = 1
Array index oob: java.lang.ArrayIndexOutOfBoundsException: 42
After try/catch blocks.

```


-
- When you use multiple catch statements, it is important to remember that exception subclasses must come before any of their superclasses.
 - This is because a catch statement that uses a superclass will catch exceptions of that type plus any of its subclasses.
 - Thus, a subclass would never be reached if it came after its superclass. Further, in Java, unreachable code is an error. For example.

```
class SuperSubCatch {  
    public static void main(String args[]) {  
        try {  
            int a = 0;  
            int b = 42 / a;  
        } catch(Exception e) {  
            System.out.println("Generic Exception catch.");  
        }  
        catch(ArithmeticException e) { // ERROR - unreachable  
            System.out.println("This is never reached.");  
        }  
    }  
}
```

Nested try Statements:

- The try statement can be nested. That is, a try statement can be inside the block of another try.
 - Each time a try statement is entered, the context of that exception is pushed on the stack.
 - If an inner try statement does not have a catch handler for a particular exception, the stack is unwound and the next try statement's catch handlers are inspected for a match.
 - This continues until one of the catch statements succeeds, or until all of the nested try statements are exhausted.
 - If no catch statement matches, then the Java run-time system will handle the exception.
- The following is an example that uses nested try statements. As you can see, this program nests one try block within another. The program works as follows...
 - When you execute the program with no command-line arguments, a divide-by-zero exception is generated by the outer try block.
 - Execution of the program with one command-line argument generates a divide-by-zero exception from within the nested try block.
 - Since the inner block does not catch this exception, it is passed on to the outer try block, where it is handled.
 - If you execute the program with two command-line arguments, an array boundary exception is generated from within the inner try block.

```
// An example of nested try statements.
class NestTry {
    public static void main(String args[]) {
        try {
            int a = args.length;

            /* If no command-line args are present,
             the following statement will generate
             a divide-by-zero exception. */

            int b = 42 / a;
            System.out.println("a = " + a);

            try { // nested try block
                /* If one command-line arg is used,
                 then a divide-by-zero exception
                 will be generated by the following code. */
                if(a==1) a = a/(a-a); // division by zero

                /* If two command-line args are used,
                 then generate an out-of-bounds exception. */
                if(a==2) {
                    int c[] = { 1 };
                    c[42] = 99; // generate an out-of-bounds exception
                }
            } catch(ArrayIndexOutOfBoundsException e) {
                System.out.println("Array index out-of-bounds: " + e);
            }

        } catch(ArithmeticException e) {
            System.out.println("Divide by 0: " + e);
        }
    }
}
```

```
C:\>java NestTry
Divide by 0: java.lang.ArithmeticException: / by zero

C:\>java NestTry One
a = 1
Divide by 0: java.lang.ArithmeticException: / by zero

C:\>java NestTry One Two
a = 2
Array index out-of-bounds:
java.lang.ArrayIndexOutOfBoundsException:42
```

Throw statement:

- So far, you have only been catching exceptions that are thrown by the Java run-time system.
- However, it is possible for your program to throw an exception explicitly, using the throw statement. The general form of throw is shown here.....

throw ThrowableInstance;

- Here, ThrowableInstance must be an object of type Throwable or a subclass of Throwable.

- Primitive types, such as int or char, as well as non-Throwable classes, such as String and Object, cannot be used as exceptions.
- There are two ways you can obtain a Throwable object...
 - Using a parameter in a catch clause, or creating one with the new operator.
- The flow of execution stops immediately after the throw statement; any subsequent statements are not executed.
- The nearest enclosing try block is inspected to see if it has a catch statement that matches the type of exception.
 - If it does find a match, control is transferred to that statement.
 - If not, then the next enclosing try statement is inspected, and so on.
 - If no matching catch is found, then the default exception handler halts the program and prints the stack trace.
- Here is a sample program that creates and throws an exception. The handler that catches the exception rethrows it to the outer handler.

```
// Demonstrate throw.
class ThrowDemo {
    static void demoproc() {
        try {
            throw new NullPointerException("demo");
        } catch(NullPointerException e) {
            System.out.println("Caught inside demoproc.");
            throw e; // rethrow the exception
        }
    }

    public static void main(String args[]) {
        try {
            demoproc();
        } catch(NullPointerException e) {
            System.out.println("Recaught: " + e);
        }
    }
}
```

- This program gets two chances to deal with the same error.
 - First, main() sets up an exception context and then calls demoproc().
 - The demoproc() method then sets up another exception handling context and immediately throws a new instance of NullPointerException, which is caught on the next line.
 - The exception is then rethrown. Here is the resulting output:

```
Caught inside demoproc.
Recaught: java.lang.NullPointerException: demo
```

-
- The program also illustrates how to create one of Java's standard exception objects. Pay close attention to this line:

throw new NullPointerException("demo");

- Here, new is used to construct an instance of NullPointerException.
- Many of Java's built-in run-time exceptions have at least two constructors:
 - One with no parameter and one that takes a string parameter.
- When the second form is used, the argument specifies a string that describes the exception. This string is displayed when the object is used as an argument to print() or println().
- It can also be obtained by a call to getMessage(), which is defined by Throwable.

Throws statement:

- If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception.
- You do this by including a throws clause in the method's declaration. A throws clause lists the types of exceptions that a method might throw.
- This is necessary for all exceptions, except those of type Error or RuntimeException, or any of their subclasses.
- All other exceptions that a method can throw must be declared in the throws clause. If they are not, a compile-time error will result.
- This is the general form of a method declaration that includes a throws clause.....

```
type method-name(parameter-list) throws exception-list
{
    // body of method
}
```

- Here, exception-list is a comma-separated list of the exceptions that a method can throw.
- Following is an example of an incorrect program that tries to throw an exception that it does not catch.

```
// This program contains an error and will not compile.
class ThrowsDemo {
    static void throwOne() {
        System.out.println("Inside throwOne.");
        throw new IllegalAccessException("demo");
    }
    public static void main(String args[]) {
        throwOne();
    }
}
```

-
- Because the program does not specify a throws clause to declare this fact, the program will not compile
 - To make this example compile, you need to make two changes...
 - First, you need to declare that throwOne() throws IllegalAccessException.
 - Second, main() must define a try/catch statement that catches this exception..
 - The corrected example is shown here.....

