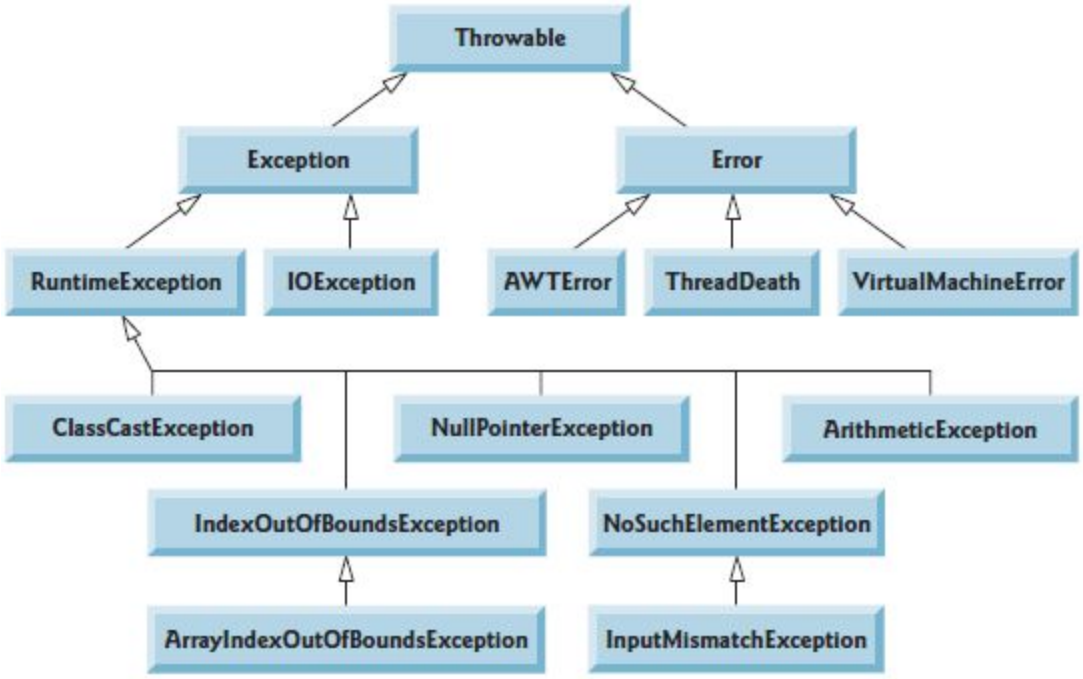
Exception Hierarchy



Example: 1 Divided By Zero Exception

```
// Integer division without exception handling.
import java.util.Scanner;

public class DivideByZeroNoExceptionHandling
{
   // demonstrates throwing an exception when a divide-by-zero occurs
   public static int quotient(int numerator, int denominator)
   {
      return numerator / denominator; // possible division by zero
   }

   public static void main(String[] args)
   {
      Scanner scanner = new Scanner(System.in);

      System.out.print("Please enter an integer numerator: ");
      int numerator = scanner.nextInt();
      System.out.print("Please enter an integer denominator: ");
```

```
      int denominator = scanner.nextInt();

      int result = quotient(numerator, denominator);
      System.out.printf(
         "%nResult: %d / %d = %d%n", numerator, denominator, result);
   }
} // end class DivideByZeroNoExceptionHandling
```

Example: 2 Divided By Zero with Exception Handling

```
// Handling ArithmeticExceptions and InputMismatchExceptions.
import java.util.InputMismatchException;
import java.util.Scanner;

public class DivideByZeroWithExceptionHandling
{
   // demonstrates throwing an exception when a divide-by-zero occurs
   public static int quotient(int numerator, int denominator)
      throws ArithmeticException
   {
      return numerator / denominator; // possible division by zero
   }

   public static void main(String[] args)
   {
      Scanner scanner = new Scanner(System.in);
      boolean continueLoop = true; // determines if more input is needed
```

```
      do
      {
         try // read two numbers and calculate quotient
         {
            System.out.print("Please enter an integer numerator: ");
            int numerator = scanner.nextInt();
            System.out.print("Please enter an integer denominator: ");
            int denominator = scanner.nextInt();

            int result = quotient(numerator, denominator);
            System.out.printf("%nResult: %d / %d = %d%n", numerator,
               denominator, result);
            continueLoop = false; // input successful; end looping
         }
         catch (InputMismatchException inputMismatchException)
         {
            System.err.printf("%nException: %s%n",
               inputMismatchException);
            scanner.nextLine(); // discard input so user can try again
            System.out.printf(
               "You must enter integers. Please try again.%n%n");
         }
         catch (ArithmeticException arithmeticException)
         {
            System.err.printf("%nException: %s%n", arithmeticException);
            System.out.printf(
               "Zero is an invalid denominator. Please try again.%n%n");
         }
      } while (continueLoop);
   }
} // end class DivideByZeroWithExceptionHandling
```

