- **Java Exceptions**

Exceptions are runtime errors. Exception handling enables a program to deal with runtime errors and continue its normal execution. Runtime errors occur while a program is running if the JVM detects an operation that is impossible to carry out. For example, if you access an array using an index that is out of bounds, you will get a runtime error with an **ArrayIndexOutOfBoundsException**. If you enter a double value when your program expects an integer, you will get a runtime error with an **InputMismatchException**. There are several causes for Java exceptions:

- Caused by the users—key in wrong data
  - Values out of range
  - Divided by 0
- Caused by program errors – bugs
- Errors outside of program control
  - File doesn't exist, or
  - Some external resources unavailable
- Recovery from unusual, but not unexpected event

In Java, runtime errors are thrown as exceptions. An **exception** is an object that represents an error or a condition that prevents execution from proceeding normally. If the exception is not handled, the program will terminate abnormally. How can you handle the exception so the program can continue to run or else terminate gracefully?

Exceptions are thrown from a method. The caller of the method can catch and handle the exception. An exception handler is a piece of program code that is automatically invoked when an exception occurs. Java has a hierarchy of classes that support exception handling. Relevant Java syntax for handling the exceptions are **try**, **catch**, **finally** blocks and **throws** clause. For example,

```
import java.lang.ArithmeticException;
import java.util.Scanner;

public class QuotientWithException {
    public static int quotient(int num1, int num2) {
        if (num2 == 0)
            throw new ArithmeticException("divisor cannot be zero.");
        return num1 / num2;
    }

    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.println("Enter 2 integers.");
        int num1 = input.nextInt();
        int num2 = input.nextInt();
        try {
            System.out.println(num1 + " / " + num2 + " = "
                               + quotient(num1, num2) + "\n");
        }
        catch (ArithmeticException e) { //e is the exception object
            System.out.println(e.toString() + "\n");
            input.close();
        }
          System.out.println("Program teminated");
    }
}
```

The value thrown, in this case **new ArithmeticException("divisor cannot be zero")**, is called an exception. The execution of a throw statement is called **throwing an exception**. The exception is an object created from an exception class. In this case, the exception class is **java.lang.ArithmeticException**. The constructor ArithmeticException(str) is invoked to construct an exception object, where str is a message that describes the exception.

- **Java Throwable class**

An exception in Java is treated as an object that is an instance of the super class **java.lang.Throwable**, or an instance of one of its subclasses. **Throwable class** is the superclass of all errors and exceptions in the Java language. Only objects that are instances of this class (or one of its subclasses) are thrown by the Java Virtual Machine (JVM) or can be thrown by the Java throws statement. Only this class or one of its subclasses can be the argument type in a catch clause Below is a list of subclasses extended from the Throwable class.

- java.lang.**Throwable** (implements java.io.Serializable)
  - java.lang.**Error**
    - java.lang.**AssertionError**
    - java.lang.**LinkageError**
      - java.lang.**BootstrapMethodError**
      - java.lang.**ClassCircularityError**
      - java.lang.**ClassFormatError**
        - java.lang.**UnsupportedClassVersionError**
      - java.lang.**ExceptionInInitializerError**
      - java.lang.**IncompatibleClassChangeError**
        - java.lang.**AbstractMethodError**
        - java.lang.**IllegalAccessError**
        - java.lang.**InstantiationError**
        - java.lang.**NoSuchFieldError**
        - java.lang.**NoSuchMethodError**
      - java.lang.**NoClassDefFoundError**
      - java.lang.**UnsatisfiedLinkError**
      - java.lang.**VerifyError**
    - java.lang.**ThreadDeath**
    - java.lang.**VirtualMachineError**
      - java.lang.**InternalError**
      - java.lang.**OutOfMemoryError**
      - java.lang.**StackOverflowError**
      - java.lang.**UnknownError**
  - java.lang.**Exception**
    - java.lang.**CloneNotSupportedException**
    - java.lang.**InterruptedException**
    - java.lang.**ReflectiveOperationException**
      - java.lang.**ClassNotFoundException**
      - java.lang.**IllegalAccessException**
      - java.lang.**InstantiationException**
      - java.lang.**NoSuchFieldException**
      - java.lang.**NoSuchMethodException**
    - java.lang.**RuntimeException**
      - java.lang.**ArithmeticException**
      - java.lang.**ArrayStoreException**

