Unit IV

Exception Handling

Exception

- An exception is a run-time error.
- An *exception is an object that's created when an error occurs in a Java program*
- Java can't automatically fix the error.
- The exception object contains information about the type of error that occurred.
- The cause of the error — name of the exception class used to create the exception.

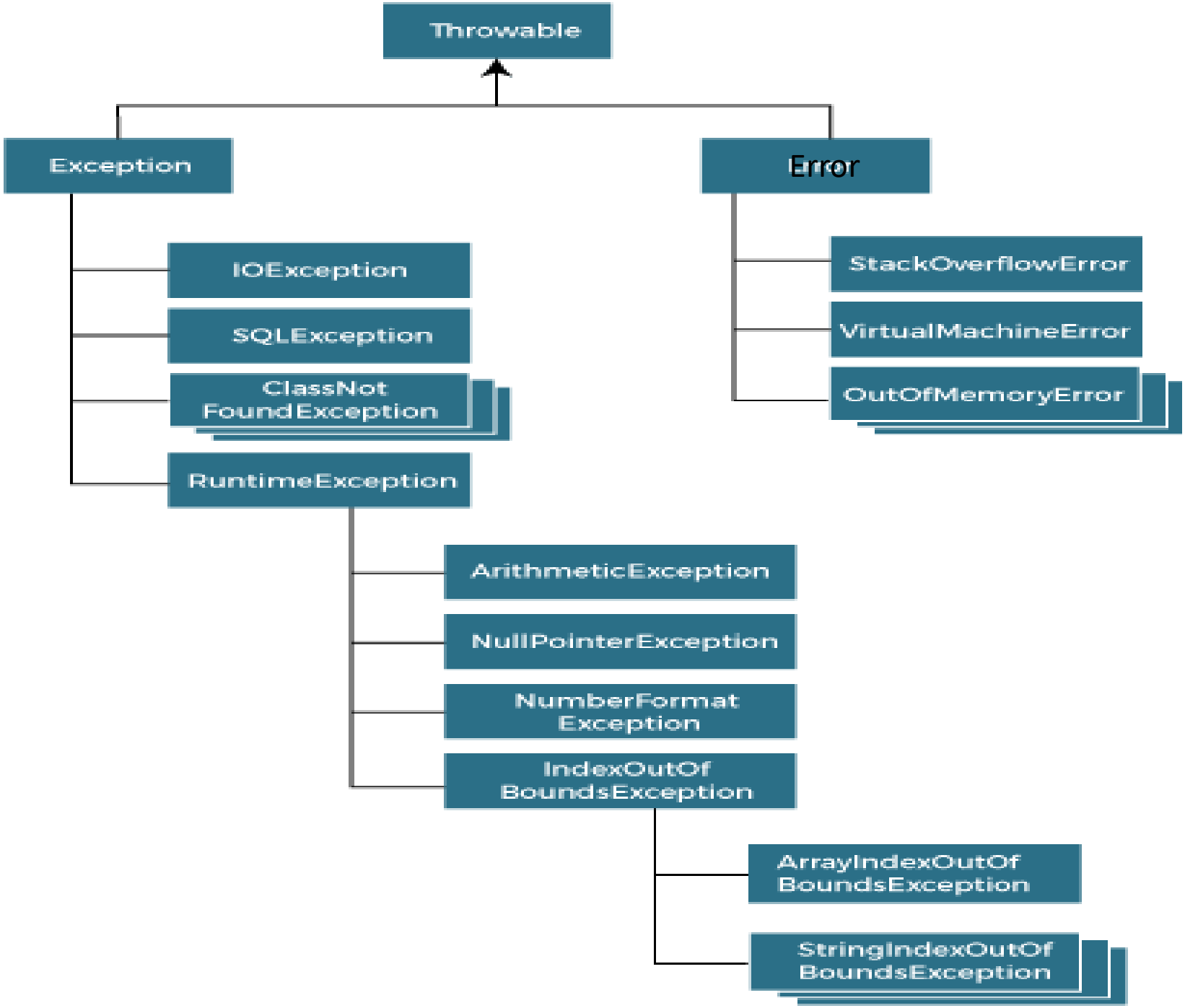
- The **Exception Handling in Java** is one of the powerful *mechanism to handle the runtime errors* so that the normal flow of the application can be maintained.
- Exception is an abnormal condition.
- In Java, an exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

What is Exception Handling?

