



Talent Battle

Java Exception Handling

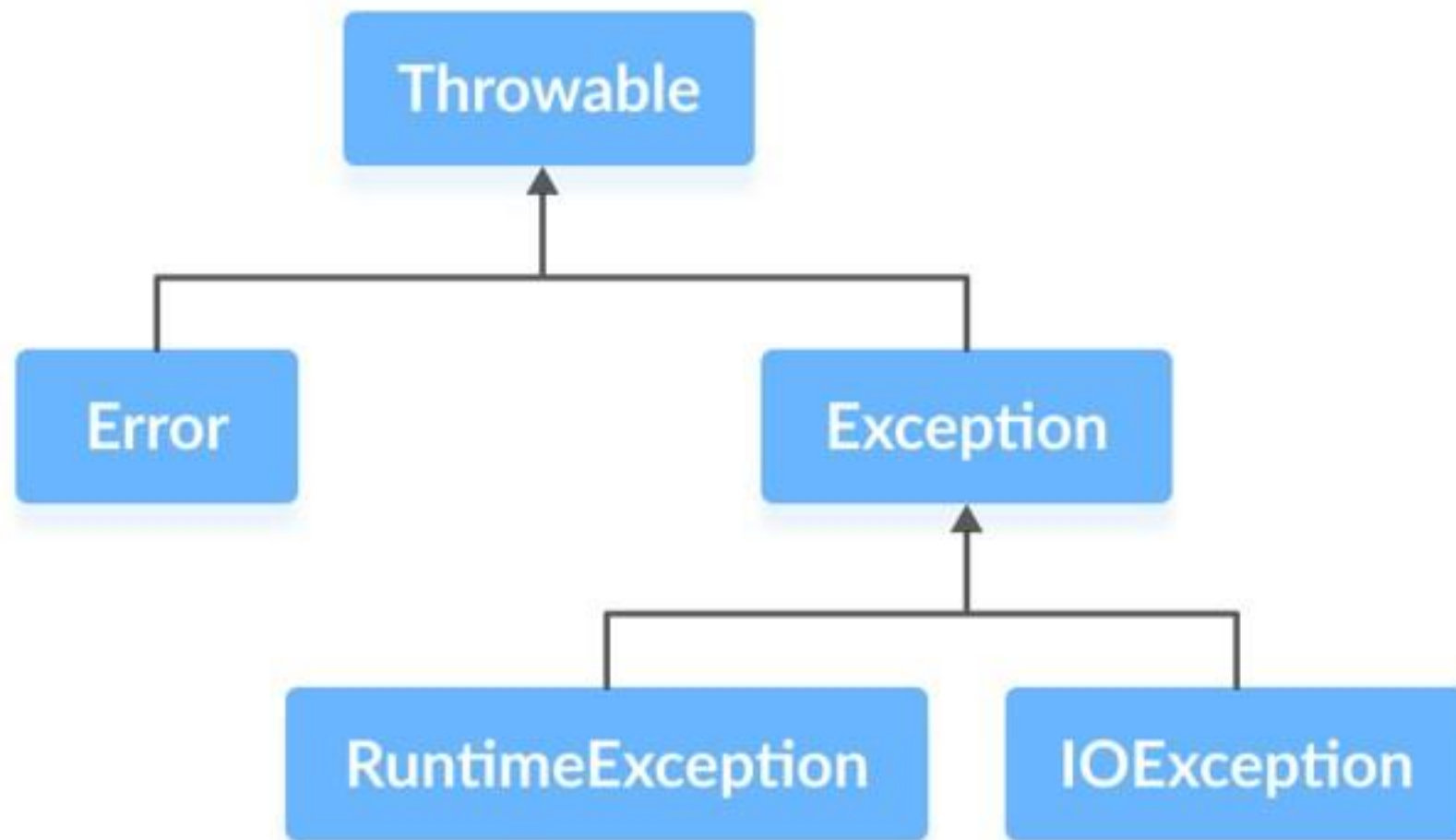
Java Exceptions

An **exception** is an unexpected event that occurs during program execution. It affects the flow of the program instructions which can cause the program to terminate abnormally.

