Exceptions - the throw statement
If we attempt to invoke a method on a pointer that is
null, a NullPointerException is thrown and the
program (generally) aborts.

```java
Rectangle r = null;
r.name();
```
What happens is that an exception with the value
NullPointerException is thrown.

You can throw exceptions with the throw statement:
```java
throw e;
```
The object that is "thrown" must be a subclass of the
class Throwable.

Exceptions - the try statement
Exceptions are "caught" with a catch clause of the try
statement, for example:
```java
try {
    statements;
    ...
} catch (NullPointerException e) {
    statements;
    ...
} catch (ArrayIndexOutOfBoundsException e) {
    statements;
    ...
} finally {
    statements;
    ....
}
```
All exceptions that can be thrown by a method that
are not a subclass of RuntimeException and Error
must be declared in the method header:
```java
public void aMethod() throws AnException {
}
```

Execution of the try statement
The statements inside the try block are executed.
If the try block finishes normally then the finally block,
if any, is executed.
If the try block terminates with a thrown exception
then the catch phrases are examined in sequence for
a catch clause variable type that the thrown exception
can be assigned to.
If such a clause is found then the exception object is
assigned to the variable and the clause is executed
after which the finally clause statements are executed.
If an exception is thrown from the catch clause then it
will be thrown after the finally clause statements are
executed.
If none are found then the finally clause statements
are executed and the exception is continued to be
thrown.
If an exception is waiting to be thrown after the finally
clause statements are executed and one of these
statements throws an exception then the previous
exception is forgotten and the new exception is
thrown.

Exceptions
An exception is an event that occurs during the
normal flow of execution of a program that disrupts
the normal flow of execution.

For example, array index out of bounds, end-of-file, null
pointer encountered, etc.

In Java, an exception is an object that is "thrown". The
thrown object is passed up the method call stack until it
is "caught" by an exception handler. If the exception
is not caught, the program will terminate when the
exception propagates out of the main method.

All exceptions must be a subclass of class
Throwable. There are two direct subclasses of
Throwable: Exception and Error.

Classes Error, Exception, and
RuntimeException
An Error indicates serious problems that a normal
application should not try to catch, for example,
VirtualMachineError.
An Exception indicates conditions that a normal
application might want to catch, for example,
IllegalArgumentException.

RuntimeException is a direct subclass of Exception.
IndexOutOfBoundsException,
NullPointerException, ArithmeticException,
and many other exceptions are subclasses of
RuntimeException.

Exceptions that are not subclasses of Error or
RuntimeException are called checked exceptions.
All checked exceptions have to be caught or declared
in a method's throw clause.

Exceptions Propagate
The code:
```java
public class ZeroDivide {
    public static int divide( int n, int d ) { return n / d;
    }
    public static int divideSelf( int n ) { int result = divide( n, n ); return result;
    }
    public static void main( String args[] ) {
        System.out.println( divideSelf( 3 ) );
        System.out.println( divideSelf( 0 ) );
    }
}
```
produces the following output:
```
1
Exception in thread "main"
java.lang.ArithmeticException: / by zero
  at ZeroDivide.divide(ZeroDivide.java:3)
  at ZeroDivide.divideSelf(ZeroDivide.java:7)
  at ZeroDivide.main(ZeroDivide.java:13)
```
Handling the exception

We can handle the exception with this code:

```java
public class ZeroDivide {
    public static int divide( int n, int d ) {
        try {
            return n / d;
        } catch (ArithmeticException e) {
            System.out.println("ArithmeticException " + e.getMessage() + " caught.");
            return 0;
        }
    }

    public static int divideSelf( int n ) {
        int result = divide( n, n );
        return result;
    }

    public static void main( String args[] ) {
        System.out.println( divideSelf( 3 ) );
        System.out.println( divideSelf( 0 ) );
    }
}
```

and this produces the following output:

```
1
ArithmeticException / by zero caught.
0
```

Handling the exception in a different method

We can handle the exception in divide:

```java
public class ZeroDivide {
    public static int divide( int n, int d ) {
        return n / d;
    }

    public static int divideSelf( int n ) {
        int result;
        try {
            result = divide( n, n );
        } catch (ArithmeticException e) {
            System.out.println("ArithmeticException " + e.getMessage() + " caught.");
            result = 0;
        }
        return result;
    }

    public static void main( String args[] ) {
        System.out.println( divideSelf( 3 ) );
        System.out.println( divideSelf( 0 ) );
    }
}
```

and this produces this result:

```
1
ArithmeticException / by zero caught.
0
```

The try / catch / finally statement

The try block is wrapped around the code that might cause an exception we want to catch.

