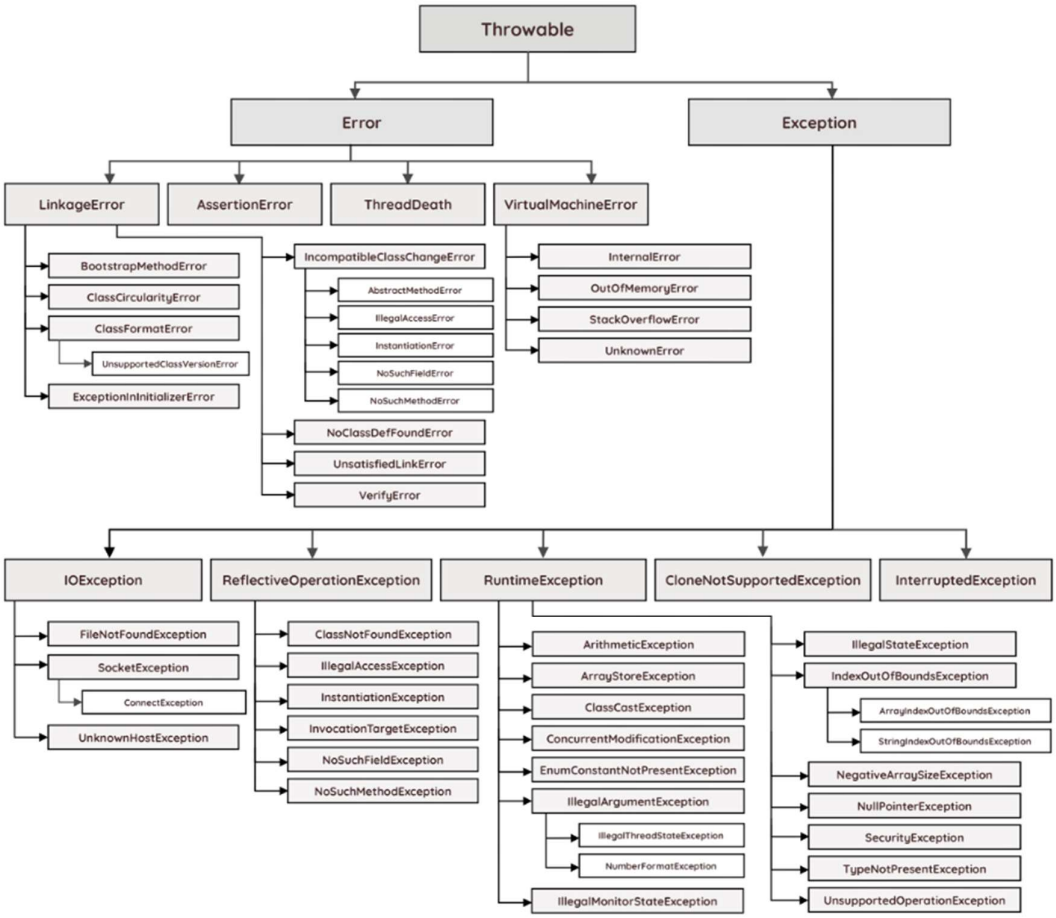
Exception Handling

Exceptions are a mechanism used by many programming languages to describe what to do when something unexpected happens.

Typically, something unexpected is an error of some sort, for example when a method is invoked with unacceptable arguments, or a network connection fails, or the use asks to open a non-existent file.

The Exception Hierarchy



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

```
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
}
```

The Consequences of an Uncaught Exception

Technically, any thrown exceptions must be handled some where within the program. 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.

Using Multiple catch Statements

- 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.

```
//Demonstrate multiple catch statements.
class MultipleCatches {
	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.");
	}
}
```

Catching Subclass Exceptions

There is one important point about multiple catch statements that relates to subclasses.

- A catch clause for a superclass will also match any of its subclasses.
- For example, since the superclass of all exceptions is Throwable, to catch all possible exceptions, catch Throwable.
- If you want to catch exceptions of both a superclass type and a subclass type, put the subclass first in the catch sequence. If you don't, then the superclass catch will also catch all derived classes.
- This rule is self-enforcing because putting the superclass first causes unreachable code to be created, since the subclass catch clause can never execute.
- In Java, unreachable code is an error.

Try Blocks Can Be Nested

- The try statement can be nested. That is, a try statement can be inside another try block.
- If an inner try statement does not have a catch handler for a particular exception, the stack in unwound and the next try statement's catch handlers are inspected for a match.
- This continues until one of the catch statements succeed, or until all the nested try statements are exhausted.
- If no catch statement matches, then the Java run-time system (default handler) will handle the exception.

