Super

In Java, the super keyword is used to refer to the immediate parent class of a subclass (derived class).

Variable : You can use super to access the instance variables of the superclass. This can be useful when you want to refer to the superclass's instance variables with the same name as those in the subclass.

```
class Animal {
   int legs = 4;
}

class Spider extends Animal {
   int legs = 8;

   void displayLegs() {
      System.out.println("Spider legs: " + legs);
      System.out.println("Animal legs: " + super.legs);
   }
}
```

Method : You can use super to access the members (fields or methods) of the superclass (parent class) when there is a name conflict between the superclass and subclass.

```
class Animal {
   void sound() {
      System.out.println("Animal makes a sound");
   }
}

class Dog extends Animal {
   void sound() {
      super.sound(); // Calls the sound() method of the superclass
      System.out.println("Dog barks");
   }
}
```

Constructor : You can use super to call the constructor of the superclass. This is especially useful when the superclass has parameterized constructors, and you want to initialize the superclass's fields.

```
class Animal {
    String species;

    Animal(String species) {
        this.species = species;
    }
}

class Dog extends Animal {
    String breed;

    Dog(String species, String breed) {
        super(species); // Calls the constructor of the Animal class
        this.breed = breed;
    }
}
```

Constructor-implicit : The subclass constructor, if not explicitly defined, includes a hidden call to the superclass's no-argument constructor using super(). This ensures that the superclass is properly initialized before any subclass-specific code is executed.

If the superclass does not have a no-argument constructor (i.e., if it only has parameterized constructors), then the compiler will generate an error. In this case, you must explicitly define a constructor in the subclass that calls one of the superclass's parameterized constructors using super(...).

```
class Animal {
    String species;

    Animal() {
        species = "Unknown";
        System.out.println("Animal constructor called");
    }

    Animal(String species) {
        this.species = species;
        System.out.println("Animal constructor with parameter called");
    }
}

class Dog extends Animal {
    String breed;

    Dog() {
        breed = "Unknown";
        System.out.println("Dog constructor called");
    }

    Dog(String species, String breed) {
        super(species); // Calls the parameterized constructor of the Animal class
        this.breed = breed;
        System.out.println("Dog constructor with parameters called");
    }
}
```

Final

In Java, the final keyword is used to indicate that something (a variable, method, or class) cannot be changed or extended further, depending on where it is applied.

Variable

```
final int MAX_VALUE = 100;
```

Method

```
class Parent {
    final void display() {
        System.out.println("This is a final method.");
    }
}
```

Classes

```
final class MyFinalClass {
    // Class definition
}
```

Parameters

```
void printValue(final int value) {
    // 'value' cannot be modified within this method
    System.out.println("Value: " + value);
}
```

Instance initializer

In Java, an instance initializer is a block of code within a class that is used to initialize instance variables (non-static fields) of an object. It is executed when an instance of the class is created, before any other constructor is invoked. Instance initializers are enclosed within curly braces {} and are not associated with any method or constructor.

```
class MyClass {
   // instance variable
   int x;

   // instance initializer
   {
      // code to initialize x
      x = 10;
      System.out.println("Instance initializer executed.");
   }

   // constructors and other methods can follow
   // ...
}
```

String

In Java, the String class is a fundamental class that represents a sequence of characters. It is part of the java.lang package, which is automatically imported into every Java program, so you don't need to explicitly import it. The String class is widely used in Java for working with textual data.

Immutable : Strings in Java are immutable, which means their values cannot be changed after they are created. When you perform operations on a String, such as concatenation or substring extraction, a new String object is created with the result.

String Literals : Java supports string literals, which are sequences of characters enclosed in double quotes. String literals are automatically converted to String objects.

String Pool : Java maintains a special memory area called the "string pool" for storing string literals. When you create a string using a literal that already exists in the pool, Java reuses the existing string instead of creating a new one. This helps save memory and improves performance.