```
// This is now correct.
class ThrowsDemo {
    static void throwOne() throws IllegalAccessException {
        System.out.println("Inside throwOne.");
        throw new IllegalAccessException("demo");
    }
    public static void main(String args[]) {
        try {
            throwOne();
        } catch (IllegalAccessException e) {
            System.out.println("Caught " + e);
        }
    }
}
```

Here is the output generated by running this example program:

```
inside throwOne
caught java.lang.IllegalAccessException: demo
```

Finally:

- When exceptions are thrown, execution in a method takes a rather abrupt, nonlinear path that alters the normal flow through the method.
- Depending upon how the method is coded, it is even possible for an exception to cause the method to return prematurely. This could be a problem in some methods.
 - For example, if a method opens a file upon entry and closes it upon exit, then you will not want the code that closes the file to be bypassed by the exception-handling mechanism.
- The finally keyword is designed to address this contingency.
- finally creates a block of code that will be executed after a try/catch block has completed and before the code following the try/catch block.
- The finally block will execute whether or not an exception is thrown.
 - If an exception is thrown, the finally block will execute even if no catch statement matches the exception.
 - Any time a method is about to return to the caller from inside a try/catch block, via an uncaught exception or an explicit return statement, the finally clause is also executed just before the method returns.

- This can be useful for closing file handles and freeing up any other resources that might have been allocated at the beginning of a method with the intent of disposing of them before returning.
- The finally clause is optional. However, each try statement requires at least one catch or a finally clause.
- Here is an example program that shows three methods that exit in various ways, none without executing their finally clauses.
 - In this example, procA() prematurely breaks out of the try by throwing an exception. The finally clause is executed on the way out.
 - procB()'s try statement is exited via a return statement. The finally clause is executed before procB() returns.
 - In procC(), the try statement executes normally, without error. However, the finally block is still executed.

<pre>// Demonstrate finally. class FinallyDemo { // Through an exception out of the method. static void procA() { try { System.out.println("inside procA"); throw new RuntimeException("demo"); } finally { System.out.println("procA's finally"); } } // Return from within a try block. static void procB() { try { System.out.println("inside procB"); return; } finally { System.out.println("procB's finally"); } } }</pre>	<pre>// Execute a try block normally. static void procC() { try { System.out.println("inside procC"); } finally { System.out.println("procC's finally"); } } public static void main(String args[]) { try { procA(); } catch (Exception e) { System.out.println("Exception caught"); } procB(); procC(); }</pre>
---	---

Here is the output generated by the preceding program:

```
inside procA
procA's finally
Exception caught
inside procB
procB's finally
inside procC
procC's finally
```

Java's Built-in Exceptions:

- Inside the standard package `java.lang`, Java defines several exception classes. A few have been used by the preceding examples.
- The most general of these exceptions are subclasses of the standard type `RuntimeException`.
- As previously explained, these exceptions need not be included in any method's throws list.
- In the language of Java, these are called unchecked exceptions because the compiler does not check to see if a method handles or throws these exceptions.
- The unchecked exceptions defined in `java.lang` are listed in Table.

Exception	Meaning
<code>ArithmeticException</code>	Arithmetic error, such as divide-by-zero.
<code>ArrayIndexOutOfBoundsException</code>	Array index is out-of-bounds.
<code>ArrayStoreException</code>	Assignment to an array element of an incompatible type.
<code>ClassCastException</code>	Invalid cast.
<code>EnumConstantNotPresentException</code>	An attempt is made to use an undefined enumeration value.
<code>IllegalArgumentException</code>	Illegal argument used to invoke a method.
<code>IllegalMonitorStateException</code>	Illegal monitor operation, such as waiting on an unlocked thread.
<code>IllegalStateException</code>	Environment or application is in incorrect state.
<code>IllegalThreadStateException</code>	Requested operation not compatible with current thread state.
<code>IndexOutOfBoundsException</code>	Some type of index is out-of-bounds.
<code>NegativeArraySizeException</code>	Array created with a negative size.
<code>NullPointerException</code>	Invalid use of a null reference.
<code>NumberFormatException</code>	Invalid conversion of a string to a numeric format.
<code>SecurityException</code>	Attempt to violate security.
<code>StringIndexOutOfBoundsException</code>	Attempt to index outside the bounds of a string.
<code>TypeNotPresentException</code>	Type not found.
<code>UnsupportedOperationException</code>	An unsupported operation was encountered.

TABLE 10-1 Java's Unchecked `RuntimeException` Subclasses Defined in `java.lang`

- Table 10-2 lists those exceptions defined by `java.lang` that must be included in a method's throws list if that method can generate one of these exceptions and not handle it itself.

Exception	Meaning
<code>ClassNotFoundException</code>	Class not found.
<code>CloneNotSupportedException</code>	Attempt to clone an object that does not implement the <code>Cloneable</code> interface.
<code>IllegalAccessException</code>	Access to a class is denied.
<code>InstantiationException</code>	Attempt to create an object of an abstract class or interface.
<code>InterruptedException</code>	One thread has been interrupted by another thread.
<code>NoSuchFieldException</code>	A requested field does not exist.
<code>NoSuchMethodException</code>	A requested method does not exist.

TABLE 10-2 Java's Checked Exceptions Defined in `java.lang`

Creating Your Own Exception Subclasses:

- Although Java's built-in exceptions handle most common errors, you will probably want to create your own exception types to handle situations specific to your applications.
- This is quite easy to do: just define a subclass of `Exception` (which is, of course, a subclass of `Throwable`).
- Your subclasses don't need to actually implement anything—it is their existence in the type system that allows you to use them as exceptions.
- The `Exception` class does not define any methods of its own. It does, of course, inherit those methods provided by `Throwable`.
- Thus, all exceptions, including those that you create, have the methods defined by `Throwable` available to them. They are shown in Table 10-3.

Method	Description
<code>Throwable fillInStackTrace()</code>	Returns a Throwable object that contains a completed stack trace. This object can be rethrown.
<code>Throwable getCause()</code>	Returns the exception that underlies the current exception. If there is no underlying exception, null is returned.
<code>String getLocalizedMessage()</code>	Returns a localized description of the exception.
<code>String getMessage()</code>	Returns a description of the exception.
<code>StackTraceElement[] getStackTrace()</code>	Returns an array that contains the stack trace, one element at a time, as an array of StackTraceElement . The method at the top of the stack is the last method called before the exception was thrown. This method is found in the first element of the array. The StackTraceElement class gives your program access to information about each element in the trace, such as its method name.
<code>Throwable initCause(Throwable causeExc)</code>	Associates <i>causeExc</i> with the invoking exception as a cause of the invoking exception. Returns a reference to the exception.
<code>void printStackTrace()</code>	Displays the stack trace.
<code>void printStackTrace(PrintStream stream)</code>	Sends the stack trace to the specified stream.
<code>void printStackTrace(PrintWriter stream)</code>	Sends the stack trace to the specified stream.
<code>void setStackTrace(StackTraceElement elements[])</code>	Sets the stack trace to the elements passed in <i>elements</i> . This method is for specialized applications, not normal use.
<code>String toString()</code>	Returns a String object containing a description of the exception. This method is called by <code>println()</code> when outputting a Throwable object.

TABLE 10-3 The Methods Defined by `Throwable`

- You may also wish to override one or more of these methods in exception classes that you create.
- `Exception` defines four constructors. Two were added by JDK 1.4 to support chained exceptions, described in the next section. The other two are shown here...

Exception();

Exception(String msg);

- The first form creates an exception that has no description.
- The second form lets you specify a description of the exception.
- Although specifying a description when an exception is created is often useful, sometimes it is better to override `toString()`. Here's why:
 - The version of `toString()` defined by `Throwable` (and inherited by `Exception`) first displays the name of the exception followed by a colon, which is then followed by your description.
 - By overriding `toString()`, you can prevent the exception name and colon from being displayed. This makes for a cleaner output, which is desirable in some cases.
- The following example declares a new subclass of `Exception` and then uses that subclass to signal an error condition in a method.
 - It overrides the `toString()` method, allowing a carefully tailored description of the exception to be displayed.