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

- 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.
- 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.
- Java exception handling is managed via five keywords: **try, catch, throw, throws, and finally.**


Throwable
Exception
Error
IOException
SQLException
ClassNot FoundException
RuntimeException
StackOverflowError
VirtualMachineError
OutOfMemoryError
ArithmeticException
NullPointerException
NumberFormat Exception
IndexOutOf BoundsException
ArrayIndexOutOf BoundsException
StringIndexOutOf BoundsException

```
public class TryCatchExample {

    public static void main(String[] args) {

        int data=50/0; //may throw exception

        System.out.println("rest of the code");

    }

}
```

1) Checked Exception

The classes that directly inherit the Throwable class except RuntimeException and Error are known as checked exceptions. For example, IOException, SQLException, etc. Checked exceptions are checked at compile-time.

2) Unchecked Exception

The classes that inherit the RuntimeException are known as unchecked exceptions. For example, ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException, etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

3) Error

Error is irrecoverable. Some example of errors are OutOfMemoryError, VirtualMachineError, AssertionError etc.

try	**The "try" keyword is used to specify a block where we should place an exception code. It means we can't use try block alone. The try block must be followed by either catch or finally.**
catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
finally	The "finally" block is used to execute the necessary code of the program. It is executed whether an exception is handled or not.
throw	The "throw" keyword is used to throw an exception.
throws	The "throws" keyword is used to declare exceptions. It specifies that there may occur an exception in the method. It doesn't throw an exception. It is always used with method signature.

```
public class JavaExceptionExample{
  public static void main(String args[]){
   try{
      //code that may raise exception
      int data=100/0;
   }catch(ArithmeticException e){System.out.println(e);}
   //rest code of the program
   System.out.println("rest of the code...");
  }
}
```

In the above example, 100/0 raises an ArithmeticException which is handled by a try-catch block.

Scenario where ArithmeticException occurs

- If we divide any number by zero, there occurs an ArithmeticException.
- **int** a=50/0;//ArithmeticException

- scenario where NullPointerException occurs
- If we have a null value in any variable, performing any operation on the variable throws a NullPointerException.
- String s=**null**;
- System.out.println(s.length());//NullPointerException

- scenario where NumberFormatException occurs
- If the formatting of any variable or number is mismatched, it may result into NumberFormatException. Suppose we have a string variable that has characters; converting this variable into digit will cause NumberFormatException.
- String s="abc";
- **int** i=Integer.parseInt(s);//NumberFormatExceptio n

- scenario where ArrayIndexOutOfBoundsException occurs
- When an array exceeds to it's size, the ArrayIndexOutOfBoundsException occurs. there may be other reasons to occur ArrayIndexOutOfBoundsException. Consider the following statements.
- **int** a[]=**new int**[5];
- a[10]=50; //ArrayIndexOutOfBoundsException

```
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
}
```

- Java try block
- Java **try** block is used to enclose the code that might throw an exception. It must be used within the method.
- If an exception occurs at the particular statement in the try block, the rest of the block code will not execute. So, it is recommended not to keep the code in try block that will not throw an exception.
- Java try block must be followed by either catch or finally block.
- Syntax of Java try-catch
- **try**{
- //code that may throw an exception
- }**catch**(Exception_class_Name ref){}