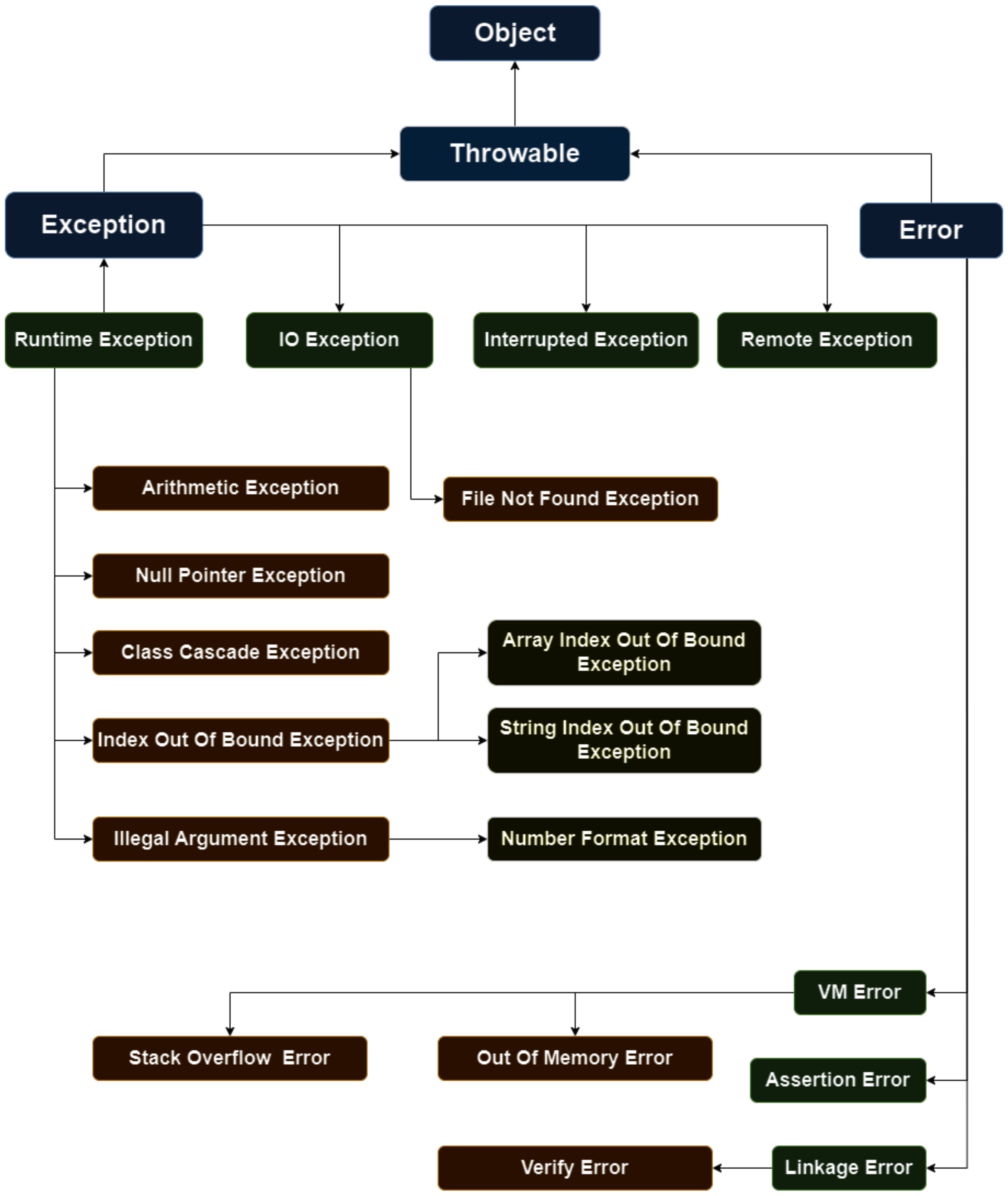
Refer to the practical example for below points.

