Exceptions

An **exception** is a problem that occurs during program execution. Exceptions cause abnormal termination of the program.
Exception handling is a powerful mechanism that handles runtime errors to maintain normal application flow.

An exception can occur for many different reasons. Some examples:
- A user has entered invalid data.
- A file that needs to be opened cannot be found.
- A network connection has been lost in the middle of communications.
- Insufficient memory and other issues related to physical resources.

As you can see, exceptions are caused by user error, programmer error, or physical resource issues.
However, a well-written program should handle all possible exceptions.

Exception Handling

Exceptions can be caught using a combination of the **try** and **catch** keywords.
A try/catch block is placed around the code that might generate an exception.

Syntax:

```
try {
//some code
} catch (Exception e) {
//some code to handle errors
}
```

A **catch** statement involves declaring the type of exception you are trying to catch. If an exception occurs in the **try** block, the **catch** block that follows the **try** is checked. If the type of exception that occurred is listed in a **catch** block, the exception is passed to the **catch** block much as an argument is passed into a method parameter.
The **Exception** type can be used to catch all possible exceptions.

The example below demonstrates exception handling when trying to access an array index that does not exist:

```
public class MyClass {
   public static void main(String[ ] args) {
      try {
         int a[ ] = new int[2];
         System.out.println(a[5]);
      } catch (Exception e) {
         System.out.println("An error occurred");
      }
   }
}
```

Output: An error occurred
Without the **try/catch** block this code should crash the program, as a[5] does not exist.
Notice the (**Exception e)** statement in the **catch** block - it is used to catch all possible Exceptions.

throw

The **throw** keyword allows you to manually generate exceptions from your methods. Some of the numerous available exception types include the IndexOutOfBoundsException, IllegalArgumentException, ArithmeticException, and so on.
For example, we can throw an ArithmeticException in our method when the parameter is 0.

```
public class Program {

   static int div(int a, int b) throws ArithmeticException {
      if(b == 0) {
         throw new ArithmeticException("Division by Zero");
      } else {
         return a / b;
      }
   }

   public static void main(String[] args) {
      System.out.println(div(42, 0));
   }

}
```

Output:

```
Exception in thread "main" java.lang.ArithmeticException: Division by Zero
        at Program.div(Program.java:6)
        at Program.main(Program.java:13)
```

The **throws** statement in the method definition defines the type of Exception(s) the method can throw.
Next, the **throw** keyword throws the corresponding exception, along with a custom message.
If we call the **div** method with the second parameter equal to 0, it will throw an ArithmeticException with the message "Division by Zero".
Multiple exceptions can be defined in the throws statement using a **comma-separated** list.

Exception Handling

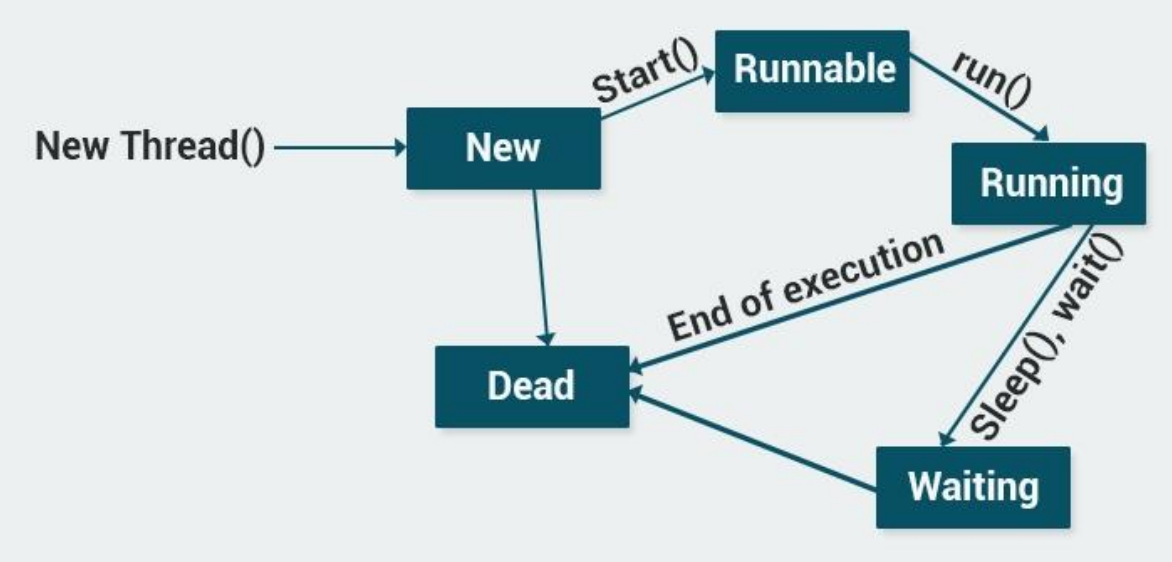
A single try block can contain multiple catch blocks that handle different exceptions separately.
Example:

```
try {
//some code
} catch (ExceptionType1 e1) {
//Catch block
} catch (ExceptionType2 e2) {
//Catch block
} catch (ExceptionType3 e3) {
//Catch block
}
```

All catch blocks should be ordered from most specific to most general.
Following the specific exceptions, you can use the **Exception** type to handle all other exceptions as the last catch.