```
// This program creates a custom exception type.
class MyException extends Exception {
    private int detail;

    MyException(int a) {
        detail = a;
    }

    public String toString() {
        return "MyException[" + detail + "]";
    }
}
```

```
class ExceptionDemo {
    static void compute(int a) throws MyException {
        System.out.println("Called compute(" + a + ")");
        if(a > 10)
            throw new MyException(a);
        System.out.println("Normal exit");
    }

    public static void main(String args[]) {
        try {
            compute(1);
            compute(20);
        } catch (MyException e) {
            System.out.println("Caught " + e);
        }
    }
}
```

-
- This example defines a subclass of Exception called MyException.
 - This subclass is quite simple: it has only a constructor plus an overloaded toString() method that displays the value of the exception.
 - The ExceptionDemo class defines a method named compute() that throws a MyException object.
 - The exception is thrown when compute()'s integer parameter is greater than 10.
 - The main() method sets up an exception handler for MyException, then calls compute() with a legal value (less than 10) and an illegal one to show both paths through the code. Here is the result...

```
Called compute(1)
Normal exit
Called compute(20)
Caught MyException[20]
```

An Example of own Exception:

Define a class OwnException, which will be thrown if a given integer is a palindrome or not. Use the same in main method while accepting array elements.

```
import java.util.Scanner;

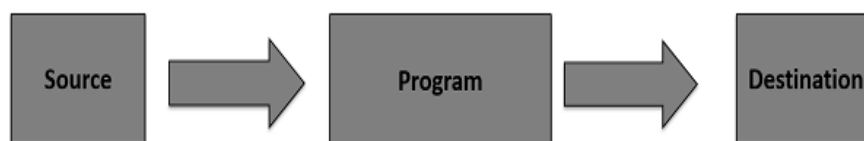
class MyException extends Exception {
    String str;
    MyException (String s) {
        str=s;
    }
    public String toString() {
        return ("MyException: "+str);
    }
}

class OwnExcepPalin {
    static void Ispalin(int num) throws MyException {
        int r, rev=0, t=num;
        while(num > 0 ) {
            r=num%10;
            rev=rev*10+r;
            num=num/10;
        }
        if ( t == rev )
            throw new MyException("Palindrome");
    }
}
```

```
public static void main(String args[]) {
    Scanner sc = new Scanner (System.in);
    int i, ele,ar[],n;
    System.out.println("Enter the Size of the array:");
    n=sc.nextInt();
    ar=new int[n];
    try {
        for(i = 0 ;i < n; i++) {
            ele=sc.nextInt();
            Ispalin(ele);
            ar[i]=ele;
        }
    } catch( Exception e) {
        System.out.println("Caught thrown exception"+e);
    }
    System.out.println("The given Array elements are:");
    for(i=0;i<n;i++)
        System.out.println(ar[i]);
}
```

File Handling:

- The java.io package contains nearly every class you might ever need to perform input and output (I/O) in Java.
- All these streams represent an input source and an output destination.
- The stream in the java.io package supports many data such as primitives, object, localized characters, etc.
- A stream can be defined as a sequence of data. There are two kinds of Streams –
 - **InPutStream** – The InputStream is used to read data from a source.
 - **OutPutStream** – The OutputStream is used for writing data to a destination.



Byte Streams:

- Java byte streams are used to perform input and output of 8-bit bytes.

- Though there are many classes related to byte streams but the most frequently used classes are, `FileInputStream` and `FileOutputStream`.
- Following is an example which makes use of these two classes to copy an input file into an output file...

```
import java.io.*;
public class CopyFile {

    public static void main(String args[]) throws IOException {
        FileInputStream in = null;
        FileOutputStream out = null;

        try {
            in = new FileInputStream("input.txt");
            out = new FileOutputStream("output.txt");

            int c;
            while ((c = in.read()) != -1) {
                out.write(c);
            }
        } finally {
            if (in != null) {
                in.close();
            }
            if (out != null) {
                out.close();
            }
        }
    }
}
```

Character Streams:

- Java Byte streams are used to perform input and output of 8-bit bytes, whereas Java Character streams are used to perform input and output for 16-bit unicode.
- Though there are many classes related to character streams but the most frequently used classes are `FileReader` and `FileWriter`.
- Though internally `FileReader` uses `FileInputStream` and `FileWriter` uses `FileOutputStream` but here the major difference is that `FileReader` reads two bytes at a time and `FileWriter` writes two bytes at a time.
- We can re-write the above example, which makes the use of these two classes to copy an input file (having unicode characters) into an output file.

Standard Streams:

- All the programming languages provide support for standard I/O where the user's program can take input from a keyboard and then produce an output on the computer screen.
- If you are aware of C or C++ programming languages, then you must be aware of three standard devices `STDIN`, `STDOUT` and `STDERR`.


```

import java.io.*;
public class CopyFile {

    public static void main(String args[]) throws IOException {
        FileReader in = null;
        FileWriter out = null;

        try {
            in = new FileReader("input.txt");
            out = new FileWriter("output.txt");

            int c;
            while ((c = in.read()) != -1) {
                out.write(c);
            }
        } finally {
            if (in != null) {
                in.close();
            }
            if (out != null) {
                out.close();
            }
        }
    }
}

```

- Similarly, Java provides the following three standard streams.....
 - **Standard Input** - This is used to feed the data to user's program and usually a keyboard is used as standard input stream and represented as **System.in**.
 - **Standard Output** - This is used to output the data produced by the user's program and usually a computer screen is used for standard output stream and represented as **System.out**.
 - **Standard Error** - This is used to output the error data produced by the user's program and usually a computer screen is used for standard error stream and represented as **System.err**.
- Following is a simple program, which creates InputStreamReader to read standard input stream until the user types a "q".

