Java Exception Handling & Thread

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");
    }
  }
}
//Outputs "An error occurred"
```

Try It Yourself

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.

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

Try It Yourself

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

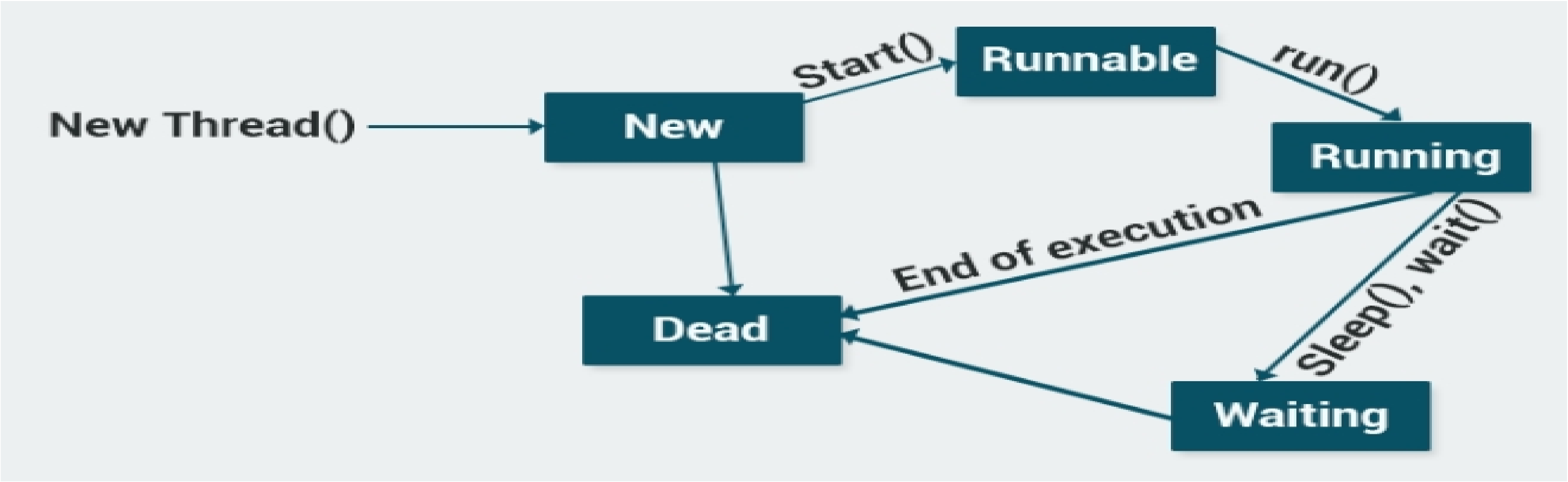
A single try block can contain multiple catch blocks that handle different exceptions separately.
Example:

```
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.

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

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();
  }
}
```

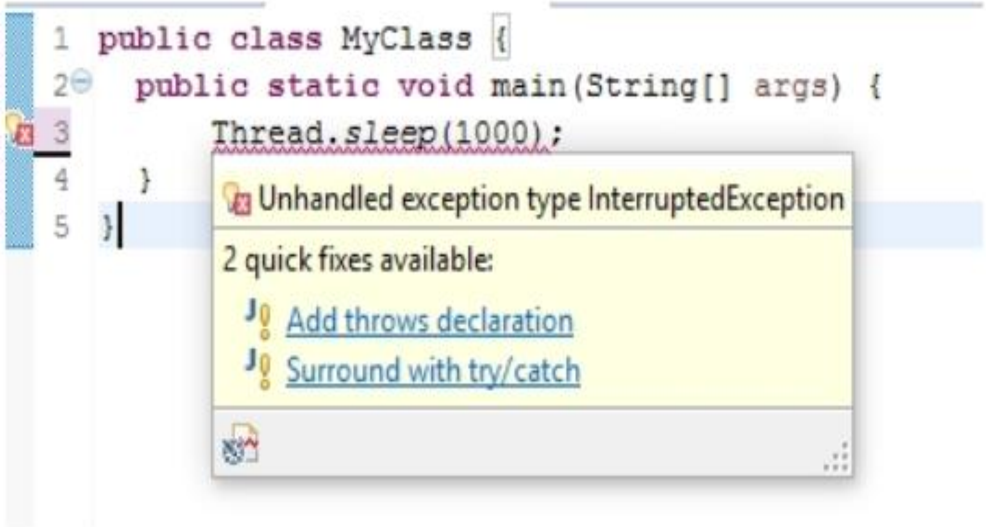
Try It Yourself

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.

Types of Exceptions

There are two exception types, **checked** and **unchecked** (also called runtime). The main difference is that checked exceptions are checked when compiled, while unchecked exceptions are checked at runtime.
As mentioned in our previous lesson, Thread.sleep() throws an InterruptedException. This is an example of a **checked** exception. Your code will not compile until you've handled the exception.