An exception can occur for many reasons. Some of them are:

- Invalid user input
- Device failure
- Loss of network connection
- Physical limitations (out of disk memory)
- Code errors
- Opening an unavailable file

Java Exception hierarchy



Errors

Errors represent irrecoverable conditions such as Java virtual machine (JVM) running out of memory, memory leaks, stack overflow errors, library incompatibility, infinite recursion, etc.

Errors are usually beyond the control of the programmer and we should not try to handle errors.

Exceptions

Exceptions can be caught and handled by the program.

When an exception occurs within a method, it creates an object. This object is called the exception object.

It contains information about the exception such as the name and description of the exception and state of the program when the exception occurred.

Java Exception Types

1. RuntimeException

A runtime exception happens due to a programming error. They are also known as unchecked exceptions.

These exceptions are not checked at compile-time but run-time. Some of the common runtime exceptions are:

Improper use of an API - `IllegalArgumentException`

Null pointer access (missing the initialization of a variable) - `NullPointerException`

Out-of-bounds array access - `ArrayIndexOutOfBoundsException`

Dividing a number by 0 - `ArithmeticException`

“If it is a runtime exception, it is your fault”.

The `NullPointerException` would not have occurred if you had checked whether the variable was initialized or not before using it.

An `ArrayIndexOutOfBoundsException` would not have occurred if you tested the array index against the array bounds.

2. IOException

An IOException is also known as a checked exception. They are checked by the compiler at the compile-time and the programmer is prompted to handle these exceptions.

Some of the examples of checked exceptions are:

Trying to open a file that doesn't exist results in FileNotFoundException

Trying to read past the end of a file

Java Exception Handling

We know that exceptions abnormally terminate the execution of a program.

This is why it is important to handle exceptions. Here's a list of different approaches to handle exceptions in Java.

- try...catch block
- finally block
- throw and throws keyword

1. Java try...catch block

The try-catch block is used to handle exceptions in Java. Here's the syntax of try...catch block:

```
try {  
    // code  
}  
catch(Exception e) {  
    // code  
}
```

Here, we have placed the code that might generate an exception inside the try block. Every try block is followed by a catch block.

When an exception occurs, it is caught by the catch block. The catch block cannot be used without the try block.


```
class Main {  
    public static void main(String[] args) {  
  
        try {  
  
            // code that generate exception  
            int divideByZero = 5 / 0;  
            System.out.println("Rest of code in try block");  
        }  
  
        catch (ArithmeticException e) {  
            System.out.println("ArithmeticException => " + e.getMessage());  
        }  
    }  
}
```

2. Java finally block

In Java, the finally block is always executed no matter whether there is an exception or not. The finally block is optional. And, for each try block, there can be only one finally block.

The basic syntax of finally block is:

```
try {  
    //code  
}  
catch (ExceptionType1 e1) {  
    // catch block  
}  
finally {  
    // finally block always executes  
}
```

If an exception occurs, the finally block is executed after the try...catch block. Otherwise, it is executed after the try block. For each try block, there can be only one finally block.

```
class Main {  
    public static void main(String[] args) {  
        try {  
            // code that generates exception  
            int divideByZero = 5 / 0;  
        }  
  
        catch (ArithmeticException e) {  
            System.out.println("ArithmeticException => " + e.getMessage());  
        }  
  
        finally {  
            System.out.println("This is the finally block");  
        }  
    }  
}
```


3. Java throw and throws keyword

The Java throw keyword is used to explicitly throw a single exception.

When we throw an exception, the flow of the program moves from the try block to the catch block.

Example: Exception handling using Java throw

```
class Main {  
    public static void divideByZero() {  
  
        // throw an exception  
        throw new ArithmeticException("Trying to divide by 0");  
    }  
  
    public static void main(String[] args) {  
        divideByZero();  
    }  
}
```

Note: In Java, we can use a try block without a catch block. However, we cannot use a catch block without a try block.

