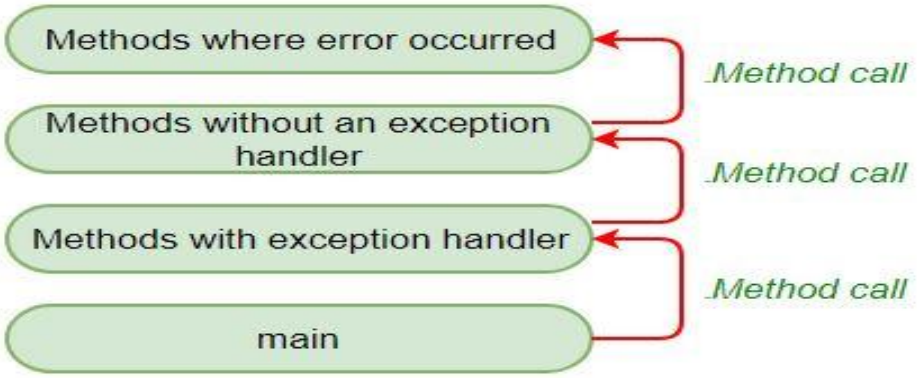
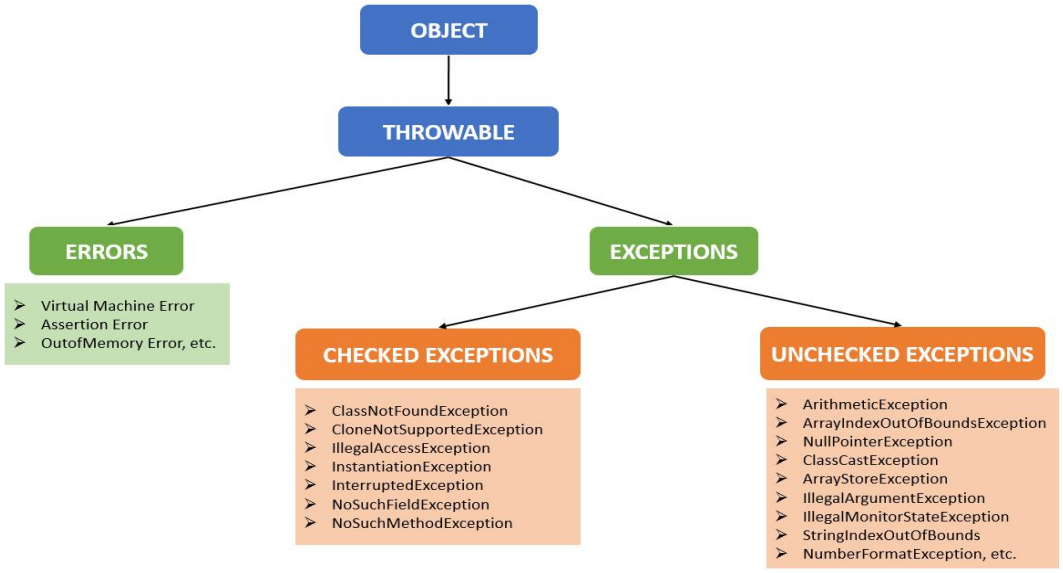
Java Exceptions

An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions. Exceptions can be caught and handled by the program. When an exception occurs it creates an object called the exception object. It contains information about the exception, such as the name and description of the exception and the state of the program when the exception occurred.

After a method throws an exception, the runtime system attempts to find something to handle it. If the exception is not handled in the method from where the exception was thrown it will propagate the exception to the method from which it was called until a handler is present.



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, the program.



There are three kinds of exceptions

The first kind of exception is the ***checked/compile-time*** exception. These are exceptional conditions that a well-written application should anticipate and recover from. For example, suppose an application prompts a user for an input file name, then opens the file by passing the name to the constructor for java.io.FileReader. Normally, the user provides the name of an existing, readable file, so the construction of the FileReader object succeeds, and the execution of the application proceeds normally. But sometimes the user supplies the name of a file which does not exist, and the constructor throws java.io.FileNotFoundException. A well-written program will catch this exception and notify the user of the mistake. Such Exceptions that can occur at compile-time are called checked exceptions since they need to be explicitly checked and handled in code.