- java.lang.**Exception**
  - java.lang.**CloneNotSupportedException**
  - java.lang.**InterruptedException**
  - java.lang.**ReflectiveOperationException**
    - java.lang.**ClassNotFoundException**
    - java.lang.**IllegalAccessException**
    - java.lang.**InstantiationException**
    - java.lang.**NoSuchFieldException**
    - java.lang.**NoSuchMethodException**
  - java.lang.**RuntimeException**
    - java.lang.**ArithmeticException**
    - java.lang.**ArrayStoreException**
    - java.lang.**ClassCastException**
    - java.lang.**EnumConstantNotPresentException**
    - java.lang.**IllegalArgumentException**
      - java.lang.**IllegalThreadStateException**
      - java.lang.**NumberFormatException**
    - java.lang.**IllegalCallerException**
    - java.lang.**IllegalMonitorStateException**
    - java.lang.**IllegalStateException**
    - java.lang.**IndexOutOfBoundsException**
      - java.lang.**ArrayIndexOutOfBoundsException**
      - java.lang.**StringIndexOutOfBoundsException**
    - java.lang.**LayerInstantiationException**
    - java.lang.**NegativeArraySizeException**
    - java.lang.**NullPointerException**
    - java.lang.**SecurityException**
    - java.lang.**TypeNotPresentException**
    - java.lang.**UnsupportedOperationException**

Each exception class in the Java API has at least two constructors: a default constructor and a constructor with a string parameter that describes the exception. This argument is called the exception message, which can be obtained by invoking **getMessage()** from an exception object. When an error occurs within a method, the method creates an object and hands it off to the runtime system. The object, called an exception object, contains information about the error, including its type and the state of the program when the error occurred. Creating an exception object and handing it to the runtime system is called **throwing an exception**. The Throwable class provides a list of useful methods as show below for getting the detail information from the exception object.

When an exception is thrown, the normal execution flow is interrupted. As the name suggests, to "throw an exception" is to pass the exception from one place to another. The statement for invoking the method is contained in a try block. The try block contains the code that is executed in normal circumstances. The exception is caught by the catch block. The code in the catch block is executed to handle the exception. Afterward, the statement after the catch block is executed.

Modifier and Type	Method	Description
void	addSuppressed(Throwable exception)	Appends the specified exception to the exceptions that were suppressed in order to deliver this exception.
Throwable	fillInStackTrace()	Fills in the execution stack trace.
Throwable	getCause()	Returns the cause of this throwable or null if the cause is nonexistent or unknown.
String	getLocalizedMessage()	Creates a localized description of this throwable.
String	getMessage()	Returns the detail message string of this throwable.
StackTraceElement[]	getStackTrace()	Provides programmatic access to the stack trace information printed by printStackTrace().
Throwable[]	getSuppressed()	Returns an array containing all of the exceptions that were suppressed, typically by the try-with-resources statement, in order to deliver this exception.
Throwable	initCause(Throwable cause)	Initializes the *cause* of this throwable to the specified value.
void	printStackTrace()	Prints this throwable and its backtrace to the standard error stream.
void	printStackTrace(PrintStream s)	Prints this throwable and its backtrace to the specified print stream.
void	printStackTrace(PrintWriter s)	Prints this throwable and its backtrace to the specified print writer.
void	setStackTrace (StackTraceElement[] stackTrace)	Sets the stack trace elements that will be returned by getStackTrace() and printed by printStackTrace() and related methods.
String	toString()	Returns a short description of this throwable.

The **Throwable class** is the root of exception classes. All Java exception classes inherit directly or indirectly from Throwable. You can create your own exception classes by extending **Exception** or a subclass of Exception. The exception classes can be classified into three major types: system errors, exceptions, and runtime exceptions. **System error**s are thrown by the JVM and are represented in the Error class. The **Error class** describes internal system errors, though such errors rarely occur. If one does, there is little you can do beyond notifying the user and trying to terminate the program gracefully. For example, LinkageError or VirtualMachineError. **Exceptions** are represented in the **Exception class**, which describes errors caused by your program and by external circumstances. These errors can be caught and handled by your program. **Runtime exceptions** are represented in the **RuntimeException class**, which describes programming errors, such as bad casting, accessing an out-of-bounds array, and numeric errors. Runtime exceptions normally indicate programming errors.