Note: There are some cases when a finally block does not execute:

- Use of System.exit() method
- An exception occurs in the finally block
- The death of a thread

Multiple Catch blocks

For each try block, there can be zero or more catch blocks. Multiple catch blocks allow us to handle each exception differently.

The argument type of each catch block indicates the type of exception that can be handled by it. For example,


```
class ListOfNumbers {  
    public int[] arr = new int[10];  
    public void writeList() {  
        try {  
            arr[10] = 11;  
        }  
        catch (NumberFormatException e1) {  
            System.out.println("NumberFormatException => " + e1.getMessage());  
        }  
        catch (IndexOutOfBoundsException e2) {  
            System.out.println("IndexOutOfBoundsException => " + e2.getMessage());  
        }  
    }  
}  
  
class Main {  
    public static void main(String[] args) {  
        ListOfNumbers list = new ListOfNumbers();  
        list.writeList();  
    }  
}
```

Java Annotations

Java annotations are metadata (data about data) for our program source code.

They provide additional information about the program to the compiler but are not part of the program itself. These annotations do not affect the execution of the compiled program.

Annotations start with @. Its syntax is:

@AnnotationName

Let's take an example of @Override annotation.

The @Override annotation specifies that the method that has been marked with this annotation overrides the method of the superclass with the same method name, return type, and parameter list.

It is not mandatory to use @Override when overriding a method. However, if we use it, the compiler gives an error if something is wrong (such as wrong parameter type) while overriding the method.


```
class Animal {  
    public void displayInfo() {  
        System.out.println("I am an animal.");  
    }  
}  
class Dog extends Animal {  
    @Override  
    public void displayInfo() {  
        System.out.println("I am a dog.");  
    }  
}  
class Main {  
    public static void main(String[] args) {  
        Dog d1 = new Dog();  
        d1.displayInfo();  
    }  
}
```

Annotation formats

Annotations may also include elements (members/attributes/parameters).

1. Marker Annotations

Marker annotations do not contain members/elements. It is only used for marking a declaration.

Its syntax is:

`@AnnotationName()`

Since these annotations do not contain elements, parentheses can be excluded. For example,

`@Override`

2. Single element Annotations

A single element annotation contains only one element.

Its syntax is:

```
@AnnotationName(elementName = "elementValue")
```

If there is only one element, it is a convention to name that element as value.

```
@AnnotationName(value = "elementValue")
```

In this case, the element name can be excluded as well. The element name will be value by default.

```
@AnnotationName("elementValue")
```


3. Multiple element Annotations

These annotations contain multiple elements separated by commas.

Its syntax is:


```
@AnnotationName(element1 = "value1", element2 = "value2")
```

Use of Annotations

Compiler instructions - Annotations can be used for giving instructions to the compiler, detect errors or suppress warnings. The built-in annotations `@Deprecated`, `@Override`, `@SuppressWarnings` are used for these purposes.

Compile-time instructions - Compile-time instructions provided by these annotations help the software build tools to generate code, XML files and many more.

Runtime instructions - Some annotations can be defined to give instructions to the program at runtime. These annotations are accessed using Java Reflection.



These annotations can be categorized as:

1. Predefined annotations

@Deprecated

@Override

@SuppressWarnings

@SafeVarargs

@FunctionalInterface

2. Custom Annotations

It is also possible to create our own custom annotations.

Its syntax is:

```
[Access Specifier] @interface<AnnotationName> {  
    DataType <Method Name>() [default value];  
}
```

Here is what you need to know about custom annotation:

Ex:

```
@interface MyCustomAnnotation {  
    String value() default "default value";  
}  
class Main {  
    @MyCustomAnnotation(value = "programming")  
    public void method1() {  
        System.out.println("Test method 1");  
    }  
    public static void main(String[] args) throws Exception {  
        Main obj = new Main();  
        obj.method1();  
    }  
}
```

