Core Java Concepts

Topics

- **Naming Convention in Java.**
- **Decision Making in Java.**
- **Java Variables.**
- **Java Return Type.**
- **Constructors in Java.**
- **Static keyword in java.**
- **Static Blocks.**
- **Static in Java.**
- **Static Class.**
- **This keyword in java**
- **Inheritance in Java**
- **Aggregation in Java**
- **Java Polymorphism**
- **Method Overloading**
- **Method Overriding in Java**
- **Interface in Java**
- **Abstract classes**
- **Access Modifier**
- **Java Enum**
- **Type conversion in Java with Examples**
- **Final Keyword In Java**
- **Runtime Polymorphism in Java**
- **Static Binding and Dynamic Binding**
- **Java instance of**
- **Encapsulation in Java**
- **What is Abstraction**
- **Wrapper classes in Java**
- **String In Java**
- **Java String Buffer class**
- **Java StringBuilder class**
- **Exception in Java**
- **throw exception in java**
- **throws Keyword**
- **Collections in Java**
- **Cloning in java**
- **Comparable in java**
- **Comparable in java**
- **Comparator in java**
- **Multithreading in java**

By Bhanu Pratap Singh
https://www.youtube.com/user/MrBhanupratap29/playlists
https://www.facebook.com/learnbybhanupratap/
https://www.udemy.com/seleniumbybhanu/

- **Others topic will be updated soon….**

What is Java

Java is a programming language first released by Sun Microsystems in 1995. There are lots of applications and websites that will not work unless you have Java installed. Java is fast, secure, and reliable. It is one of the most used programming languages.

Java Version History
There are many java versions that has been released. Current stable release of Java is Java SE 8.

1. JDK Alpha and Beta (1995)
2. JDK 1.0 (23rd Jan, 1996)
3. JDK 1.1 (19th Feb, 1997)
4. J2SE 1.2 (8th Dec, 1998)
5. J2SE 1.3 (8th May, 2000)
6. J2SE 1.4 (6th Feb, 2002)
7. J2SE 5.0 (30th Sep, 2004)
8. Java SE 6 (11th Dec, 2006)
9. Java SE 7 (28th July, 2011)
10. Java SE 8 (18th March, 2014)

Features of Java
There is given many features of java. The Java Features given below are simple and easy to understand.

1. Simple
2. Object-Oriented
3. Portable
4. Platform independent
5. Secured
6. Robust
7. Architecture neutral 8. Dynamic
8. Interpreted
9. High Performance
10. Multithreaded
11. Distributed

1. Simple
Java is easy to learn and its syntax is quite simple, clean and easy to understand. The confusing and ambiguous concepts of C++ are either left out in Java or they have been re-implemented in a cleaner way.

E.g. : Pointers and Operator Overloading are not there in java but were an important part of C++.

2. Object-Oriented
In java, everything is Object which has some data and behaviour. Java can be easily extended as it is based on Object Model.

3. Portable
Java Byte code can be carried to any platform. No implementation dependent features. Everything related to storage is predefined, example: size of primitive data types

4. platform independent
Unlike other programming languages such as C, C++ etc which are compiled into platform specific machines. Java is guaranteed to be write-once, runanywhere language.

On compilation Java program is compiled into bytecode. This bytecode is platform independent and can be run on any machine, plus this bytecode format also provide security. Any machine with Java Runtime Environment can run Java Programs.

5. Secure
When it comes to security, Java is always the first choice. With java secure features it enables us to develop virus free, temper free system. Java program always runs in Java runtime environment with almost null interaction with system OS, hence it is more secure.

6. Robust
Java makes an effort to eliminate error prone codes by emphasizing mainly on compile time error checking and runtime checking. But the main areas which Java improved were Memory Management and mishandled Exceptions by introducing automatic Garbage Collector and Exception Handling.

7. Architecture neutral
Compiler generates bytecodes, which have nothing to do with a particular computer architecture, hence a Java program is easy to interpret on any machine.

8. High Performance
Java is an interpreted language, so it will never be as fast as a compiled language like C or C++. But, Java enables high performance with the use of just-in-time compiler.

9. Multithreaded
A thread is like a separate program, executing concurrently. We can write Java programs that deal with many tasks at once by defining multiple threads. The main advantage of multi-threading is that it doesn't occupy memory for each thread. It shares a common memory area. Threads are important for multimedia, Web applications etc.