- **try**{
- //code that may throw an exception
- }**finally**{}

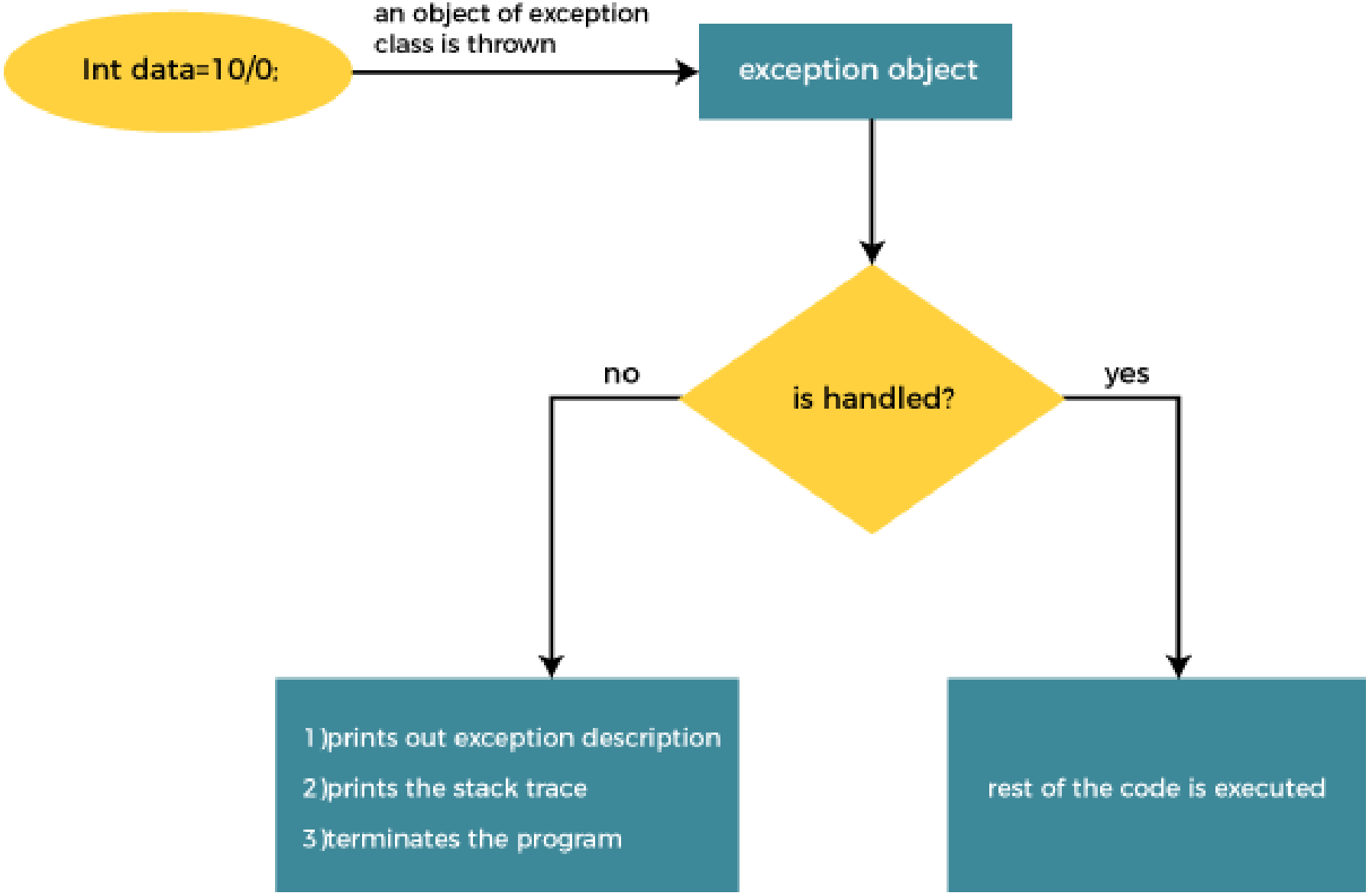
- Java catch block
- Java catch block is used to handle the Exception by declaring the type of exception within the parameter. The declared exception must be the parent class exception (i.e., Exception) or the generated exception type. However, the good approach is to declare the generated type of exception.

```
class Demo
{
	public static void main(String args[])
	{
		int a,b,c;

		try
		{
			a = Integer.parseInt(args[0]);
			b = Integer.parseInt(args[1]);
			c = 0;
			c = a/b;
			System.out.println(c);
		}
		catch(ArithmeticException e)
		{
			System.out.println("Divide by zero is not
possible");
		}
		catch(ArrayIndexOutOfBoundsException e)
		{
			System.out.println("Enter atleast 2 values");
		}
		finally
		{
			System.out.println("Inside finally");
		}
		System.out.println("Last line");
	}
}
```