3. Meta-annotations

@Retention

@Documented

@Target

@Inherited

@Repeatable



Java Logging

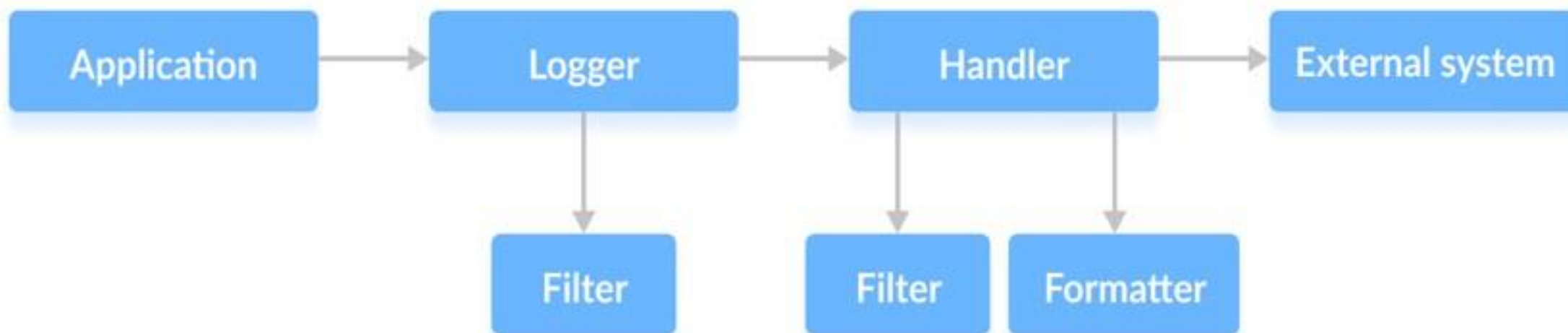
Java allows us to create and capture log messages and files through the process of logging.

In Java, logging requires frameworks and APIs. Java has a built-in logging framework in the `java.util.logging` package.

We can also use third-party frameworks like Log4j, Logback, and many more for logging purposes.

Java Logging Components

The figure below represents the core components and the flow of control of the Java Logging API (`java.util.logging`).



Advantages of Logging

Here are some of the advantages of logging in Java.

- helps in monitoring the flow of the program
- helps in capturing any errors that may occur
- provides support for problem diagnosis and debugging

Java Assertions

Assertions in Java help to detect bugs by testing code we assume to be true.

An assertion is made using the **assert** keyword.

Its syntax is:

```
assert condition;
```

Here, condition is a boolean expression that we assume to be true when the program executes.

Enabling Assertions

By default, assertions are disabled and ignored at runtime.

To enable assertions, we use:

```
java -ea:arguments
```

OR

```
java -enableassertions:arguments
```

When assertions are enabled and the condition is true, the program executes normally.

But if the condition evaluates to false while assertions are enabled, JVM throws an `AssertionError`, and the program stops immediately.

Disabling Assertions

To disable assertions, we use:

```
java -da arguments  
OR
```

```
java -disableassertions arguments
```

To disable assertion in system classes, we use:

```
java -dsa:arguments  
OR
```

```
java -disablesystemassertions:arguments
```

The arguments that can be passed while disabling assertions are the same as while enabling them.

Advantages of Assertion

1. Quick and efficient for detecting and correcting bugs.
2. Assertion checks are done only during development and testing. They are automatically removed in the production code at runtime so that it won't slow the execution of the program.
3. It helps remove boilerplate code and make code more readable.
4. Refactors and optimizes code with increased confidence that it functions correctly.