```

import java.io.*;
public class ReadConsole {

    public static void main(String args[]) throws IOException {
        InputStreamReader cin = null;

        try {
            cin = new InputStreamReader(System.in);
            System.out.println("Enter characters, 'q' to quit.");
            char c;

```

```
do {
    c = (char) cin.read();
    System.out.print(c);
} while(c != 'q');
}finally {
    if (cin != null) {
        cin.close();
    }
}
}
```

Multithreading:

- Unlike many other computer languages, Java provides built-in support for multithreaded programming.
 - A multithreaded program contains two or more parts that can run concurrently.
 - Each part of such a program is called a thread, and each thread defines a separate path of execution. Thus, multithreading is a specialized form of multitasking.
- You are almost certainly acquainted with multitasking, because it is supported by virtually all modern operating systems.
- However, there are two distinct types of multitasking: process-based and thread-based.
- It is important to understand the difference between the two. For most readers, process-based multitasking is the more familiar form.
- A process is, in essence, a program that is executing. Thus, process-based multitasking is the feature that allows your computer to run two or more programs concurrently.
 - For example, process-based multitasking enables you to run the Java compiler at the same time that you are using a text editor.
 - In process-based multitasking, a program is the smallest unit of code that can be dispatched by the scheduler.
- In a thread-based multitasking environment, the thread is the smallest unit of dispatchable code.
 - This means that a single program can perform two or more tasks simultaneously.
 - For instance, a text editor can format text at the same time that it is printing, as long as these two actions are being performed by two separate threads.

-
- Thus, process-based multitasking deals with the “big picture,” and thread-based multitasking handles the details.
 - Multitasking threads require less overhead than multitasking processes.
 - Processes are heavyweight tasks that require their own separate address spaces.
 - Inter-process communication is expensive and limited. Context switching from one process to another is also costly.
 - Threads, on the other hand, are lightweight. They share the same address space and cooperatively share the same heavyweight process.
 - Inter-thread communication is inexpensive, and context switching from one thread to the next is low cost.
 - While Java programs make use of process-based multitasking environments, process-based multitasking is not under the control of Java.
 - Multithreading enables you to write very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum.
 - This is especially important for the interactive, networked environment in which Java operates, because idle time is common. For example.....
 - The transmission rate of data over a network is much slower than the rate at which the computer can process it.
 - Even local file system resources are read and written at a much slower pace than they can be processed by the CPU.
 - And, of course, user input is much slower than the computer.
 - In a single-threaded environment, your program has to wait for each of these tasks to finish before it can proceed to the next one—even though the CPU is sitting idle most of the time.
 - Multithreading lets you gain access to this idle time and put it to good use.
 - However, the fact that Java manages threads makes multithreading especially convenient, because many of the details are handled for you.

The Java Thread Model:

- The Java run-time system depends on threads for many things, and all the class libraries are designed with multithreading in mind.
- In fact, Java uses threads to enable the entire environment to be asynchronous. This helps reduce inefficiency by preventing the waste of CPU cycles.

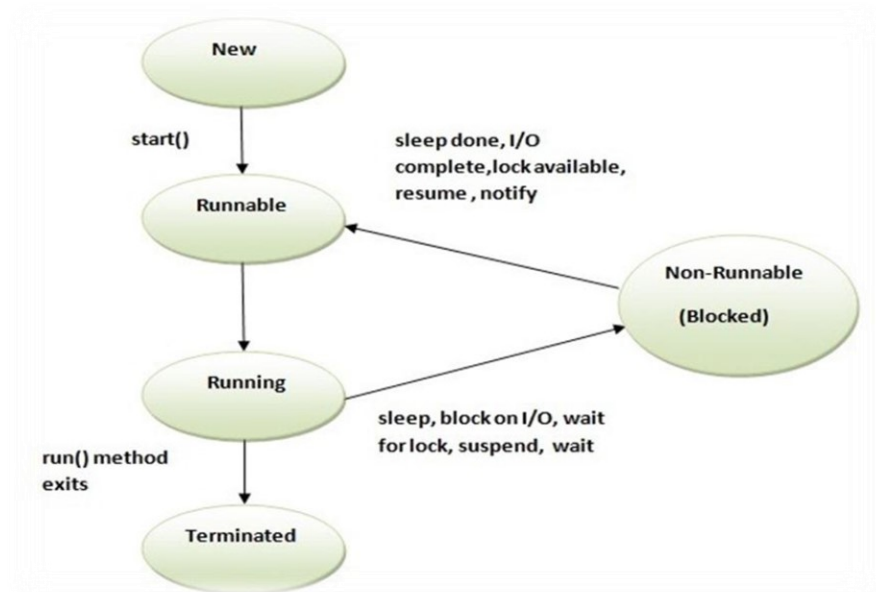
-
- Single-threaded systems use an approach called an event loop with polling.
 - In this model, a single thread of control runs in an infinite loop, polling a single event queue to decide what to do next.
 - Once this polling mechanism returns with, say, a signal that a network file is ready to be read, then the event loop dispatches control to the appropriate event handler.
 - Until this event handler returns, nothing else can happen in the system. This wastes CPU time.
 - It can also result in one part of a program dominating the system and preventing any other events from being processed.
 - In general, in a singled-threaded environment, when a thread blocks (that is, suspends execution) because it is waiting for some resource, the entire program stops running.
 - The benefit of Java's multithreading is that the main loop/polling mechanism is eliminated.
 - One thread can pause without stopping other parts of your program.
 - For example, the idle time created when a thread reads data from a network or waits for user input can be utilized elsewhere.
 - Multithreading allows animation loops to sleep for a second between each frame without causing the whole system to pause.
 - When a thread blocks in a Java program, only the single thread that is blocked pauses. All other threads continue to run.

Life cycle of a Thread (Thread States):

- A thread can be in one of the five states. The life cycle of the thread in java is controlled by JVM. The java thread states are as follows..... (refer following diagram)
 - New, Runnable(ready), Running, Non-Runnable(Blocked) and Terminated.
 - A thread can be **running**.
 - It can be **ready** to run as soon as it gets CPU time.
 - A running thread can be **suspended**, which temporarily suspends its activity.
 - A suspended thread can then be **resumed**, allowing it to pick up where it left off.
 - A thread can be **blocked** when waiting for a resource.
 - At any time, a thread can be **terminated**, which halts its execution immediately. Once terminated, a thread cannot be resumed.

The Thread Class and the Runnable Interface:

- Java's multithreading system is built upon the **Thread** class, its methods, and its companion interface, **Runnable**.
- **Thread** encapsulates a thread of execution. Since you can't directly refer to the ethereal state of a running thread, you will deal with it through its proxy, the **Thread** instance that spawned it.



- To create a new thread, your program will either extend **Thread** or implement the **Runnable** interface.
- The **Thread** class defines several methods that help manage threads, such as.....

Method	Meaning
<code>getName</code>	Obtain a thread's name.
<code>getPriority</code>	Obtain a thread's priority.
<code>isAlive</code>	Determine if a thread is still running.
<code>join</code>	Wait for a thread to terminate.
<code>run</code>	Entry point for the thread.
<code>sleep</code>	Suspend a thread for a period of time.
<code>start</code>	Start a thread by calling its run method.

The Main Thread:

- When a Java program starts up, one thread begins running immediately.
 - This is usually called the main thread of your program, because it is the one that is executed when your program begins.
- The main thread is important for two reasons.....

-
- It is the thread from which other “child” threads will be spawned.
 - Often, it must be the last thread to finish execution because it performs various shutdown actions.
 - Although the main thread is created automatically when your program is started, it can be controlled through a **Thread** object.
 - To do so, you must obtain a reference to it by calling the method `currentThread()`, which is a **public static** member of **Thread**.
 - Its general form is shown here: **static Thread currentThread()**;
 - This method returns a reference to the thread in which it is called.
 - Once you have a reference to the main thread, you can control it just like any other thread. Let’s begin by reviewing the following example...