```
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;
  }
}
/*
Exception in thread "main" java.lang.ArithmeticException: / by zero
  at MyClass.main(MyClass.java:4)
*/
```

Try It Yourself

ArrayList

The Java API provides special classes to store and manipulate groups of objects.
One such class is the **ArrayList**. Standard Java arrays are of a fixed length, which means that after they are created, they cannot expand or shrink.
On the other hand, **ArrayLists** are created with an initial size, but when this size is exceeded, the collection is automatically enlarged.
When objects are removed, the ArrayList may shrink in size. Note that the ArrayList class is in the **java.util** package, so it's necessary to import it before using it.
Create an ArrayList as you would any object.

```
import java.util.ArrayList;
//...
ArrayList colors = new ArrayList();
```

You can optionally specify a **capacity** and **type** of objects the ArrayList will hold:

```
ArrayList<String> colors = new ArrayList<String>(10);
```

The code above defines an ArrayList of Strings with 10 as its initial size.

> ArrayLists store objects. Thus, the type specified must be a class type. You cannot pass, for example, **int** as the objects' type. Instead, use the special **class types** that correspond to the desired value type, such as **Integer** for int, **Double** for double, and so on.

ArrayList

The ArrayList class provides a number of useful methods for manipulating its objects.
The **add()** method adds new objects to the ArrayList. Conversely, the **remove()** methods remove objects from the ArrayList.
Example:

```
import java.util.ArrayList;

public class MyClass {
  public static void main(String[ ] args) {
    ArrayList<String> colors = new ArrayList<String>();
    colors.add("Red");
    colors.add("Blue");
    colors.add("Green");
    colors.add("Orange");
    colors.remove("Green");

    System.out.println(colors);
  }
}
// Output: [Red, Blue, Orange]
```

Try It Yourself

Other useful methods include the following:
- **contains()**: Returns true if the list contains the specified element
- **get**(int index): Returns the element at the specified position in the list
- **size()**: Returns the number of elements in the list
- **clear()**: Removes all of the elements from the list

Note: As with arrays, the indexing starts with **0**.

LinkedList

The LinkedList is very similar in syntax to the ArrayList.
You can easily change an ArrayList to a LinkedList by changing the object type.

```
import java.util.LinkedList;

public class MyClass {
  public static void main(String[ ] args) {
    LinkedList<String> c = new LinkedList<String>();
    c.add("Red");
    c.add("Blue");
    c.add("Green");
    c.add("Orange");
    c.remove("Green");
    System.out.println(c);
  }
}
// Outputs [Red, Blue, Orange]
```

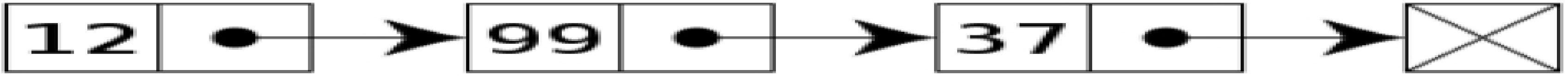
Try It Yourself

You cannot specify an initial capacity for the LinkedList.

LinkedList vs. ArrayList

The most notable difference between the LinkedList and the ArrayList is in the way they store objects.
The ArrayList is better for **storing** and **accessing** data, as it is very similar to a normal array.
The LinkedList is better for **manipulating** data, such as making numerous inserts and deletes.

In addition to storing the object, the LinkedList stores the memory address (or link) of the element that follows it. It's called a LinkedList because each element contains a link to the neighboring element.



You can use the enhanced for loop to iterate over its elements.