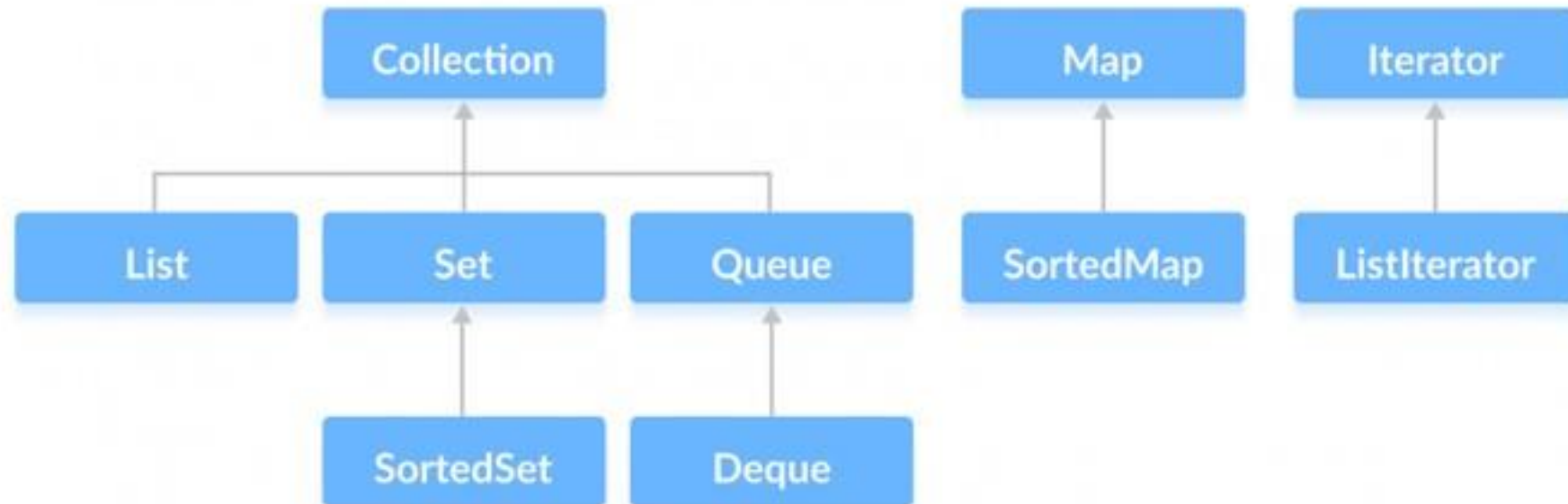
Talent Battle

Java List

Java Collections Framework

The Java **collections** framework provides a set of interfaces and classes to implement various data structures and algorithms.

Java Collections Framework



Java Collection Interface

The Collection interface is the root interface of the collections framework hierarchy.

Java does not provide direct implementations of the Collection interface but provides implementations of its sub interfaces like **List**, **Set**, and **Queue**.

Why the Collections Framework?

The Java collections framework provides various data structures and algorithms that can be used directly. This has two main advantages:

We do not have to write code to implement these data structures and algorithms manually.

Our code will be much more efficient as the collections framework is highly optimized. Moreover, the collections framework allows us to use a specific data structure for a particular type of data. Here are a few examples,

If we want our data to be unique, then we can use the Set interface provided by the collections framework.

To store data in key/value pairs, we can use the Map interface.

The ArrayList class provides the functionality of resizable arrays.