RuntimeException, Error, and their subclasses are known as **unchecked exceptions**. All other exceptions are known as **checked exceptions**, meaning the compiler forces the programmer to check and deal with them in a try-catch block or declare it in the method header. For example, the IOException class.

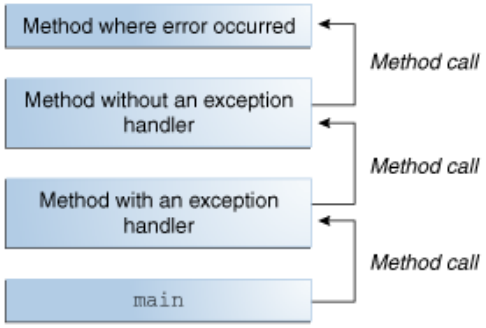
- java.lang.**Throwable** (implements java.io.Serializable)
  - java.lang.**Error**
    - java.io.**IOError**
  - java.lang.**Exception**
    - java.io.**IOException**
      - java.io.**CharConversionException**
      - java.io.**EOFException**
      - java.io.**FileNotFoundException**
      - java.io.**InterruptedIOException**
      - java.io.**ObjectStreamException**
        - java.io.**InvalidClassException**
        - java.io.**InvalidObjectException**
        - java.io.**NotActiveException**
        - java.io.**NotSerializableException**
        - java.io.**OptionalDataException**
        - java.io.**StreamCorruptedException**
        - java.io.**WriteAbortedException**
      - java.io.**SyncFailedException**
      - java.io.**UnsupportedEncodingException**
      - java.io.**UTFDataFormatException**
    - java.lang.**RuntimeException**
      - java.io.**UncheckedIOException**

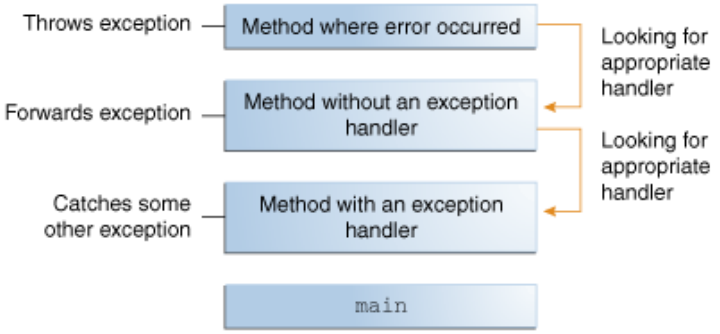
In most cases, unchecked exceptions reflect programming logic errors that are unrecoverable. For example, a **NullPointerException** is thrown if you access an object through a reference variable before an object (reference) is assigned to it; an **IndexOutOfBoundsException** is thrown if you access an element in an array outside the bounds of the array. These are logic errors that should be corrected in the program. Unchecked exceptions can occur anywhere in a program. To avoid cumbersome overuse of try-catch blocks, Java does not mandate that you write code to catch or declare unchecked exceptions.

However, **Java forces you to deal with checked exceptions**. If a method declares a checked exception (i.e., an exception other than Error or RuntimeException), you must invoke it in a try-catch block or declare to throw the exception in the calling method.

- **Handling the Exceptions**

A "handler" for an exception is found by propagating the exception backward through a chain of method calls, starting from the current method where the exception is thrown. When a method throws an exception, the runtime system attempts to find something to handle it. The set of possible "somethings" to handle the exception is the ordered list of methods that had been called to get to the method where the error occurred.





The runtime system searches the call stack for a method that contains the exception handler. The search begins with the method in which the error occurred and proceeds through the call stack in the reverse order in which the methods were called. When an appropriate handler is found, the runtime system passes the exception to the handler. The exception handler chosen is said to catch the exception. If the runtime system exhaustively searches all the methods on the call stack without finding an appropriate exception handler, the runtime system terminates.