```
// Controlling the main Thread.
class CurrentThreadDemo {
    public static void main(String args[]) {
        Thread t = Thread.currentThread();

        System.out.println("Current thread: " + t);

        // change the name of the thread
        t.setName("My Thread");
        System.out.println("After name change: " + t);

        try {
            for(int n = 5; n > 0; n--) {
                System.out.println(n);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println("Main thread interrupted");
        }
    }
}
```

- In this program, a reference to the current thread (the main thread, in this case) is obtained by calling **currentThread()**, and this reference is stored in the local variable **t**.
- Next, the program displays information about the thread. The program then calls `setName()` to change the internal name of the thread. Information about the thread is then redisplayed.
- Next, a loop counts down from five, pausing one second between each line. The pause is accomplished by the `sleep()` method. The argument to `sleep()` specifies the delay period in milliseconds.

-
- Notice the try/catch block around this loop. The sleep() method in Thread might throw an InterruptedException. This would happen if some other thread wanted to interrupt this sleeping one.
 - This example just prints a message if it gets interrupted. In a real program, you would need to handle this differently. Here is the output generated by this program.
 - Notice the output produced when t is used as an argument to println(). This displays, in order: the name of the thread, its priority, and the name of its group.

```
Current thread: Thread[main,5,main]
After name change: Thread[My Thread,5,main]
5
4
3
2
1
```

- By default, the name of the main thread is main. Its priority is 5, which is the default value, and main is also the name of the group of threads to which this thread belongs.
- A thread group is a data structure that controls the state of a collection of threads as a whole.
- After the name of the thread is changed, t is again output. This time, the new name of the thread is displayed.

Creating a Thread:

- In the most general sense, you create a thread by instantiating an object of type Thread. Java defines two ways in which this can be accomplished:
 - You can implement the **Runnable** interface.
 - You can extend the **Thread** class, itself.

Implementing Runnable:

- The easiest way to create a thread is to create a class that implements the **Runnable** interface.
- **Runnable** abstracts a unit of executable code. You can construct a thread on any object that implements **Runnable**.
- To implement **Runnable**, a class need only implement a single method called **run()**, which is declared like this: `public void run();`
- Inside **run()**, you will define the code that constitutes the new thread.

-
- It is important to understand that **run()** can call other methods, use other classes, and declare variables, just like the main thread can.
 - The only difference is that **run()** establishes the entry point for another, concurrent thread of execution within your program. This thread will end when **run()** returns.
 - After you create a class that implements **Runnable**, you will instantiate an object of type **Thread** from within that class.
 - **Thread** defines several constructors. The one that we will use is shown here.....

Thread(Runnable threadOb, String threadName);

- In this constructor, threadOb is an instance of a class that implements the **Runnable** interface. This defines where execution of the thread will begin.
- The name of the new thread is specified by threadName.
- After the new thread is created, it will not start running until you call its **start()** method, which is declared within **Thread**.
- In essence, **start()** executes a call to **run()**. The **start()** method is shown here.....**void start();**
- Here is an example that creates a new thread and starts it running.....
 - Passing this as the first argument indicates that you want the new thread to call the **run()** method on this object.
 - Next, **start()** is called, which starts the thread of execution beginning at the **run()** method. This causes the child thread's for loop to begin.
 - After calling **start()**, **NewThread's** constructor returns to **main()**. When the main thread resumes, it enters its for loop. Both threads continue running, sharing the CPU, until their loops finish.
 - The output produced by this program is as follows. (Your output may vary based on processor speed and task load.)

```
// Create a second thread.
class NewThread implements Runnable {
    Thread t;

    NewThread() {
        // Create a new, second thread
        t = new Thread(this, "Demo Thread");
        System.out.println("Child thread: " + t);
        t.start(); // Start the thread
    }

    // This is the entry point for the second thread.
    public void run() {
        try {
            for(int i = 5; i > 0; i--) {
                System.out.println("Child Thread: " + i);
                Thread.sleep(500);
            }
        } catch (InterruptedException e) {
            System.out.println("Child interrupted.");
        }
        System.out.println("Exiting child thread.");
    }
}
```

```
class ThreadDemo {
    public static void main(String args[]) {
        new NewThread(); // create a new thread

        try {
            for(int i = 5; i > 0; i--) {
                System.out.println("Main Thread: " + i);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println("Main thread interrupted.");
        }
        System.out.println("Main thread exiting.");
    }
}
```

Inside **NewThread**'s constructor, a new **Thread** object is created by the following statement:

```
t = new Thread(this, "Demo Thread");
```

```
Child thread: Thread[Demo Thread,5,main]
Main Thread: 5           Main Thread: 3
Child Thread: 5          Child Thread: 1
Child Thread: 4          Exiting child thread.
Main Thread: 4           Main Thread: 2
Child Thread: 3           Main Thread: 1
Child Thread: 2           Main thread exiting.
```

Extending Thread:

- The second way to create a thread is to create a new class that extends **Thread**, and then to create an instance of that class.
 - The extending class must override the **run()** method, which is the entry point for the new thread.
 - It must also call **start()** to begin execution of the new thread.
- Here is the preceding program rewritten to extend **Thread**...
 - This program generates the same output as the preceding version.
 - As you can see, the child thread is created by instantiating an object of **NewThread**, which is derived from **Thread**.
 - Notice the call to **super()** inside **NewThread**. This invokes the following form of the **Thread** constructor: **public Thread(String threadName);**
 - Here, **threadName** specifies the name of the thread.