The catch blocks following the try block are exception handlers and contain code to handle the exception.

The first handler (or catch block) immediately following the try block whose parameter type is compatible with the thrown exception object will be executed.

If there is no handler, the exception is propagated up the method call stack.

There is an optional finally block that contains code that is always executed when control leaves the try statement.

Using the finally clause

Here is code using the finally clause:

```java
public class ZeroDivide {
    public static int divide( int n, int d ) {
        return n / d;
    }

    public static int divideSelf( int n ) {
        int result;
        try {
            result = divide( n, n );
        } catch (ArithmeticException e) {
            System.out.println("ArithmeticException " + e.getMessage() + " caught.");
            result = 0;
        }
        finally {
            System.out.println("Always executed!!!");
        }
        return result;
    }

    public static void main( String args[] ) {
        System.out.println( divideSelf( 3 ) );
        System.out.println( divideSelf( 0 ) );
    }
}
```

produces the following output:

```
Always executed!!!
1
ArithmeticException / by zero caught.
Always executed!!!
0
```
Multiple catch blocks
Here are multiple catch blocks:

```java
class ZeroDivide {
    public static int divide(Integer n, Integer d) {
        int result = 0;
        try {
            result = n.intValue() / d.intValue();
        } catch (ArithmeticException e) {
            System.out.println("ArithmeticException "+ e.getMessage() + " caught.");
        } catch (NullPointerException e) {
            System.out.println("NullPointerException "+ e.getMessage() + " caught.");
        }
        return result;
    }
    public static int divideSelf(Integer n) {
        int result = divide(n, n);
        return result;
    }
    public static void main(String[] args) {
        System.out.println(divideSelf(new Integer(3)));
        System.out.println(divideSelf(new Integer(0)));
        System.out.println(divideSelf(null));
    }
}
```

and this produces this output:

```
1
ArithmeticException / by zero caught.
0
NullPointerException null caught.
0
```

Using exception superclasses
If you want to handle all exceptions in the same way, one handler will do the job:

```java
class ZeroDivide {
    public static int divide(Integer n, Integer d) {
        int result = 0;
        try {
            result = n.intValue() / d.intValue();
        } catch (Exception e) {
            System.out.println("Exception "+ e.getMessage() + " caught.");
        }
        return result;
    }
    public static int divideSelf(Integer n) {
        int result = divide(n, n);
        return result;
    }
    public static void main(String[] args) {
        System.out.println(divideSelf(new Integer(3)));
        System.out.println(divideSelf(new Integer(0)));
        System.out.println(divideSelf(null));
    }
}
```

and this produces:

```
1
Exception / by zero caught.
0
Exception null caught.
0
```

Throwing exceptions
Any Java method can throw any of the predefined exceptions. For example

```java
class ZeroDivide {
    public static int divide(int n, int d) {
        if (n % d != 0) {
            throw new ArithmeticException("non-integer result");
        }
        return n / d;
    }
}
```

Creating your own exceptions
You can create your own exception types:

```java
class MeasurementException extends Exception {
    public MeasurementException(String message) {
        super("Measurement error: " + message);
    }
}
```

```java
class Measurement {
    private double inches;
    public Measurement {
        /**
         * Creates a measurement set to a specified number of inches
         * @param newInches measurement in inches
         * @exception MeasurementException if negative
         */
        public Measurement(double newInches) throws MeasurementException {
            if (newInches < 0) {
                throw new MeasurementException("Negative inches in Measurement constructor");
            }
            inches = newInches;
        }
    }
    public static void main(String[] args) {
        throws MeasurementException {
            System.out.println(new Measurement(5));
            System.out.println(new Measurement(-5));
        }
    }
```

Creating your own exceptions
this code outputs:

```
Exception in thread "main" MeasurementException: Measurement error: Negative inches in Measurement constructor
at Measurement.<init>(Measurement.java:22)
at Measurement.main(Measurement.java:31)
```

The throws clause in method declarations
If a method can throw a checked exception that it does not catch it must declare the exception in the throws clause

```java
class ZeroDivide {
    public void doit(int arg) throws MeasurementException {
        Statements
        ...
    }
```