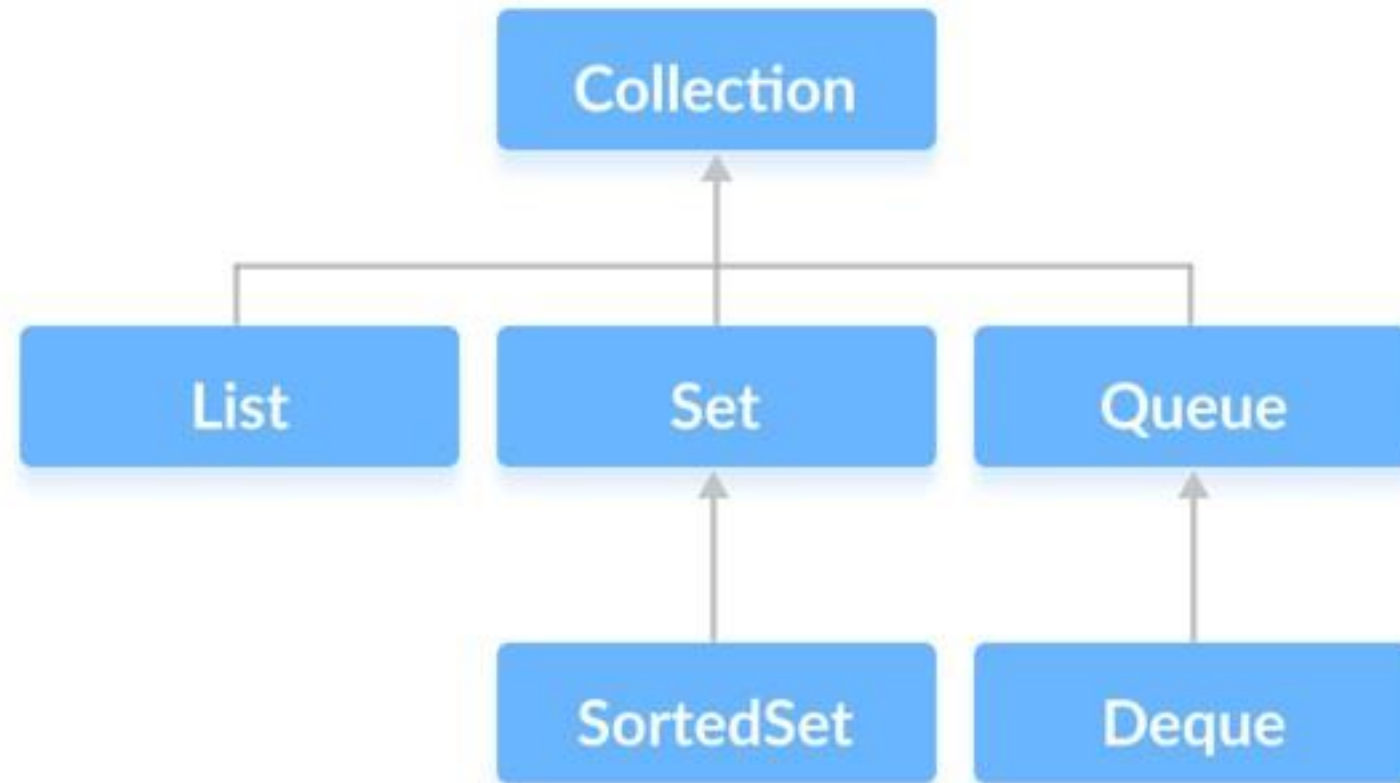
Example: ArrayList Class of Collections

// The Collections framework is defined in the java.util package

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

```
class Main {  
    public static void main(String[] args){  
        ArrayList<String> animals = new ArrayList<>();  
        // Add elements  
        animals.add("Dog");  
        animals.add("Cat");  
        animals.add("Horse");  
  
        System.out.println("ArrayList: " + animals);  
    }  
}
```

Java Collection Interface



The Collection interface is the root interface of the Java collections framework.