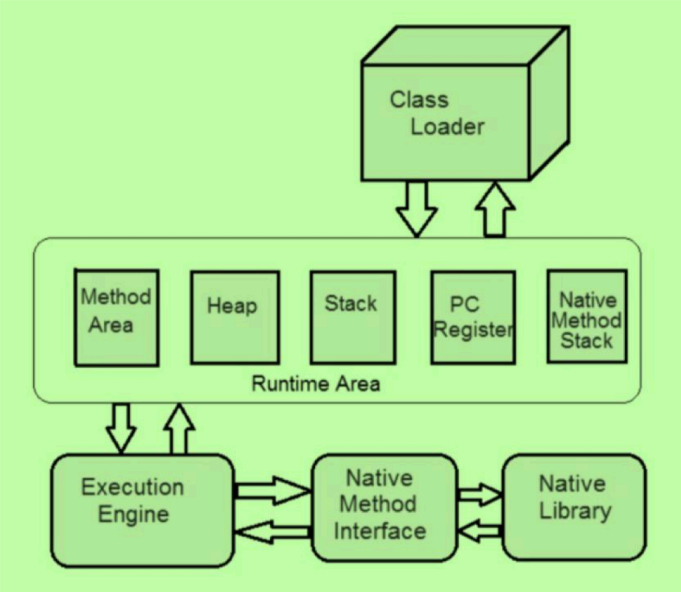
10. Distributed
We can create distributed applications in java. RMI and EJB are used for creating distributed applications. We may access files by calling the methods from any machine on the internet.

JDK, JRE and JVM
• **JDK –** Java Development Kit (in short JDK) is Kit which provides the environment to develop and execute(run) the Java program. JDK is a kit (or package) which includes two things

• **JRE –** Java Runtime Environment (to say JRE) is an installation package which provides environment to only run (not develop) the java program (or application) onto your machine. JRE is only used by them who only wants to run the Java Programs i.e. end users of your system.
• **JVM –** Java Virtual machine(JVM) is a very important part of both JDK and JRE

because it is contained or inbuilt in both. Whatever Java program you run using JRE or JDK goes into JVM and JVM is responsible for executing the java program line by line hence it is also known as interpreter.
Note: JDK is only used by Java Developers.

Difference between JDK, JRE and JVM. To understand the difference between these three, let us consider the following diagram. JVM Architecture



Method Area:
1) Java Virtual Machine Method Area can be used for storing all the class code and method code.
2) All classes bytecode is loaded and stored in this run time area , and all static

3) variables are created in this area.

Heap Memory:

1) JVM Heap Area can be used for storing all the objects that are created.
It is the main memory of JVM , all objects of classes :- non static variables memory are created in this run time area.
2) This runtime area memory is finite memory.
3) This area can be configured at the time of setting up of runtime environment using non-standard option like
java -xms <size> class-name.
4) This can be expandable by its own , depending on the object creation.
5) Method area and Heap area both are sharable memory areas.

Java Stack area:

1) JVM stack Area can be used for storing the information of the methods. That is under execution. The java stack can be considered as the combination of stack frames where every frame will contain the stat of a single method.
2) In this runtime area all Java methods are executed.
In this runtime JVM by default creates two threads. they are
1.main method
2.Garbage Collection Thread
main thread responsible to execute java methods starts with main method.
Also, responsible to create objects in heap area if its finds new keyword in any method logic.
Garbage collection thread is responsible to destroy all unused objects from heap area.

PC Register (program counter) area:

1) The PC Register Java Virtual Machine will contain address of the next instruction that have to be executed.

Java Native Stack:

2) Java native stack area is used for storing non-java coding available in the application. The non-java code is called as native code.

Execution Engine:

The Execution Engine of JVM is Responsible for executing the program and it **contains two parts.**

1. Interpreter.
2. JIT Compiler (just in time compiler).

1) The java code will be executed by both interpreter and JIT compiler simultaneously which will reduce the execution time and them by providing high performance. The code that is executed by JIT compiler is called as

HOTSPOTS (company).

Object, Class and Method

Class

A class is a user defined blueprint or prototype from which objects are created. It represents the set of properties or methods that are common to all objects of one type.