Threads

Java is a **multi-threaded** programming language. This means that our program can make optimal use of available resources by running two or more components concurrently, with each component handling a different task.
You can subdivide specific operations within a single application into individual **threads** that all run in parallel.
The following diagram shows the life-cycle of a thread.



There are two ways to create a thread.
1. Extend the Thread class
Inherit from the **Thread** class, override its **run**() method, and write the functionality of the thread in the **run**() method.
Then you create a new object of your class and call it's **start** method to run the thread.
Example:

```
class Loader extends Thread {
public void run() {
System.out.println("Hello");
}
}

class MyClass {
public static void main(String[ ] args) {
Loader obj = new Loader();
obj.start();
}
}
```

Try it Yourself

```
class Loader extends Thread {
   public void run() {
      System.out.println("Hello");
   }
}

class MyClass {
   public static void main(String[ ] args) {
      Loader obj = new Loader();
```

```
        obj.start();
    }
}
```
Output:
Hello

As you can see, our Loader class extends the Thread class and overrides its **run()** method.
When we create the **obj** object and call its **start**() method, the **run**() method statements execute on a different thread.
Every Java thread is prioritized to help the operating system determine the order in which to schedule threads. The priorities range from 1 to 10, with each thread defaulting to priority 5. You can set the thread priority with the **setPriority**() method.

Threads

The other way of creating Threads is **implementing the Runnable interface**.
Implement **the** run() method. Then, create a new Thread object, pass the Runnable class to its constructor, and start the Thread by calling the **start**() method.
Example:

```
class Loader implements Runnable {
public void run() {
System.out.println("Hello");
}
}

class MyClass {
public static void main(String[ ] args) {
Thread t = new Thread(new Loader());
t.start();
}
}
```

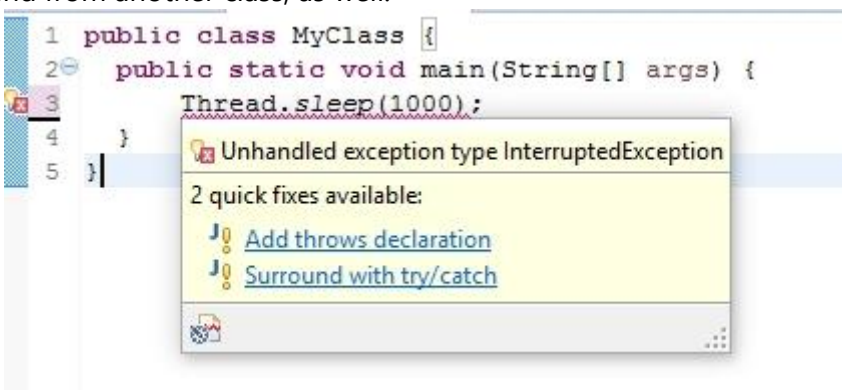
Try it Yourself

```
class Loader implements Runnable {
    public void run() {
        System.out.println("Hello");
    }
}

class MyClass {
    public static void main(String[ ] args) {
        Thread t = new Thread(new Loader());
        t.start();
    }
}
```
Output:
Hello

The **Thread.sleep()** method pauses a Thread for a specified period of time. For example, calling **Thread.sleep(1000);** pauses the thread for one second. Keep in mind that **Thread.sleep**() throws an InterruptedException, so be sure to surround it with a **try/catch** block.

It may seem that implementing the Runnable interface is a bit more complex than extending from the Thread class. However, implementing the Runnable interface is the preferred way to start a Thread, because it enables you to extend from another class, as well.




```
public class MyClass {
public static void main(String[ ] args) {
try {
Thread.sleep(1000);
} catch (InterruptedException e) {
//some code
}
}
}
```

We have seen examples of **unchecked** exceptions, which are checked at runtime, in previous lessons.
Example (when attempting to divide by 0):

```
public class MyClass {
    public static void main(String[ ] args) {
        int value = 7;
        value = value / 0;
    }
}
```

Output:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at MyClass.main(MyClass.java:5)
```

It is good to know the Types of Exceptions because they can help you debug your code faster.