- Concatenation
- Comparison
- StringBuffer vs. StringBuilder
- String methods(charAt, concat, contains, endsWith, equals, indexOf, lastIndexOf, length, replace, split, startWith, subString, toCharArray, toLowerCase, toUpperCase, trim, join)

Selenium TestNG basic example

Code is provided with this material.

Exception handling


Object
Throwable
Exception
Error
Runtime Exception
IO Exception
Interrupted Exception
Remote Exception
Arithmetic Exception
File Not Found Exception
Null Pointer Exception
Class Cascade Exception
Array Index Out Of Bound Exception
Index Out Of Bound Exception
String Index Out Of Bound Exception
Illegal Argument Exception
Number Format Exception
VM Error
Stack Overflow Error
Out Of Memory Error
Assertion Error
Verify Error
Linkage Error

Exception handling in Java is a mechanism that allows you to gracefully handle unexpected or exceptional situations that may occur during the execution of a program. Java provides a robust and structured way to deal with exceptions using the try, catch, throw, throws, and finally keywords.

Exceptions in Java are divided into two main categories

- Checked exception : These are exceptions that are checked at compile time. They must either be caught using catch blocks or declared using the throws keyword in the method signature.

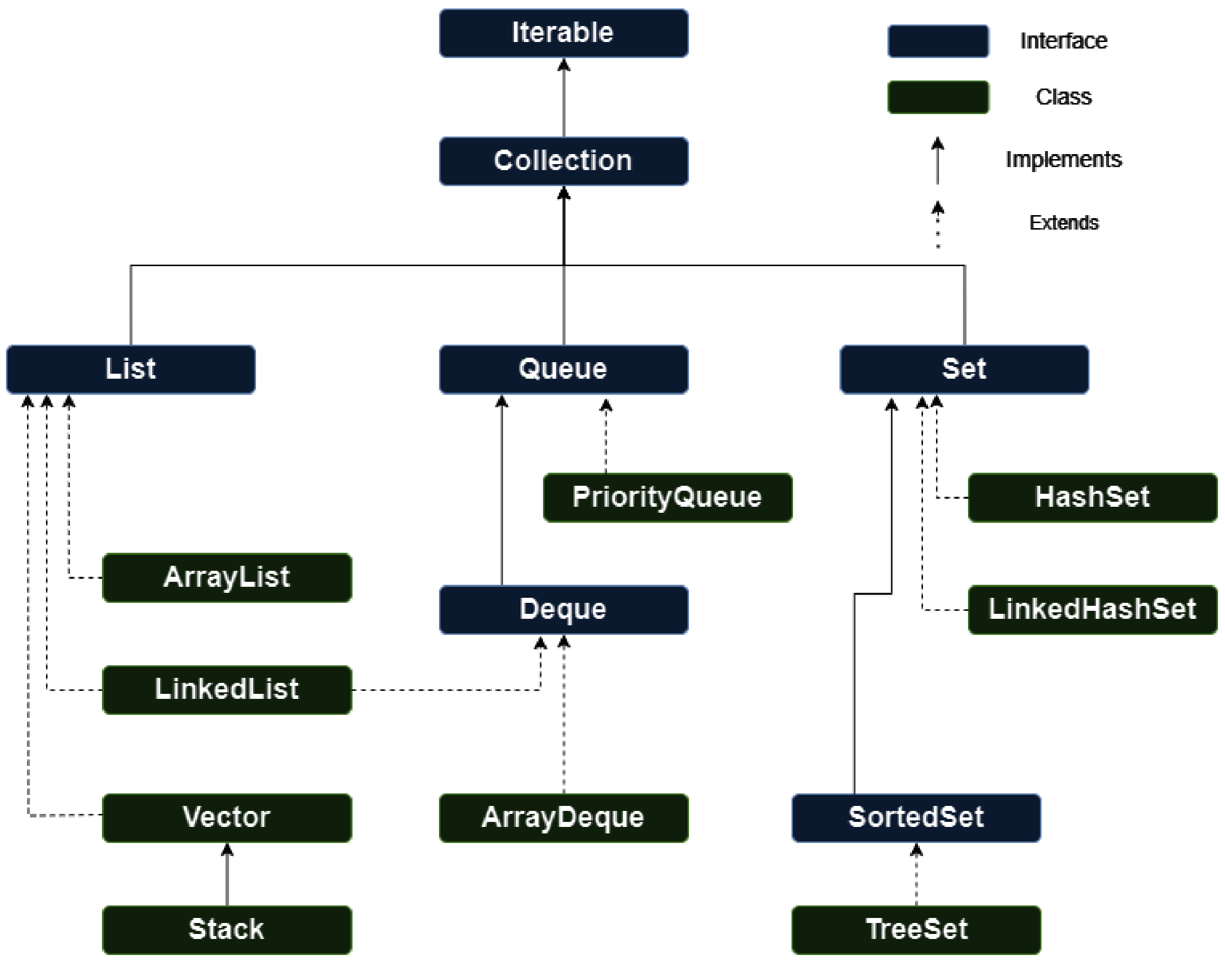
- Unchecked Exceptions (Runtime Exceptions) : These are exceptions that are not checked at compile time. They can occur at runtime and are subclasses of RuntimeException. They do not require explicit handling but can be handled if needed.