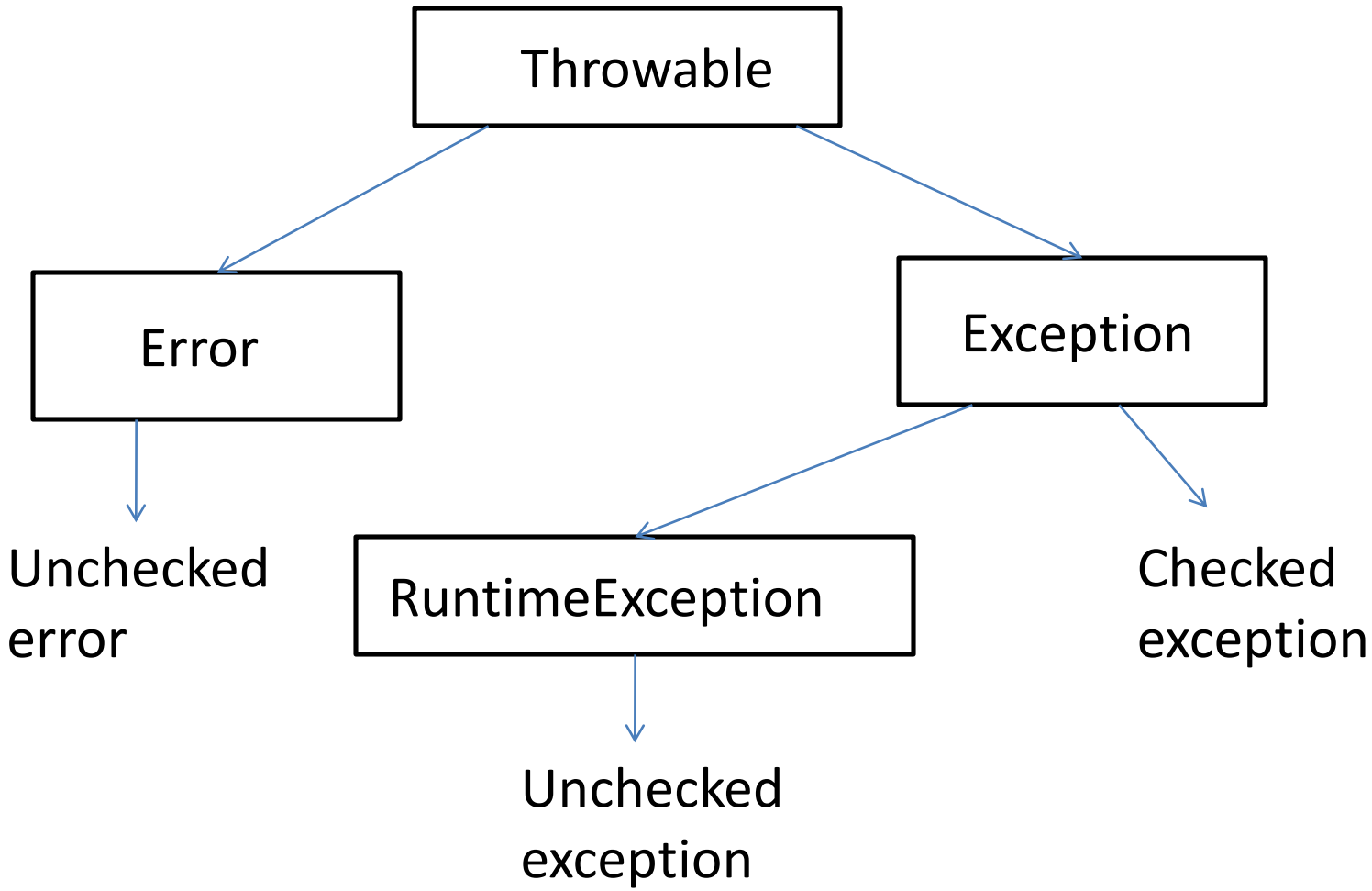
finally Block

- executes
  - after a **try/catch block has** completed
  - before the code following the **try/catch block**
  - 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.**


Int data=10/0;
an object of exception class is thrown
exception object
is handled?
no
yes
1)prints out exception description
2)prints the stack trace
3)terminates the program
rest of the code is executed

- The JVM firstly checks whether the exception is handled or not. If exception is not handled, JVM provides a default exception handler that performs the following tasks:
- Prints out exception description.
- Prints the stack trace (Hierarchy of methods where the exception occurred).
- Causes the program to terminate.
- But if the application programmer handles the exception, the normal flow of the application is maintained, i.e., rest of the code is executed.

Exception Types



Checked Exceptions

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

Checked vs. Unchecked Exceptions

In Java, there are two types of exceptions:

- **Checked:**
  - The exceptions that are checked at compile time.
  - If some code within a method throws a checked exception, then the method must either handle the exception or it must specify the exception using *throws* keyword.

- **Unchecked**
  - The exceptions that are not checked at compiled time.
  - It is not forced by the compiler to either handle or specify the exception.
  - It is up to the programmers to be civilized, and specify or catch the exceptions.