Java's exception-handling model is based on three operations: declaring an exception, throwing an exception, and catching an exception. The throw statement is analogous to a method call, but instead of calling a method, it calls a **catch block**. In this sense, a catch block is like a method definition with a parameter that matches the type of the value being thrown. Unlike a method, however, after the catch block is executed, **the program control does not return to the throw statement; instead, it executes the next statement after the catch block.**

In the previous example, the identifier **e** in the catch–block header: **catch (ArithmeticException e)** acts very much like a parameter in a method. Thus, this parameter is referred to as a **catch–block parameter**. The type (e.g., ArithmeticException) preceding the exception object **e** specifies what kind of exception the catch block can catch. Once the exception is caught, you can access the thrown value from this parameter **e** in the body of a catch block.

The advantage of using exception handling is – it enables a method to throw an exception to its caller, enabling the caller to handle the exception. Without this capability, the called method itself must handle the exception or terminate the program. But oftentimes the method being called does not know what to do in case of an error. Therefore, the exception is thrown back to the calling method to properly handle the error. The key benefit of exception handling is separating the detection of an error (done in a called method) from the handling of an error (done in the calling method).

The keyword to declare an exception is **throws**, and the keyword to throw an exception is **throw**. For example: **public void myMethod() throws Exception1, Exception2, ..., ExceptionN** in a method header is to declare what exceptions might occur in the method. On the other hand, **throw new IllegalArgumentException("wrong argument");** statement in the try block is to actually throw an exception.

- **Catching the Exceptions**

When an exception is thrown, it can be caught and handled in a try-catch block.

```
try {
   //code that could throw exceptions
}
catch {
   //exception handler
}
finally {
   //a finally block is always executed even if the try
   //block runs without throwing an exception
}
```

```
try {
...
}
catch (IndexOutOfBoundsException e) {
   System.err.println("IndexOutOfBoundsException: "
         + e.getMessage());
}
catch (IOException e) {
   System.err.println("Caught IOException: " + e.getMessage());
}
catch (...) {
  ...
}
...
```

Each catch block is an exception handler

A **try block** can have one or more catch blocks. If no exceptions arise during the execution of the try block, the catch blocks are skipped. If one of the statements inside the try block throws an exception, Java skips the remaining statements in the try block and starts the process of finding the code to handle the exception. Each catch block is examined sequentially, from the first to the last, to see whether the type of the exception object is an instance of the exception class in the catch block. If so, the exception object is assigned to the variable declared and the code in the catch block is executed. If no handler is found, Java exits this method, passes the exception to the method's caller, and continues the same process to find a handler. If no handler is found in the chain of methods being invoked, the program terminates and prints an error message on the console.

If an exception is thrown, only the **first matching catch block is executed**. Various exception classes can be derived from a common superclass. If a catch block catches exception objects of a superclass, it can catch all the exception objects of the subclasses of that superclass. Therefore, **the order in which exceptions are specified in catch blocks is important**. A compile error will result if a catch block for a superclass type appears before a catch block for a subclass type. For example, the ordering in (a) below is erroneous, because RuntimeException is a subclass of Exception. The correct ordering should be as shown in (b).

(a)

```
try { ... }
catch (Exception ex) { ... }
catch (RuntimeException ex) { ... }
```

(b)

```
try { ... }
catch (RuntimeException ex) { ... }
catch (Exception ex) { ... }
```

- **Getting information about the Exceptions**

As mentioned earlier, an exception object contains valuable information about the exception. You may use the instance methods in the **java.lang.Throwable class** to get information regarding the exception. For example, in addition to the getMessage() and toString() methods, the **printStackTrace()** method prints stack trace information on the console. The stack trace lists all the methods in the call stack, which provides valuable information for debugging runtime errors. The **getStackTrace()** method provides programmatic access to the stack trace information printed by printStackTrace().

- **The finally Block**

The **finally** block is always executed when the try block exits. Throwing an exception OR exits normally. This ensures that the finally block is executed even if an unexpected exception occurs. It is useful for more than just exception handling – it allows the programmer to avoid having cleanup code accidentally bypassed by a return, continue, or break. Putting cleanup code in a **finally block** is always a good practice, even when no exceptions are anticipated. When closing a file or otherwise recovering resources, place the code in a finally block to ensure that resource is always recovered.