```
LinkedList<String> c = new LinkedList<String>();
c.add("Red");
c.add("Blue");
c.add("Green");
c.add("Orange");
c.remove("Green");

for(String s: c) {
  System.out.println(s);
}
/* Output:
Red
Blue
Orange
 */
```

Summary:
- Use an **ArrayList** when you need rapid access to your data.
- Use a **LinkedList** when you need to make a large number of inserts and/or deletes.

HashMap

Arrays and Lists store elements as ordered collections, with each element given an integer index.
HashMap is used for storing data collections as key and value pairs. One object is used as a key (index) to another object (the value).
The **put**, **remove**, and **get** methods are used to add, delete, and access values in the HashMap.
Example:

```
import java.util.HashMap;
public class MyClass {
  public static void main(String[ ] args) {
    HashMap<String, Integer> points = new HashMap<String, Integer>();
    points.put("Amy", 154);
    points.put("Dave", 42);
    points.put("Rob", 733);
    System.out.println(points.get("Dave"));
  }
}
// Outputs 42
```

Try It Yourself

We have created a HashMap with Strings as its keys and Integers as its values.

> Use the **get** method and the corresponding key to access the HashMap elements.

HashMap

A **HashMap** cannot contain duplicate keys. Adding a new item with a key that already exists overwrites the old element.
The HashMap class provides **containsKey** and **containsValue** methods that determine the presence of a specified key or value.
If you try to get a value that is not present in your map, it returns the value of **null**.

> **null** is a special type that represents the absence of a value.

Sets

A **Set** is a collection that cannot contain duplicate elements. It models the mathematical set abstraction.
One of the implementations of the Set is the **HashSet** class.
Example:

```
import java.util.HashSet;

public class MyClass {
  public static void main(String[ ] args) {
    HashSet<String> set = new HashSet<String>();
    set.add("A");
    set.add("B");
    set.add("C");
    System.out.println(set);
  }
}
//  Output: [A, B, C]
```

Try It Yourself

You can use the **size()** method to get the number of elements in the HashSet.

Sorting Lists

For the manipulation of data in different collection types, the Java API provides a **Collections** class, which is included in the java.util package.
One of the most popular **Collections** class methods is **sort()**, which sorts the elements of your collection type. The methods in the **Collections** class are static, so you don't need a Collections object to call them.
Example:

```
public class MyClass {
  public static void main(String[ ] args) {
    ArrayList<String> animals = new ArrayList<String>();
    animals.add("tiger");
    animals.add("cat");
    animals.add("snake");
    animals.add("dog");

    Collections.sort(animals);

    System.out.println(animals);
  }
}
/* Outputs:
[cat, dog, snake, tiger]
*/
```

Try It Yourself

As you can see, the elements have been sorted alphabetically.

You can call the **sort()** methods on different types of Lists, such as Integers.

```
import java.util.ArrayList;
import java.util.Collections;

public class MyClass {
  public static void main(String[ ] args) {
    ArrayList<Integer> nums = new ArrayList<Integer>();
    nums.add(3);
    nums.add(36);
    nums.add(73);
    nums.add(40);
    nums.add(1);

    Collections.sort(nums);
    System.out.println(nums);
  }
}
/* Outputs:
[1, 3, 36, 40, 73]
*/
```

Try It Yourself

Other useful methods in the **Collections** class:
max(Collection c): Returns the maximum element in c as determined by natural ordering.
min(Collection c): Returns the minimum element in c as determined by natural ordering.
reverse(List list): Reverses the sequence in list.
shuffle(List list): Shuffles (i.e., randomizes) the elements in list.

Iterators

An Iterator is an object that enables to cycle through a collection, obtain or remove elements. Before you can access a collection through an iterator, you must obtain one. Each of the collection classes provides an **iterator()** method that returns an iterator to the start of the collection. By using this iterator object, you can access each element in the collection, one element at a time.

The Iterator class provides the following methods:
hasNext(): Returns true if there is at least one more element; otherwise, it returns false.
next(): Returns the next object and advances the iterator.
remove(): Removes the last object that was returned by next from the collection.

The Iterator class must be imported from the **java.util** package.
Example:

```
import java.util.Iterator;
import java.util.LinkedList;

public class MyClass {
  public static void main(String[ ] args) {
  LinkedList<String> animals = new LinkedList<String>();
  animals.add("fox");
  animals.add("cat");
  animals.add("dog");
  animals.add("rabbit");