Example: 3 Using try catch and finally

```
/ Fig. 11.5: UsingExceptions.java
// try...catch...finally exception handling mechanism.

public class UsingExceptions
{
   public static void main(String[] args)
```

```
{
   try
   {
      throwException();
   }
   catch (Exception exception) // exception thrown by throwException
   {
      System.err.println("Exception handled in main");
   }

   doesNotThrowException();
}

// demonstrate try...catch...finally
public static void throwException() throws Exception
{
   try // throw an exception and immediately catch it
   {
      System.out.println("Method throwException");
      throw new Exception(); // generate exception
   }
   catch (Exception exception) // catch exception thrown in try
   {
      System.err.println(
         "Exception handled in method throwException");
      throw exception; // rethrow for further processing

      // code here would not be reached; would cause compilation errors

   }
   finally // executes regardless of what occurs in try...catch
   {
      System.err.println("Finally executed in throwException");
   }

   // code here would not be reached; would cause compilation errors

}

// demonstrate finally when no exception occurs
public static void doesNotThrowException()
{
   try // try block does not throw an exception
   {
      System.out.println("Method doesNotThrowException");
```

```
      }
      catch (Exception exception) // does not execute
      {
         System.err.println(exception);
      }
      finally // executes regardless of what occurs in try...catch
      {
         System.err.println(
            "Finally executed in doesNotThrowException");
      }

      System.out.println("End of method doesNotThrowException");
   }
} // end class UsingExceptions
```

Example: 4 Obtaining data from exception object

```
// Fig. 11.6: UsingExceptions.java
// Stack unwinding and obtaining data from an exception object.

public class UsingExceptions
{
   public static void main(String[] args)
   {
      try
      {
         method1();
      }
      catch (Exception exception) // catch exception thrown in method1
      {
         System.err.printf("%s%n%n", exception.getMessage());
         exception.printStackTrace();

         // obtain the stack-trace information
         StackTraceElement[] traceElements = exception.getStackTrace();
```

```
            System.out.printf("%nStack trace from getStackTrace:%n");
            System.out.println("Class\t\tFile\t\t\tLine\tMethod");

            // loop through traceElements to get exception description
            for (StackTraceElement element : traceElements)
            {
               System.out.printf("%s\t", element.getClassName());
               System.out.printf("%s\t", element.getFileName());
               System.out.printf("%s\t", element.getLineNumber());
               System.out.printf("%s%n", element.getMethodName());
            }
         }
      } // end main

      // call method2; throw exceptions back to main
      public static void method1() throws Exception
      {
         method2();
      }

      // call method3; throw exceptions back to method1
      public static void method2() throws Exception
      {
         method3();
      }

      // throw Exception back to method2
      public static void method3() throws Exception
      {
         throw new Exception("Exception thrown in method3");
      }
   } // end class UsingExceptions
```

Chained Exceptions

Sometimes a method responds to an exception by throwing a different exception type that's specific to the current application. If a **catch** block throws a new exception, the original exception's information and stack trace are lost. Earlier Java versions provided no mechanism to wrap the original exception information with the new exception's information to provide a complete stack trace showing where the original problem occurred. This made debugging such problems particularly difficult. Chained exceptions enable an exception object to maintain the complete stack-trace information from the original exception. Following example demonstrates chained exceptions.

Example: 5 Using chained exceptions

```
// Chained exceptions.

public class UsingChainedExceptions
{
   public static void main(String[] args)
   {
      try
      {
         method1();
      }
      catch (Exception exception) // exceptions thrown from method1
      {
         exception.printStackTrace();
      }
   }

   // call method2; throw exceptions back to main
   public static void method1() throws Exception
   {
      try
      {
         method2();
      }
      catch (Exception exception) // exception thrown from method2
      {
         throw new Exception("Exception thrown in method1", exception);
      }
   } // end method method1
```

```
   // call method3; throw exceptions back to method1
   public static void method2() throws Exception
   {
      try
      {
         method3();
      }
      catch (Exception exception) // exception thrown from method3
      {
         throw new Exception("Exception thrown in method2", exception);
      }
   } // end method method2

   // throw Exception back to method2
   public static void method3() throws Exception
   {
      throw new Exception("Exception thrown in method3");
   }
} // end class UsingChainedExceptions
```

Assertions

When implementing and debugging a class, it's sometimes useful to state conditions that should be true at a particular point in a method. These conditions, called assertions, help ensure a program's validity by catching potential bugs and identifying possible logic errors during development. Preconditions and postconditions are two types of assertions. Preconditions are assertions about a program's state when a method is invoked, and postconditions are assertions about its state after a method finishes.

While assertions can be stated as comments to guide you during program development, Java includes two versions of the **assert** statement for validating assertions programatically. The **assert** statement evaluates a boolean expression and, if false, throws an AssertionError (a subclass of Error). The first form of the **assert** statement is

assert *expression;*

assert *expression1 : expression2;*

which evaluates **expression1** and throws an **AssertionError** with **expression2** as the error message if **expression1** is **false**,

You use assertions primarily for debugging and identifying logic errors in an application.

You must explicitly enable assertions when executing a program, because they reduce performance and are unnecessary for the program's user. To do so, use the java command's **-ea** command-line option, as in

java -ea AssertTest

Example: 5 Using Assertions

```
// Checking with assert that a value is within range.
import java.util.Scanner;

public class AssertTest
{
   public static void main(String[] args)
   {
      Scanner input = new Scanner(System.in);

      System.out.print("Enter a number between 0 and 10: ");
      int number = input.nextInt();

      // assert that the value is >= 0 and <= 10
      assert (number >= 0 && number <= 10) : "bad number: " + number;

      System.out.printf("You entered %d%n", number);
   }
} // end class AssertTest
```