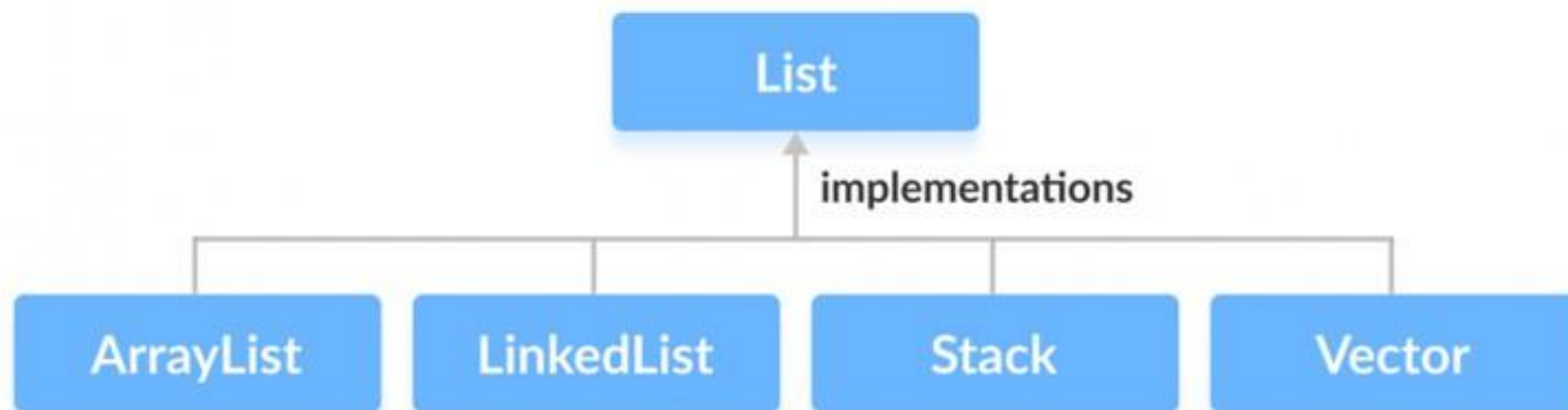
Methods of Collection

The Collection interface includes various methods that can be used to perform different operations on objects. These methods are available in all its sub interfaces.

- `add()` - inserts the specified element to the collection
- `size()` - returns the size of the collection
- `remove()` - removes the specified element from the collection
- `iterator()` - returns an iterator to access elements of the collection
- `addAll()` - adds all the elements of a specified collection to the collection
- `removeAll()` - removes all the elements of the specified collection from the collection
- `clear()` - removes all the elements of the collection

Java List

In Java, the List interface is an ordered collection that allows us to store and access elements sequentially. It extends the Collection interface.



How to use List?

In Java, we must import java.util.List package in order to use List.

```
// ArrayList implementation of List  
List<String> list1 = new ArrayList<>();
```

```
// LinkedList implementation of List  
List<String> list2 = new LinkedList<>();
```

Here, we have created objects list1 and list2 of classes ArrayList and LinkedList.

These objects can use the functionalities of the List interface.

Methods of List

- `add()` - adds an element to a list
- `addAll()` - adds all elements of one list to another
- `get()` - helps to randomly access elements from lists
- `iterator()` - returns iterator object that can be used to sequentially access elements of lists
- `set()` - changes elements of lists
- `remove()` - removes an element from the list
- `removeAll()` - removes all the elements from the list
- `clear()` - removes all the elements from the list (more efficient than `removeAll()`)
- `size()` - returns the length of lists
- `toArray()` - converts a list into an array
- `contains()` - returns true if a list contains specified element


```
import java.util.List;  
import java.util.ArrayList;
```

```
class Main {
```

```
    public static void main(String[] args) {  
        // Creating list using the ArrayList class  
        List<Integer> numbers = new ArrayList<>();
```

```
        // Add elements to the list
```

```
        numbers.add(1);
```

```
        numbers.add(2);
```

```
        numbers.add(3);
```

```
        System.out.println("List: " + numbers);
```

```
        // Access element from the list
```

```
        int number = numbers.get(2);
```

```
        System.out.println("Accessed Element: " + number);
```

```
        // Remove element from the list
```

```
        int removedNumber = numbers.remove(1);
```

```
        System.out.println("Removed Element: " + removedNumber);
```

```
    }
```

```
}
```



```
import java.util.List;
import java.util.LinkedList;
class Main {
    public static void main(String[] args) {
        // Creating list using the LinkedList class
        List<Integer> numbers = new LinkedList<>();
        // Add elements to the list
        numbers.add(1);
        numbers.add(2);
        numbers.add(3);
        System.out.println("List: " + numbers);
        // Access element from the list
        int number = numbers.get(2);
        System.out.println("Accessed Element: " + number);
        // Using the indexOf() method
        int index = numbers.indexOf(2);
        System.out.println("Position of 3 is " + index);
        // Remove element from the list
        int removedNumber = numbers.remove(1);
        System.out.println("Removed Element: " + removedNumber);
    }
}
```

Java List vs. Set

Both the List interface and the Set interface inherits the Collection interface. However, there exists some difference between them.

Lists can include duplicate elements. However, sets cannot have duplicate elements. Elements in lists are stored in some order. However, elements in sets are stored in groups like sets in mathematics.