```

// Create a second thread by extending Thread
class NewThread extends Thread {

    NewThread() {
        // Create a new, second thread
        super("Demo Thread");
        System.out.println("Child thread: " + this);
        start(); // Start the thread
    }

    // This is the entry point for the second thread.
    public void run() {
        try {
            for(int i = 5; i > 0; i--) {
                System.out.println("Child Thread: " + i);
                Thread.sleep(500);
            }
        } catch (InterruptedException e) {
            System.out.println("Child interrupted.");
        }
        System.out.println("Exiting child thread.");
    }
}

class ExtendThread {
    public static void main(String args[]) {
        new NewThread(); // create a new thread

        try {
            for(int i = 5; i > 0; i--) {
                System.out.println("Main Thread: " + i);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println("Main thread interrupted.");
        }
        System.out.println("Main thread exiting.");
    }
}

```

Choosing an Approach:

- At this point, you might be wondering why Java has two ways to create child threads, and which approach is better.....?
 - The answers to these questions turn on the same point. The **Thread** class defines several methods that can be overridden by a derived class.

- Of these methods, the only one that *must* be overridden is **run()**. This is, of course, the same method required when you implement **Runnable**.
- Many Java programmers feel that classes should be extended only when they are being enhanced or modified in some way.
- So, if you will not be overriding any of **Thread**'s other methods, it is probably best simply to implement **Runnable**.

Creating Multiple Threads:

- So far, you have been using only two threads: the main thread and one child thread.
- However, your program can spawn as many threads as it needs. For example, the following program creates three child threads.

```
// Create multiple threads.
class NewThread implements Runnable {
    String name; // name of thread
    Thread t;

    NewThread(String threadname) {
        name = threadname;
        t = new Thread(this, name);
        System.out.println("New thread: " + t);
        t.start(); // Start the thread
    }

    // This is the entry point for thread.
    public void run() {
        try {
            for(int i = 5; i > 0; i--) {
                System.out.println(name + ": " + i);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println(name + "Interrupted");
        }
        System.out.println(name + " exiting.");
    }
}
```

```
class MultiThreadDemo {
    public static void main(String args[]) {
        new NewThread("One"); // start threads
        new NewThread("Two");
        new NewThread("Three");

        try {
            // wait for other threads to end
            Thread.sleep(10000);
        } catch (InterruptedException e) {
            System.out.println("Main thread Interrupted")
        }

        System.out.println("Main thread exiting.");
    }
}
```

The output from this program is shown here:

```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
New thread: Thread[Three,5,main]
One: 5
Two: 5
Three: 5
One: 4
Two: 4
Three: 4
One: 3
Three: 3
Two: 3
One: 2
Three: 2
Two: 2
One: 1
Three: 1
Two: 1
One exiting.
Two exiting.
Three exiting.
Main thread exiting.
```

Using `isAlive()` and `join()`:

- As mentioned, often you will want the main thread to finish last.
- In the preceding examples, this is accomplished by calling `sleep()` within `main()`, with a long enough delay to ensure that all child threads terminate prior to the main thread.
- However, this is hardly a satisfactory solution, and it also raises a larger question: How can one thread know when another thread has ended...?
 - Fortunately, `Thread` provides a means by which you can answer this question.
 - Two ways exist to determine whether a thread has finished. First, you can call `isAlive()` on the thread. This method is defined by `Thread`, and its general form is shown here:

`final boolean isAlive();`

- The `isAlive()` method returns `true` if the thread upon which it is called is still running. It returns `false` otherwise.
- While `isAlive()` is occasionally useful, the method that you will more commonly use to wait for a thread to finish is called `join()`, shown here...

`final void join() throws InterruptedException`

- This method waits until the thread on which it is called terminates. Its name comes from the concept of the calling thread waiting until the specified thread joins it.
- Additional forms of **join()** allow you to specify a maximum amount of time that you want to wait for the specified thread to terminate.
- Here is an improved version of the preceding example that uses.....
- **join()** to ensure that the main thread is the last to stop, and it also demonstrates the **isAlive()** method.

```
// Using join() to wait for threads to finish.
class NewThread implements Runnable {
    String name; // name of thread
    Thread t;

    NewThread(String threadname) {
        name = threadname;
        t = new Thread(this, name);
        System.out.println("New thread: " + t);
        t.start(); // Start the thread
    }

    // This is the entry point for thread.
    public void run() {
        try {
            for(int i = 5; i > 0; i--) {
                System.out.println(name + ": " + i);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println(name + " interrupted.");
        }
        System.out.println(name + " exiting.");
    }
}
```

```
class DemoJoin {
    public static void main(String args[]) {
        NewThread ob1 = new NewThread("One");
        NewThread ob2 = new NewThread("Two");
        NewThread ob3 = new NewThread("Three");

        System.out.println("Thread One is alive: " + ob1.t.isAlive());
        System.out.println("Thread Two is alive: " + ob2.t.isAlive());
        System.out.println("Thread Three is alive: " + ob3.t.isAlive());

        // wait for threads to finish
        try {
            System.out.println("Waiting for threads to finish.");
            ob1.t.join();
            ob2.t.join();
            ob3.t.join();
        } catch (InterruptedException e) {
            System.out.println("Main thread Interrupted");
        }
    }
}
```

```

        System.out.println("Thread One is alive: " + ob1.t.isAlive());
        System.out.println("Thread Two is alive: " + ob2.t.isAlive());
        System.out.println("Thread Three is alive: " + ob3.t.isAlive());