  Iterator<String> it = animals.iterator();
  String value = it.next();
  System.out.println(value);
}
}
//Outputs "fox"
```

it.next() returns the first element in the list and then moves the iterator on to the next element.
Each time you call **it.next()**, the iterator moves to the next element of the list.

Iterators

Typically, **iterators** are used in loops. At each iteration of the loop, you can access the corresponding list element.
Example:

```
import java.util.Iterator;
import java.util.LinkedList;

public class MyClass {
  public static void main(String[ ] args) {
    LinkedList<String> animals = new LinkedList<String>();
    animals.add("fox");
    animals.add("cat");
    animals.add("dog");
    animals.add("rabbit");

    Iterator<String> it = animals.iterator();
    while(it.hasNext()) {
      String value = it.next();
      System.out.println(value);
     }
  }
}
/*
fox
cat
dog
rabbit
*/
```

Here, the **while** loop determines whether the iterator has additional elements, prints the value of the element, and advances the iterator to the next.

Working with Files

The **java.io** package includes a **File** class that allows you to work with files.
To start, create a **File** object and specify the path of the file in the constructor.

```
import java.io.File;
...
File file = new File("C:\\data\\input-file.txt");
```

With the **exists()** method, you can determine whether a file exists.

```
import java.io.File;

public class MyClass {
  public static void main(String[ ] args) {
    File x = new File("C:\\sololearn\\test.txt");
    if(x.exists()) {
     System.out.println(x.getName() +  "exists!");
    }
    else {
     System.out.println("The file does not exist");
    }
  }
}
```

The code above prints a message stating whether or not the file exists at the specified path.

> The **getName()** method returns the name of the file.
> Note that we used double backslashes in the path, as one backslash should be escaped in the path String.

Reading a File

Files are useful for storing and retrieving data, and there are a number of ways to read from a files.
One of the simplest ways is to use the **Scanner** class from the **java.util** package.
The constructor of the **Scanner** class can take a **File** object as input.
To read the contents of a text file at the path "C:\\sololearn\\test.txt", we would need to create a File object with the corresponding path and pass it to the Scanner object.

```
try {
  File x = new File("C:\\sololearn\\test.txt");
  Scanner sc = new Scanner(x);
}
 catch (FileNotFoundException e) {

}
```

We surrounded the code with a try/catch block, because there's a chance that the file may not exist.

Reading a File

The **Scanner** class inherits from the **Iterator**, so it behaves like one.
We can use the Scanner object's **next()** method to read the file's contents.

```
try {
  File x = new File("C:\\sololearn\\test.txt");
  Scanner sc = new Scanner(x);
  while(sc.hasNext()) {
   System.out.println(sc.next());
  }
  sc.close();
} catch (FileNotFoundException e) {
  System.out.println("Error");
}
```

The file's contents are output word by word, because the **next()** method returns each word separately.

> It is always good practice to close a file when finished working with it. One way to do this is to use the Scanner's **close()** method.

Creating Files

Formatter, another useful class in the java.util package, is used to create content and write it to files.
Example:

```
import java.util.Formatter;

public class MyClass {
  public static void main(String[ ] args) {
  try {
    Formatter f = new Formatter("C:\\sololearn\\test.txt");
  } catch (Exception e) {
     System.out.println("Error");
  }
  }
}
```

This creates an empty file at the specified path. If the file already exists, this will overwrite it.

> Again, you need to surround the code with a **try/catch** block, as the operation can fail.

Writing to Files

Once the file is created, you can write content to it using the same Formatter object's **format()** method.
Example:

```
import java.util.Formatter;

public class MyClass {
  public static void main(String[ ] args) {
  try {
    Formatter f = new Formatter("C:\\sololearn\\test.txt");
    f.format("%s %s %s", "1","John", "Smith \r\n");
    f.format("%s %s %s", "2","Amy", "Brown");
    f.close();
  } catch (Exception e) {
    System.out.println("Error");
  }
  }
}
```

Try It Yourself

The **format()** method formats its parameters according to its first parameter.
%s mean a string and get's replaced by the first parameter after the format. The second %s get's replaced by the next one, and so on. So, the format **%s %s %s** denotes three strings that are separated with spaces.
Note: **\r\n** is the newline symbol in Windows.
The code above creates a file with the following content:

```
1 John Smith
2 Amy Brown
```

Don't forget to **close** the file once you're finished writing to it!