Class has

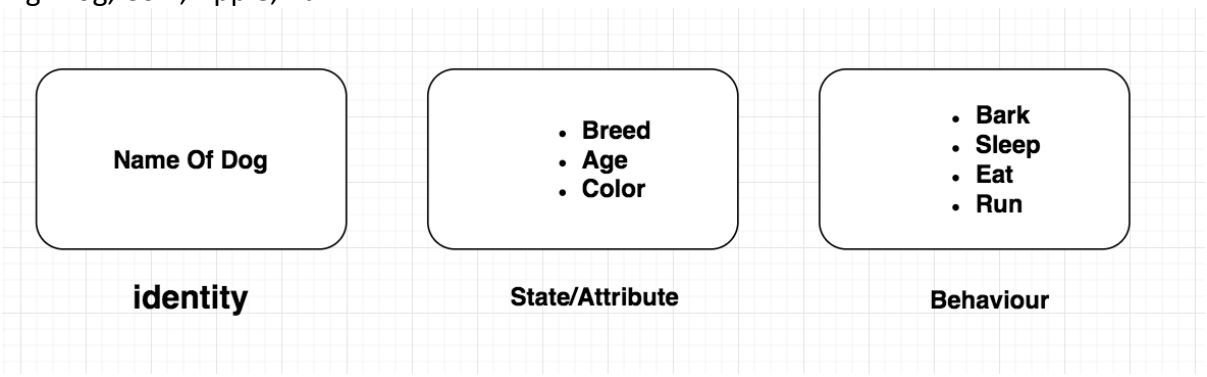
1. **Modifiers** : A class can be public or has default access.
2. **Class name:** The name should begin with a initial letter (capitalized by convention).
3. **Superclass(if any):** The name of the class's parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.
4. **Interfaces(if any):** A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.
5. **Body:** The class body surrounded by braces, { }.

Objects correspond to things found in the real world.

An object consists of :

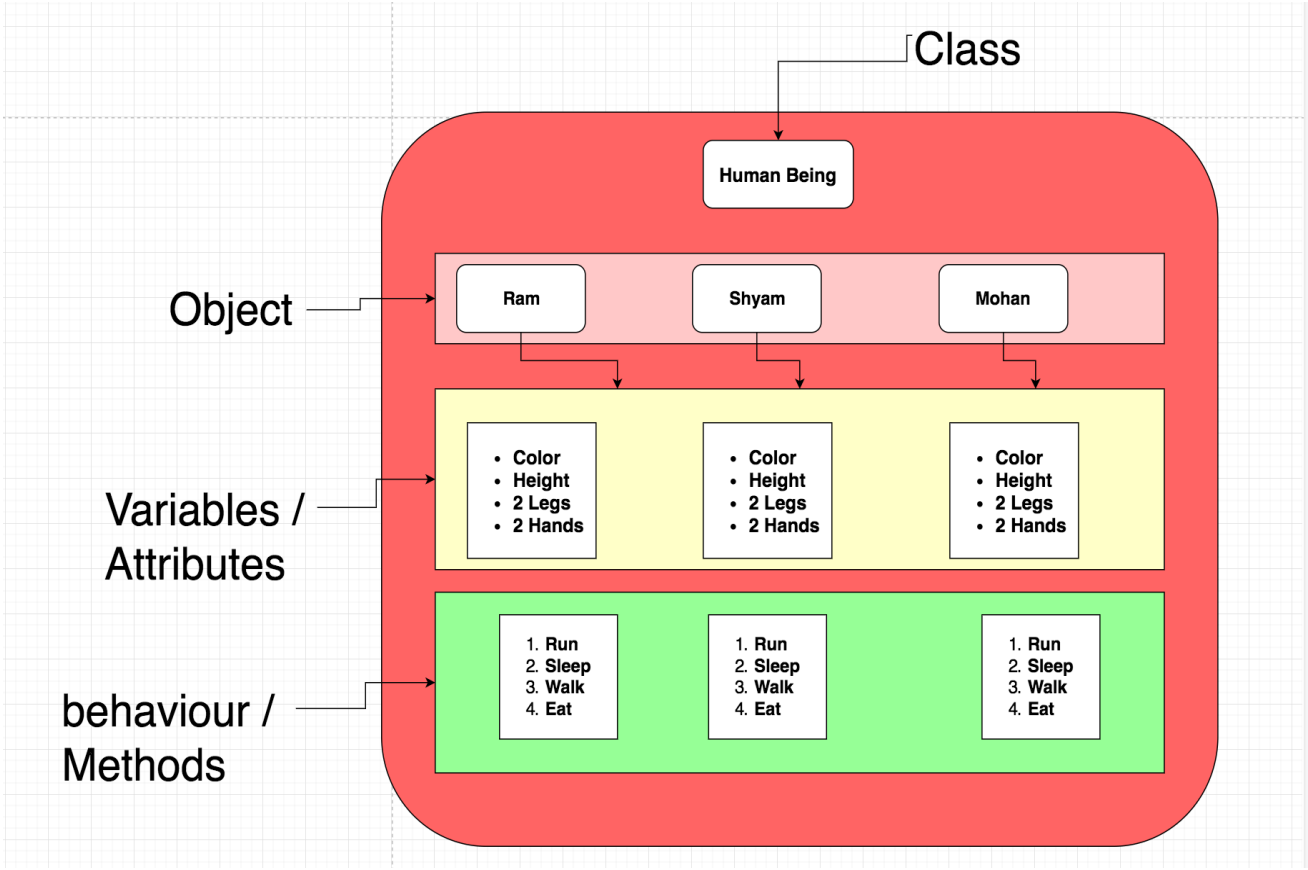
1. **State** : It is represented by attributes of an object. It also reflects the properties of an object.
2. **Behaviour** : It is represented by methods of an object. It also reflects the response of an object with other objects.
3. **Identity** : It gives a unique name to an object and enables one object to interact with other objects.