        System.out.println("Main thread exiting.");
    }
}

```

```

New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
New thread: Thread[Three,5,main]
Thread One is alive: true
Thread Two is alive: true
Thread Three is alive: true
Waiting for threads to finish.
One: 5
Two: 5
Three: 5
One: 4
Two: 4
Three: 4
One: 3
Two: 3
Three: 3
One: 2
Two: 2
Three: 2
One: 1
Two: 1
Three: 1
Two exiting.
Three exiting.
One exiting.
Thread One is alive: false
Thread Two is alive: false
Thread Three is alive: false
Main thread exiting.

```

Thread Priorities:

- Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run.
 - In theory, higher-priority threads get more CPU time than lower-priority threads.
 - In practice, the amount of CPU time that a thread gets often depends on several factors besides its priority.
- A higher-priority thread can also preempt a lower-priority one.
 - For instance, when a lower-priority thread is running and a higher-priority thread resumes (from sleep or wait), it will preempt the lower priority thread.

- In theory, threads of equal priority should get equal access to the CPU. Remember, Java is designed to work in a wide range of environments.
- For safety, threads that share the same priority should yield control once in a while. This ensures that all threads have a chance to run under a non-preemptive operating system.
- In practice, even in non-preemptive environments, most threads still get a chance to run, because most threads inevitably encounter some blocking situation, such as waiting for I/O.
- To set a thread's priority, use the **setPriority()** method, which is a member of **Thread**, its general form:

final void setPriority(int level);

- Here, level specifies the new priority setting for the calling thread.
- The value of level must be within the range **MIN_PRIORITY** and **MAX_PRIORITY**. Currently, these values are 1 and 10, respectively.
- To return a thread to default priority, specify **NORM_PRIORITY**, which is currently 5. These priorities are defined as **static final** variables within **Thread**.
- You can obtain the current priority setting by calling the **getPriority()** method of **Thread**, shown here.....

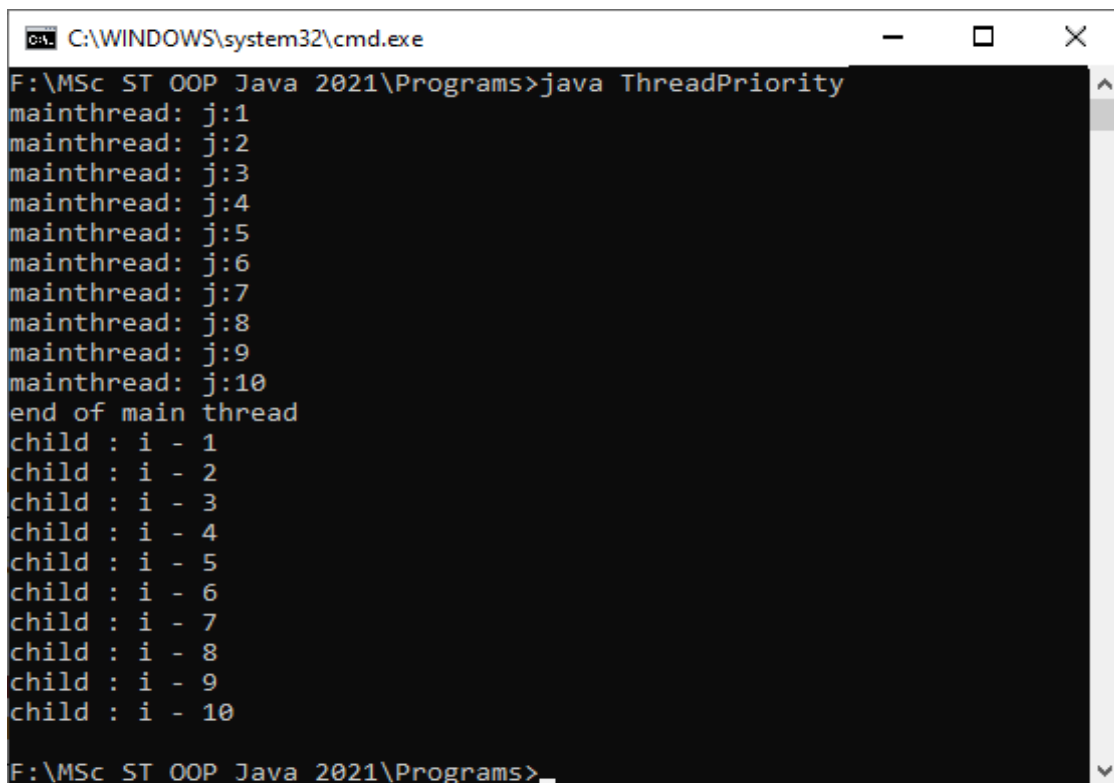
final int getPriority();

- The following example demonstrates two threads at different priorities, which do not run on a preemptive platform in the same way as they run on a non-preemptive platform.

```
class mythread extends Thread{
    mythread(String name){
        super(name);
        start();
    }
    public void run(){
        setPriority(MIN_PRIORITY);
        try{
            sleep(200);
        }catch (Exception e){
        }
        for(int i=1;i<=10;i++){
            System.out.println(getName()+" : i - "+i);
        }
    }
}
```

```
public class ThreadPriority {
    public static void main(String[] args) {
        mythread th1 = new mythread("child");

        Thread mt = Thread.currentThread();
        mt.setName("mainthread");
        mt.setPriority(Thread.MAX_PRIORITY);
        for(int j=1;j<=10;j++){
            System.out.println("mainthread: j:"+j);
        }
        System.out.println("end of main thread");
    }
}
```



```
C:\WINDOWS\system32\cmd.exe
F:\MSc ST OOP Java 2021\Programs>java ThreadPriority
mainthread: j:1
mainthread: j:2
mainthread: j:3
mainthread: j:4
mainthread: j:5
mainthread: j:6
mainthread: j:7
mainthread: j:8
mainthread: j:9
mainthread: j:10
end of main thread
child : i - 1
child : i - 2
child : i - 3
child : i - 4
child : i - 5
child : i - 6
child : i - 7
child : i - 8
child : i - 9
child : i - 10
F:\MSc ST OOP Java 2021\Programs>
```

Synchronization:

- When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time.
- The process by which this is achieved is called synchronization. As you will see, Java provides unique, language-level support for it.
- Key to synchronization is the concept of the monitor (also called a semaphore). A monitor is an object that is used as a mutually exclusive lock, or mutex.
 - Only one thread can own a monitor at a given time.

-
- When a thread acquires a lock, it is said to have entered the monitor.
 - All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor.
 - These other threads are said to be waiting for the monitor.
 - A thread that owns a monitor can reenter the same monitor if it so desires.
 - If you have worked with synchronization when using other languages, such as C or C++, you know that it can be a bit tricky to use. This is because
 - These languages, to synchronize threads, your programs need to utilize operating system primitives.
 - Fortunately, because Java implements synchronization through language elements, most of the complexity associated with synchronization has been eliminated.
 - You can synchronize your code in either of two ways. Both involve the use of the **synchronized** keyword, and both are examined here.

Using Synchronized Methods:

- Synchronization is easy in Java, because all objects have their own implicit monitor associated with them.
 - To enter an object's monitor, just call a method that has been modified with the **synchronized** keyword.
 - While a thread is inside a synchronized method, all other threads that try to call it (or any other synchronized method) on the same instance have to wait.
 - To exit the monitor and relinquish control of the object to the next waiting thread, the owner of the monitor simply returns from the synchronized method.
- To understand the need for synchronization, let's begin with a simple example that does not use it - but should. The following program has three simple classes.....
 - The first one, **Callme**, has a single method named **call()**.
 - The **call()** method takes a **String** parameter called **msg**. This method tries to print the **msg** string inside of square brackets.
 - The interesting thing to notice is that after **call()** prints the opening bracket and the **msg** string, it calls **Thread.sleep(1000)**, which pauses the current thread for one second.
 - The constructor of the next class, **Caller**, takes a reference to an instance of the **Callme** class and a **String**, which are stored in **target** and **msg**, respectively.

- The constructor also creates a new thread that will call this object's `run()` method. The thread is started immediately.
- The `run()` method of **Caller** calls the `call()` method on the **target** instance of **Callme**, passing in the `msg` string.
- Finally, the **Synch** class starts by creating a single instance of **Callme**, and three instances of **Caller**, each with a unique message string.
- The same instance of **Callme** is passed to each **Caller**.