There are 3 possible scenarios executing a try-catch-finally block:

1) If no exception arises in the try block, the finally block is executed and the next statement after the try-catch-finally statement is executed.
2) If a statement causes an exception in the try block that is caught in a catch block, the rest of the statements in the try block are skipped, the catch block is executed, and the finally block is executed. The next statement after the try-catch-finally statement is executed.
3) If one of the statements causes an exception that is not caught in any catch block, the other statements in the try block are skipped, the finally block is executed, and the exception is passed to the caller of this method.

The finally block will be executed in all cases. In addition, the finally block executes even if there is a return statement prior to reaching the finally block. The catch block may be omitted when the finally clause is used.

- **When to use Exceptions**

A method should throw an exception if the error needs to be handled by its caller. The try block contains the code that is executed in normal circumstances. The catch block contains the code that is executed in exceptional circumstances. Exception handling separates error-handling code from normal programming tasks, thus making programs easier to read and to modify. Be aware, however, that exception handling usually requires more time and resources, because it requires instantiating a new exception object, rolling back the call stack, and propagating the exception through the chain of method calls to search for the handler.

An exception occurs in a method. If you want the exception to be processed by its caller, you should create an exception object and throw it. If you can handle the exception in the method where it occurs, there is no need to throw or use exceptions. In general, common exceptions that may occur in multiple classes in a project are candidates for exception classes. Simple errors that may occur in individual methods are best handled without throwing exceptions. This can be done by using if statements to check for errors. When should you use a try-catch block in the code? Use it when you have to deal with unexpected error conditions. **Do not use a try-catch block to deal with simple, expected situations.**

Java allows an exception handler to **rethrow** the exception if the handler cannot process the exception, or simply wants to let its caller be notified of the exception. For example,

```
try { statements; }
catch (TheException ex) {
    perform operations before exits;
    throw ex;  //rethrow ex to the calling method
}
```

Sometimes, you may need to throw a new exception (with additional information) along with the original exception. This is called **chained exception**. For example, the main method in the code below invokes method1, method1 invokes method2, and method2 throws an exception. This exception is caught in the catch block in method1 and is wrapped in a new exception. The new exception is thrown and caught in the catch block in the main method.

```
public class ChainedExceptionDemo {
    public static void main(String[] args) {
        try {
            method1();
        }
        catch (Exception ex) {
            ex.printStackTrace();
        }
    }

    public static void method1() throws Exception {
        try {
            method2();
        }
        catch (Exception ex) {
            throw new Exception("New info from method1", ex);  // chained exception
        }
    }

    public static void method2() throws Exception {
        throw new Exception("New info from method2");  // throw exception
    }
}
```

Example 1 – What exception type does the following program throw?

```
public class Example1 {
    public static void main(String[] args) {
        System.out.println(1 / 0);
    }
}
```

(a) ArithmeticException
(b) ArrayIndexOutOfBoundsException
(c) StringIndexOutOfBoundsException
(d) ClassCastException
(e) No exception

Example 2

```
public class Example2 {
    public static void main(String[] args) {
        try {
            p();
            System.out.println("After the method call");
        }
        catch (NumberFormatException ex) {
            System.out.println("NumberFormatException");
        }
        catch (RuntimeException ex) {
            System.out.println("RuntimeException");
        }
     }
     static void p() {
          String s = "5.6";
          Integer.parseInt(s); // Cause a NumberFormatException
         int i = 0;
         int y = 2 / i;
         System.out.println("Welcome to Java");
     }
}
```

(a) The program displays NumberFormatException.
(b) The program displays NumberFormatException followed by After the method call.
(c) The program displays NumberFormatException followed by RuntimeException.
(d) The program has a compile error.
(e) The program displays RuntimeException.

Example 3

```
public class Example3 {
    public static void main(String[] args) {
        try {
            System.out.println("Welcome to Java");
            int i = 0;
            int y = 2 / i;
            System.out.println("Welcome to HTML");
        }
        finally {
            System.out.println("The finally clause is executed");
        }
    }
}
```

(a) Welcome to Java, then an error message.
(b) Welcome to Java followed by The finally clause is executed in the next line, then an error message.
(c) The program displays three lines: Welcome to Java, Welcome to HTML, The finally clause is executed, then an error message.
(d) None of the above.