E.g. Dog, Cow, Apple, Ram

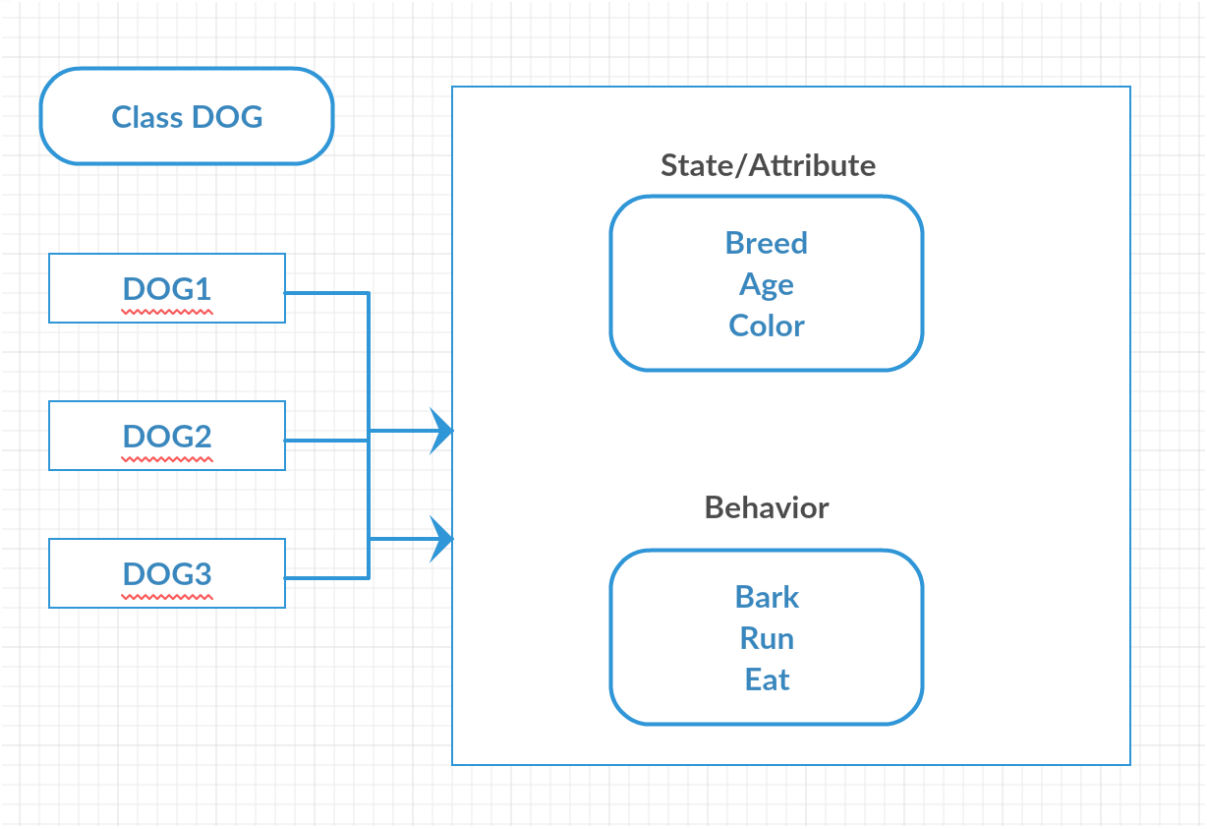


If we make 5 Object of same class, each object will have one copy of attributes/Behaviour