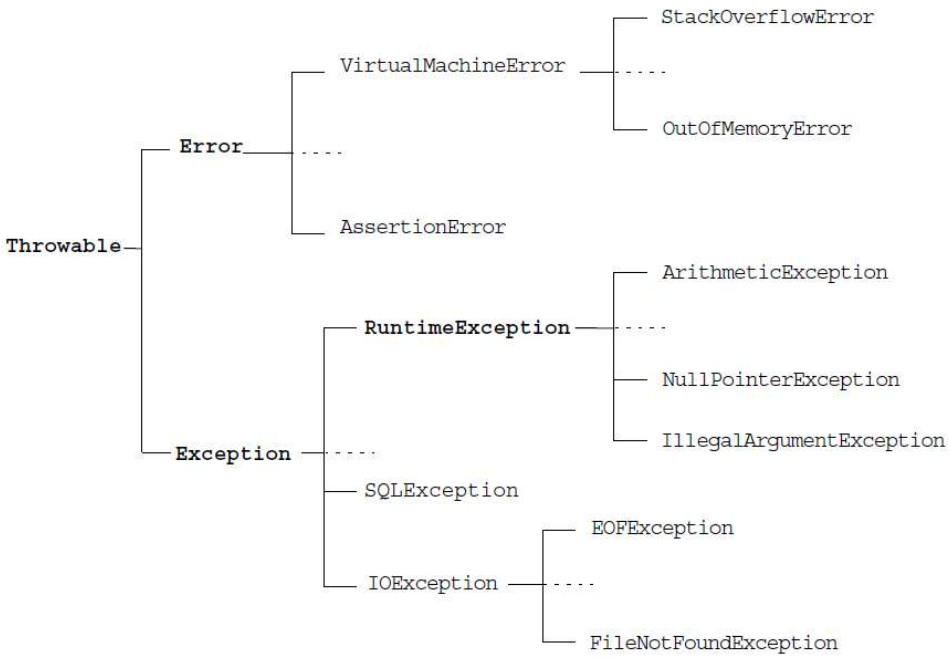
```
//An example of nested try statements.
class NestTry {
	public static void main(String args[]) {
		try {
			int a = args.length;
			/*
			 * If no command-line args are present, the following
statement will generate a
			 * divide-by-zero exception.
			 */
			int b = 42 / a;
			System.out.println("a = " + 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);
		}
	}
}
```

Throwing an Exception

- Before you can catch an exception, some code somewhere must throw one.
- Any code can throw an exception: your code, code from a package written by someone else such as the packages that come with the Java platform, or the Java runtime environment.
- Regardless of what throws the exception, it's always thrown with the throw statement.

A Closer Look at Throwable

- The class java.lang.Throwable acts as the parent class for all objects that can be thrown and caught using the exception-handling mechanisms.
- Methods defined in the Throwable class retrieve the error message associated with the exception and print the stack trace showing where the exception occurred.
- There are three key subclasses of Throwable: Error, RuntimeException and Exception.



Using throw

- To throw an exception explicitly, using the throw statement. The general form of throw is shown here:

  throw *ThrowableInstance*;

- 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.

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

Using throws

- If a method can cause an exception that it does not handle, it must specify this behaviour 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 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);
		}
	}
}
```

Using finally

- 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.
- 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.

```
//Demonstrate finally.
class FinallyDemo {
	//Throw 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");
		}
	}

	//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();
	}
}
```

Three Additional Exception Features

Beginning with JDK 7, three interesting and useful features have been added to the exception system. The first automates the process of releasing a resource, such as a file, when it is no longer needed. The second feature is called multi-catch, and the third is sometimes referred to as final rethrow or more precise rethrow.

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. The try-with-resources statement ensures that each resource is closed at the end of the statement. Any object that implements java.lang.AutoCloseable, which includes all objects which implement java.io.Closeable, can be used as a resource.

The following example reads the first line from a file. It uses an instance of BufferedReader to read data from the file. BufferedReader is a resource that must be closed after the program is finished with it:

```
static String readFirstLineFromFile(String path) throws IOException {
    try (BufferedReader br =
                   new BufferedReader(new FileReader(path))) {
        return br.readLine();
    }
}
```

Handling More Than One Type of Exception

In Java SE 7 and later, a single catch block can handle more than one type of exception. This feature can reduce code duplication and lessen the temptation to catch an overly broad exception.

```
catch (IOException|SQLException ex) {
    logger.log(ex);
    throw ex;
}
```

Rethrowing Exceptions with More Inclusive Type Checking

The Java SE 7 compiler performs more precise analysis of rethrown exceptions than earlier releases of Java SE. This enables you to specify more specific exception types in the throws clause of a method declaration.

Consider the following example:

```
static class FirstException extends Exception {
}

static class SecondException extends Exception {
}

public void rethrowException(String exceptionName) throws Exception {
  try {
    if (exceptionName.equals("First")) {
      throw new FirstException();
    } else {
      throw new SecondException();
    }
  } catch (Exception e) {
    throw e;
  }
}
```

This examples's try block could throw either FirstException or SecondException. Suppose you want to specify these exception types in the throws clause of the rethrowException method declaration. In releases prior to Java SE 7, you cannot do so. Because the exception parameter of the catch clause, e, is type Exception, and the catch block rethrows the exception parameter e, you can only specify the exception type Exception in the throws clause of the rethrowException method declaration.

However, in Java SE 7, you can specify the exception types FirstException and SecondException in the throws clause in the rethrowException method declaration. The Java SE 7 compiler can determine that the exception thrown by the statement throw e must have come from the try block, and the only exceptions thrown by the try block can be FirstException and SecondException. Even though the exception parameter of the catch clause, e, is type Exception, the compiler can determine that it is an instance of either FirstException or SecondException:

```
public void rethrowException(String exceptionName)
               throws FirstException, SecondException {
                 try {
                   // ...
                 }
                 catch (Exception e) {
                   throw e;
                 }
               }
```

Java's Built-in Exceptions

- Inside the standard package java.lang, Java defines several exception classes.
- There are two broad categories of exceptions, known as checked and unchecked exceptions.
- The most general of these unchecked exceptions are subclasses of the standard type RuntimeException. 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.
- Checked exceptions are those that the programmer is expected to handle in the program, and that arise from external conditions that can readily occur in a working program. Examples would be a requested file not being found or a network failure.

Creating Exception Subclasses

- This is quite easy to do: just define a subclass of Exception.
- Your subclasses don't need to implement anything - it is their existence in the type of system that allows you to use them as exceptions.

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