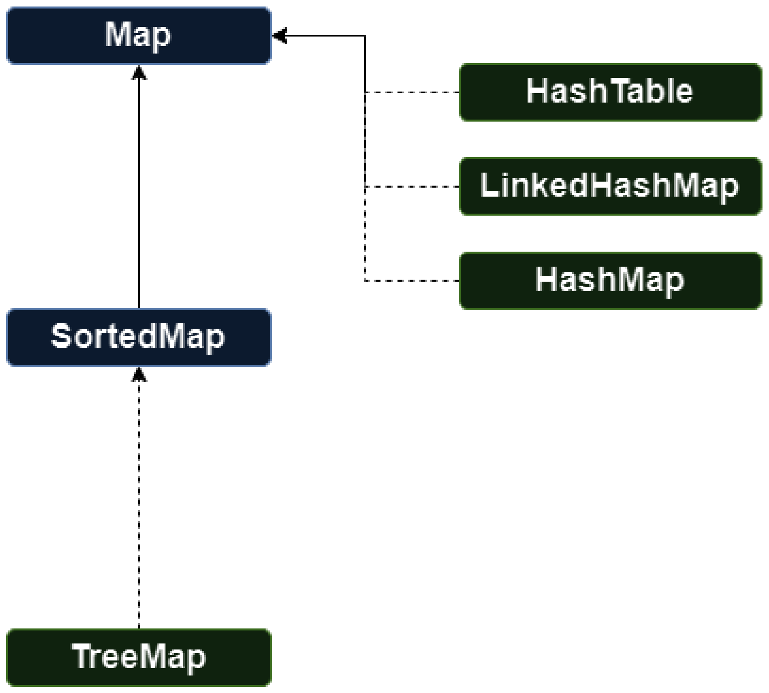
Way of handling Exceptions

- TryCatch/ TryCatchFinally
  The **finally** block is used to specify code that should be executed regardless of whether an exception occurs or not. It is often used for cleanup operations, such as closing resources (e.g., files, database connections).
- Multi-Catch
- Try with resource
- Throw vs throws
  The **throw** keyword is used to explicitly throw an exception in your code.
  The **throws** keyword is used in method signatures to declare that a method might throw certain
  exceptions. Calling code must either catch these exceptions or propagate them using the throws clause.

Java collection framework

- Iterator through collection
- List(ArrayList)
- Map(HashMap)
- Set(HashSet)


Iterable
Interface
Class
Collection
Implements
Extends
List
Queue
Set
PriorityQueue
HashSet
ArrayList
Deque
LinkedHashSet
LinkedList
Vector
ArrayDeque
SortedSet
Stack
TreeSet



File Handling

- read/write something.properties file in java