By Bhanu Pratap Singh
https://www.youtube.com/user/MrBhanupratap29/playlists
https://www.facebook.com/learnbybhanupratap/
https://www.udemy.com/seleniumbybhanu/



By Bhanu Pratap Singh
https://www.youtube.com/user/MrBhanupratap29/playlists
https://www.facebook.com/learnbybhanupratap/
https://www.udemy.com/seleniumbybhanu/



Method in Java

In Java, a method is like a function which is used to expose the behaviour of an object.

Advantage of Method

Code reusability

Code optimisation.

```
package ClassExample;
public class Example1 {

	int age;

	public int getAge() {
		return age;
	}
	public void setAge(int age) {
		this.age = age;
	}
	public static void main(String[] args) {
		Example1 obj1 = new Example1();
		obj1.setAge(4);
```

By Bhanu Pratap Singh

https://www.youtube.com/user/MrBhanupratap29/playlists

https://www.facebook.com/learnbybhanupratap/

https://www.udemy.com/seleniumbybhanu/

```
            obj1.getAge();

            Example1 obj2 = new Example1();
            obj1.setAge(6);
            obj2.getAge();
        }

}
```

public static void main(String[] args)

Java main method is the entry point of any java program. Its syntax is always public static void main(String[] args). You can only change the name of String array argument, for example you can change args to myStringArgs.

Also String array argument can be written as String... args or String args[].

Public

This is the access modifier of the main method. It has to be public so that java runtime can execute this method. Remember that if you make any method non-public then it's not allowed to be executed by any program, there are some access restrictions applied. So it means that the main method has to be public.

Static

When java runtime starts, there is no object of the class present. That's why the main method has to be static so that JVM can load the class into memory and call the main method. If the main method won't be static, JVM would not be able to call it because there is no object of the class is present

Void

Java programming mandates that every method provide the return type. Java main method doesn't return anything, that's why it's return type is void. This has been done to keep things simple because once the main method is finished executing, java program terminates. So there is no point in returning anything, there is nothing that can be done for the returned object by JVM. If we try to return something from the main method, it will give compilation error as an unexpected return value.

```
package ClassExample;
public class Example1 {

        public static void main(String[] args) {
            return 0;
        }
}
```

String[] args

Java main method accepts a single argument of type String array. This is also called as java command line arguments.

Naming Convention in Java.

- Class name Should always start with Uppercase.
- Method should start with lower class.
- Package name should always be lowercase.
- Constant should be in uppercase.

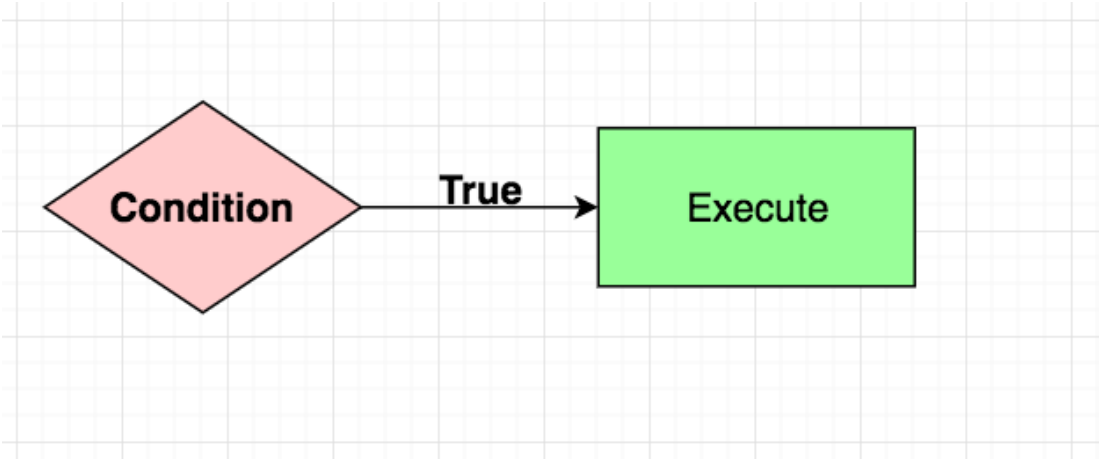
```
package classExample;
public class Example1 {
	int age;
	static final int MAX_AGE = 18;

	public int getAge() {
		return age;
	}
	public void setAge(int age) {
		this.age = age;
	}
	public static void main(String[] args) {
		Example1 obj1 = new Example1();
		obj1.setAge(4);
		obj1.getAge();

		Example1 obj2 = new Example1();
		obj1.setAge(6);
		obj2.getAge();
	}
}
```