The second kind of exception is the ***error***. These are exceptional conditions that are external to the application, and that the application usually cannot anticipate or recover from. For example, suppose that an application successfully opens a file for input, but is unable to read the file because of a hardware or system malfunction. The unsuccessful read will throw java.io.IOError. Or if there is a recursive call which no base conditions the method will call itself over and over again until it reaches the maximum size of the Java thread stack, at which point it exits with a StackOverflowError
Errors are not subject to the Catch or Specify Requirement. Errors are those exceptions indicated by Error and its subclasses.

The third kind of exception is the ***runtime/unchecked*** exception. These are exceptional conditions that are internal to the application, and that the application usually cannot anticipate or recover from. For example, consider in a program a string has been declared and not initialised yet , if there is any method like length() etc called with this null string during runtime it will throw NullPointerException . Runtime exceptions are ignored at the time of compilation.

An exception is caught using a combination of the try, catch, and finally blocks . A try/catch block is placed around the code that might generate an exception. Finally block are usually used to tackle the closeable resources.

The try Block

The first step in constructing an exception handler is to enclose the code that might throw an exception within a try block.
In general, a try block looks like :

```
try {
    code
}
catch and finally blocks . . .
```

When an exception occurs, that exception occurred is handled by catch block associated with it. Every try block should be immediately followed either by a catch block or finally block.

The catch Block

You associate exception handlers with a try block by providing one or more catch blocks directly after the try block.

```
try {

} catch (ExceptionType name) {

} catch (ExceptionType name) {

}
```

From Java SE 7 and later, a single catch block can handle more than one type of exception.

```
catch (ExceptionType | ExceptionType ex) {

}
```

The finally Block

This block always executes when the try block exits. This ensures that the finally block is executed even if an unexpected exception occurs.
But finally is useful for more than just exception handling — it allows the programmer to avoid having cleanup code .

Note: The finally block may not execute if the JVM exits while the try or catch code is being executed.

The try-with-resources Statement

The try-with-resources statement is a try statement that declares one or more resources. A resource is an object that must be closed after the program is finished with it. Prior to Java SE 7, a finally block was used to ensure that a resource is closed regardless of whether the try statement completes normally or abruptly.

For eg

```
 try (FileReader fr = new FileReader(path);
         BufferedReader br = new BufferedReader(fr)) {
         return br.readLine();
     } catch (IOException ex) {

}
```

throws

throws is a keyword in Java that is used in the signature of a method to indicate that this method might throw one of the listed type exceptions. The caller to these methods has to handle the exception using a try-catch block.

For eg
public void doSomething() throws IOException, IndexOutOfBoundsException { }

To specify that doSomething method can throw two exceptions, add a throws clause to the method declaration for the method. The throws clause comprises the throws keyword followed by a comma-separated list of all the exceptions thrown by that method.

throw

The throw keyword in Java is used to explicitly throw an exception from a method or any block of code.

For eg:
throw new NullPointerException("demo");

After the throw statement is executed the nearest enclosing try block is checked to see if it has a catch statement that matches the type of exception. If it finds a match, controlled is transferred to that statement otherwise next enclosing try block is checked, and so on. If no matching catch is found then the default exception handler will halt the program.

Chained Exceptions

An application often responds to an exception by throwing another exception. In effect, the first exception causes the second exception. It can be very helpful to know when one exception causes another.

The following example shows how to use a chained exception.

```
try {

} catch (IOException e) {
    throw new SampleException("Other IOException", e);
}
```

User-defined Exceptions

You can create your own exceptions in Java. The exceptions you are going to define should

- be a child of Throwable.
- If you want to write a checked exception you need to extend the Exception class.
- If you want to write a runtime exception, you need to extend the RuntimeException class.

Advantages of Exceptions

- Exceptions provide the means to separate the details of what to do when something out of the ordinary happens from the main logic of a program. Exceptions enable you to write the main flow of your code and to deal with the exceptional cases elsewhere
- Another advantage of exceptions is the ability to propagate error reporting up the call stack of methods.
- Because all exceptions thrown within a program are objects, the grouping or categorizing of exceptions is very beneficial.

Summary

- The try block identifies a block of code in which an exception can occur.
- The catch block identifies a block of code, known as an exception handler, that can handle a particular type of exception.
- The finally block identifies a block of code that is guaranteed to execute, and is the right place to close files, recover resources, and otherwise clean up after the code enclosed in the try block. (See try with resources)

The try statement should contain at least one catch block or a finally block and may have multiple catch blocks.The class of the exception object indicates the type of exception thrown. The exception object can contain further information about the error, including an error message.