Unchecked Exceptions

Exception	Meaning
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalThreadStateException	Requested operation not compatible with current thread state.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
UnsupportedOperationException	An unsupported operation was encountered.

exceptions allows us to write good quality codes where the errors are checked at the compile time instead of runtime and we can create custom exceptions making the code recovery and debugging easier.

Java throw keyword

The Java throw keyword is used to throw an exception explicitly. We specify the **exception** object which is to be thrown. The Exception has some message with it that provides the error description. These exceptions may be related to user inputs, server, etc.

We can throw either checked or unchecked exceptions in Java by throw keyword. It is mainly used to throw a custom exception. We will discuss custom exceptions later in this section.

- We can also define our own set of conditions and throw an exception explicitly using throw keyword. For example, we can throw ArithmeticException if we divide a number by another number. Here, we just need to set the condition and throw exception using throw keyword.
- The syntax of the Java throw keyword is given below.
- throw Instance i.e.,
- **throw new** exception_class("error message");

throw

- It is possible to throw an exception explicitly from any method or block, using the **throw** statement.

```
throw ThrowableInstance;
```

e.g. `throw new ArithmeticException("/ by zero");`

- *ThrowableInstance* must be an object of type **Throwable** or a subclass of **Throwable.**
- The **throw** keyword is mainly used to throw custom exceptions.

Throwing Unchecked Exception

```
public class TestThrow {
   //function to check if person is eligible to vote or not
   public static void validate(int age) {
      if(age<18) {
         //throw Arithmetic exception if not eligible to vote
         throw new ArithmeticException("Person is not eligible to vote");
      }
      else {
         System.out.println("Person is eligible to vote!!");
      }
   }
   //main method
   public static void main(String args[]){
      //calling the function
      validate(13);
      System.out.println("rest of the code...");
  }
}
```

Throwing Checked Exception

```
import java.io.*;

public class TestThrow2 {

   //function to check if person is eligible to vote or not
   public static void method() throws FileNotFoundException {

      FileReader file = new FileReader("C:\\Users\\Anurati\\Desktop\\abc.txt");
      BufferedReader fileInput = new BufferedReader(file);

      throw new FileNotFoundException();
    }
    public static void main(String args[]){    //main method
      try
      {
         method();
      }
      catch (FileNotFoundException e)
      {
         e.printStackTrace();
      }
      System.out.println("rest of the code...");
  }
}
```

Throwing User-defined Exception

```
// class represents user-defined exception
class UserDefinedException extends Exception
{
    public UserDefinedException(String str)
    {
        // Calling constructor of parent Exception
        super(str);
    }
}
// Class that uses above MyException
public class TestThrow3
{
    public static void main(String args[])
    {
        try
        {
            // throw an object of user defined exception
            throw new UserDefinedException("This is user-defined exception");
        }
        catch (UserDefinedException ude)
        {
            System.out.println("Caught the exception");
            // Print the message from MyException object
            System.out.println(ude.getMessage());
        }
    }
}
```

- There are two ways you can obtain a **Throwable** object**:**
  - using a parameter in a **catch** clause
  - 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
- If no matching catch is found, then the default exception handler halts the program

throws

- 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.
- 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.
- Exceptions that a method can throw must be declared in the **throws** clause**.**
- Otherwise, a compile-time error will result.

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

```
import java.io.IOException;
class Testthrows{
  void m()throws IOException{
    throw new IOException("device error");//checked exception
  }
  void n()throws IOException{
    m();
  }
  void p(){
   try{
    n();
   }catch(Exception e){System.out.println("exception handled");}
  }
  public static void main(String args[]){
   Testthrows obj=new Testthrows1();
   obj.p();
   System.out.println("normal flow...");
  }
}
```

```
import java.io.*;
class M{
 void method()throws IOException{
  System.out.println("device operation performed");
 }
}
class Testthrows{
   public static void main(String args[])throws IOException{//d
     eclare exception
     M m=new M();
     m.method();

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

```
/* 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();
      }
}
```

```
// 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);
            }
      }
}
```

Creating user defined exceptions

- Handle situations specific to your applications.
- Define a subclass of **Exception.**
- The **Exception** class does not define any methods of its own, but inherit from **Throwable.**

References:

Herbert Schildt, Complete Reference Java (8th edition), Tata McGraw Hill Edition