Decision Making in Java

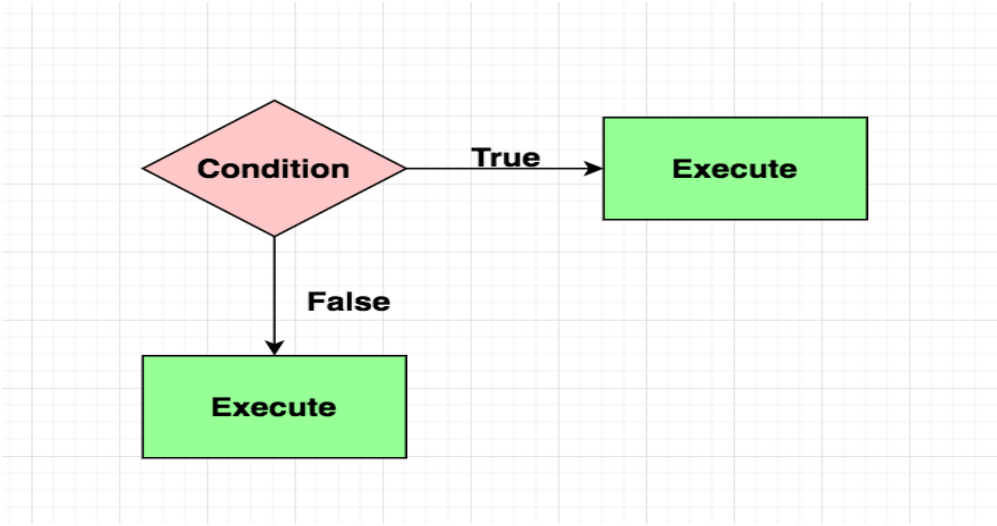
- By Java If-else
- By JAVA Switch
- By JAVA For Loop
- BY JAVA While Loop
- JAVA Break
- JAVA Continue

if statement is the most simple decision making statement. It is used to decide whether a certain statement or block of statements will be executed or not i.e if a certain condition is true then a block of statement is executed otherwise not.