```
// This program is not synchronized.
class Callme {
    void call(String msg) {
        System.out.print "[" + msg);
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            System.out.println("Interrupted");
        }
        System.out.println("]");
    }
}
```

```
class Caller implements Runnable {
    String msg;
    Callme target;
    Thread t;

    public Caller(Callme targ, String s) {
        target = targ;
        msg = s;
        t = new Thread(this);
        t.start();
    }

    public void run() {
        target.call(msg);
    }
}
```

```
class Synch {
    public static void main(String args[]) {
        Callme target = new Callme();
        Caller ob1 = new Caller(target, "Hello");
        Caller ob2 = new Caller(target, "Synchronized");
        Caller ob3 = new Caller(target, "World");

        // wait for threads to end
        try {
            ob1.t.join();
            ob2.t.join();
            ob3.t.join();
        } catch (InterruptedException e) {
            System.out.println("Interrupted");
        }
    }
}
```

Here is the output produced by this program:

```
Hello [Synchronized [World]
]
]
```

-
- As you can see, by calling `sleep()`, the `call()` method allows execution to switch to another thread. This results in the mixed-up output of the three message strings.
 - In this program, nothing exists to stop all three threads from calling the same method, on the same object, at the same time.
 - This is known as a race condition, because the three threads are racing each other to complete the method.
 - This example used `sleep()` to make the effects repeatable and obvious.
 - In most situations, a race condition is more subtle and less predictable, because you can't be sure when the context switch will occur.
 - This can cause a program to run right one time and wrong the next.
 - To fix the preceding program, you must serialize access to `call()`. That is, you must restrict its access to only one thread at a time.
 - To do this, you simply need to precede `call()`'s definition with the keyword **synchronized**, as shown here.....

```
class Callme {  
    synchronized void call(String msg) {  
        ...  
    }  
}
```

- This prevents other threads from entering `call()` while another thread is using it. After **synchronized** has been added to `call()`, the output of the program is as follows:

```
[Hello]  
[Synchronized]  
[World]
```

- Any time that you have a method, or group of methods, that manipulates the internal state of an object in a multithreaded situation, you should use the **synchronized** keyword to guard the state from race conditions.
- Remember, once a thread enters any synchronized method on an instance, no other thread can enter any other synchronized method on the same instance. However, non-synchronized methods on that instance will continue to be callable.

The **synchronized** Statement:

- While creating **synchronized** methods within classes that you create is an easy and effective means of achieving synchronization, it will not work in all cases.

-
- To understand why, consider the following.....
 - Imagine that you want to synchronize access to objects of a class that was not designed for multithreaded access. That is, the class does not use **synchronized** methods.
 - Further, this class was not created by you, but by a third party, and you do not have access to the source code. Thus, you can't add **synchronized** to the appropriate methods within the class.
 - How can access to an object of this class be synchronized?
 - Fortunately, the solution to this problem is quite easy: You simply put calls to the methods defined by this class inside a **synchronized** block.
 - This is the general form of the **synchronized** statement.

```
synchronized(object) {  
    // statements to be synchronized  
}
```

- Here, object is a reference to the object being synchronized.
- A synchronized block ensures that a call to a method that is a member of object occurs only after the current thread has successfully entered object's monitor.
- Here is an alternative version of the preceding example, using a synchronized block within the **run()** method.
 - Here, the **call()** method is not modified by **synchronized**.
 - Instead, the **synchronized** statement is used inside Caller's **run()** method.
 - This causes the same correct output as the preceding example, because each thread waits for the prior one to finish before proceeding.

SAMPLE QUESTIONS:

1. Define Exception. Explain the effect and general syntax of handling an exception with keywords used to handle the exception.
2. Define an Exception. Explain any four built-in exceptions with its purpose.
3. Write a java program to illustrate the working and use of multi-catch blocks while handling exceptions.
4. Write a Java program to accept the values (integers) as command line arguments and count the number of palindromes. Handle necessary exceptions i.e. NumberFormatException.
5. Explain the working and use of nested try-catch blocks while handling exceptions.

-
6. Explain the use of throw and throws keyword in context of checked exception with an example.
 7. Define your own exception SumDigitException{int sumdigit} with necessary methods, the same can be thrown while accepting 'N' integers into an array if the given integer's sum of digits is more than 7 and use necessary try-catch block to handle the same.
 8. Define a stream. Explain the importance of InputStream and OutputStream.
 9. Explain the byte and character stream with supporting classes used for it.
 10. Write a java program to copy the all the contents of a file "input.txt" to "output.txt" using FileReader and FileWriter classes.
 11. Write a java program to accept a line of text from the user and write the same into a file "input.txt". Now while reading from the file display only vowels along with count of the same.
 12. Define a thread. Explain the life cycle of a thread.
 13. Write a java program to illustrate getting reference of main thread with few methods.
 14. Explain the different ways of creating a thread with its general syntax and an example.
 15. Define a thread. Explain any five methods available in Thread class.
 16. Explain the use and effect of join() method while running multiple threads.
 17. Write a java program to illustrate the creation of multiple threads and its execution.
 18. Differentiate the use Synchronized methods and Synchronized blocks with an example.
 19. Write a java program to illustrate the use and effect of Synchronized methods while multiple threads sharing the same data.