If



If else

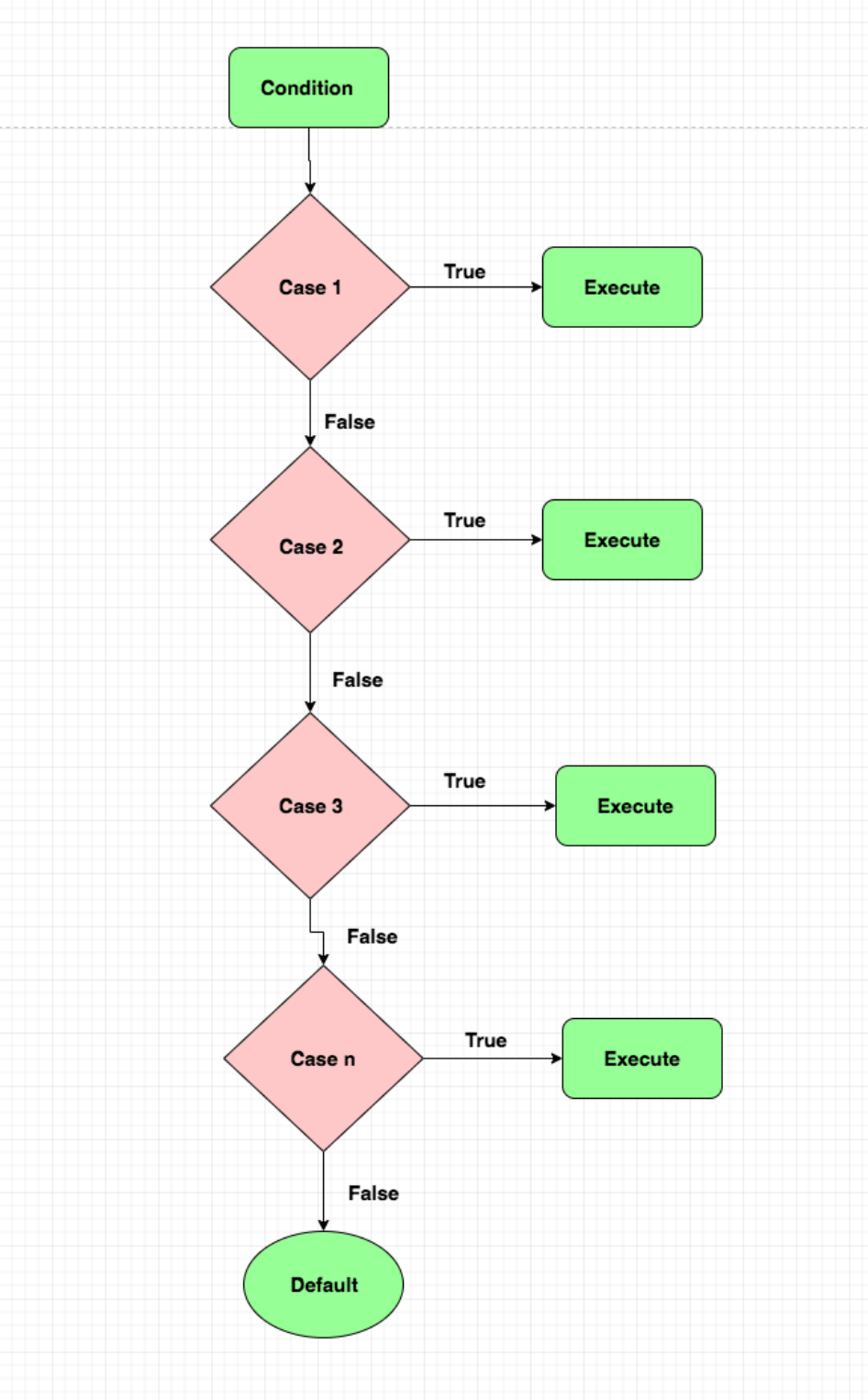



```
package controlStatements;
public class IfElseExample {

	static boolean condition;

	public static void main(String[] args) {
		if (condition) {
			System.out.println("IN IF PART");
		} else {
			System.out.println("In ELSE PART");
		}
	}
}

package controlStatements;
public class IfElseExample1 {
```

```
	public void testAge(int age) {
		if (age > 18) {
			System.out.println("Person is Major");
		} else {
			System.out.println("Person is Minor");
		}
	}
}
package controlStatements;
public class IfEleseifIF {

	public void testAge(int age) {
		if (age > 18) {
			System.out.println("Person is Major");
		} else if(age < 5) {
			System.out.println("Person is Minor");
		}

		else{
			System.out.println("Invalid data");
		}
	}
}
```

switch-case

switch-case The switch statement is a multiway branch statement. It provides an easy way to dispatch execution to different parts of code based on the value of the expression.



Syntax:

```
package controlStatements;
public class SwitchExample {
	public void checkBookTypeAndPrice(String bookName) {
		String s1;
```

```
		switch (bookName) {
		case "MATH":
			s1 = "Author Aryabhata Price=200";
			System.out.println(s1);
			break;
		case "PHYSICS":
			s1 = "Author Bhanu Prata Price=100";
			System.out.println(s1);
			break;
		case "CHEMISTRY":
			s1 = "Author Pratap Price=300";
			System.out.println(s1);
			break;
		default:
			System.out.println("No Book Found Supply Proper Input");
		}
	}

	public static void main(String[] args) {
		SwitchExample switchExample = new SwitchExample();
		switchExample.checkBookTypeAndPrice("MATH");
	}

}
package controlStatements;
public class SwitchCaseExample1 {

	public static void main(String args[]) {
		int i = 9;
		switch (i) {
		case 0:
			System.out.println("i is zero.");
			break;
		case 1:
			System.out.println("i is one.");
			break;
		case 2:
			System.out.println("i is two.");
			break;
		default:
			System.out.println("i is greater than 2.");
		}
	}
}
```

For Loop

The Java for loop is a control flow statement that iterates a part of the programs multiple times.

```
package controlStatements;
public class ForLoopExample {
public static void main(String[] args) {

		for (int i = 0; i < 8; i++) {
			String string = "Running For Loop and count is=" + i;
			System.out.println(string);
		}
	}
}
```

For Loop consists of four parts:

1. **Initialization**: It is the initial condition which is executed once when the loop starts. Here, we can initialize the variable, or we can use an already initialized variable. It is an optional condition.
2. **Condition**: It is the second condition which is executed each time to test the condition of the loop. It continues execution until the condition is false. It must return boolean value either true or false. It is an optional condition.
3. **Statement**: The statement of the loop is executed each time until the second condition is false.
4. **Increment/Decrement**: It increments or decrements the variable value. It is an optional condition.

```
package controlStatements;
public class ForLoopExample1 {

	public static void main(String[] args) {
		int array[] = { 120, 230, 404, 560, 708 };

		for (int i = 0; i < array.length; i++) {
			System.out.println(array[i]);
		}
	}
}
```

For-each Loop

```
package controlStatements;
public class ForEachLoopExample {

	public static void main(String[] args) {
		int array[] = { 120, 230, 404, 560, 708 };
```

```
		for (int i = 0; i < array.length; i++) {
			System.out.println(array[i]);
		}

		System.out.println("============");

		for (int i : array) {
			System.out.println(i);
		}
	}
}
```

Labelled For Loop

```
package controlStatements;
public class LabeledForLoop {
	public static void main(String[] args) {
		aa: for (int i = 1; i <= 5; i++) {
			bb: for (int j = 1; j <= 5; j++) {
				if (i == 2 && j == 2) {
					break aa;
				}
				System.out.println(i + " " + j);
			}
		}
	}
}
```

while loop

The Java *while loop* is used to iterate a part of the program several times. If the number of iteration is not fixed, it is recommended to use while loop.

```
package controlStatements;
public class WhileLoopExample {

	public static void main(String[] args) {

		int i = 5;
		while (i <= 10) {
			System.out.println(i);
			i++;
		}
	}
}
```

```
package controlStatements;
public class WhileExam {
```

```
	public static void main(String[] args) {
		int myNumber = 1;
		while (myNumber != 1000) {
			if ((myNumber % 2) == 0) {
				System.out.println(myNumber + " is even");
			}
			myNumber++;
		}
	}
}
```

```
package controlStatements;
public class WhileLoopExample1 {
	public static void main(String[] args) {

		while(true){
			System.out.println("while loop is running in infinitive");
		}
	}
}
```

Indefinite While Loop

Flood example, let's look at a measure of danger. If we start with a danger rate of 2% per minute, how long will it take to reach 100%? Our while loop will run as long as the total danger rate is less than 100%:

```
package controlStatements;
public class WhileLoopExample2 {
	public static void main(String[] args) {
		final double danger_rate = .02;
		int minute;
		minute = 0;
		double total_danger = 0;
		while (total_danger <= 1) {
			total_danger = danger_rate * minute;
			minute++;
		}

		System.out.println("danger hits 100% after " + minute + " minutes");
	}
}
```

Do While

```
package controlStatements;
public class DoWhileLoopExample {
	public static void main(String[] args) {
		int i = 0;
		do {
```

```
                i++;
            } while (i < 5);
        }
}
```

Java Break

```
package controlStatements;
public class BreakExample1 {

        public static void main(String[] args) {
            int i = 0;
            while (i >= 0) {
                if (i == 10) {
                    break;
                }
                System.out.println("i counter="+i);
                i++;
            }
        }
}
package controlStatements;
public class BreakExample2 {
        public static void main(String[] args) {
            int array[] = { 120, 230, 404, 560, 708 };

            for (int i = 0; i < array.length; i++) {
                System.out.println(array[i]);
                if (array[i] == 560) {
                    break;
                }
            }
        }

}
package controlStatements;
public class BreakExample2 {
        public static void main(String[] args) {
            int array[] = { 120, 230, 404, 560, 708 };

            for (int i = 0; i < array.length; i++) {
                for (int j = 0; j < array.length; j++) {
                    System.out.println(array[j]);
                    if (array[j] == 560) {
                        break;
```

```
                        }
                    }
                }
            }
}

package controlStatements;
public class BreakExample3 {
        public static void main(String[] args) {
                int array[] = { 120, 230, 404, 560, 708 };
                for (int i = 0; i < array.length; i++) {
                        for (int j = 0; j < array.length; j++) {
                                System.out.println(array[j]);
                                if (array[j] == 560) {
                                        break;
                                }
                        }
                        break;
                }
        }

}
```

Java Continue Statement

The Java Continue statement is one of the most useful statement to controls the flow of loops. We generally use this statement inside the For Loop, While Loop and Do While Loop. While executing these loops, if compiler find the Java Continue statement inside them, it will stop the current loop iteration and starts the new iteration from the beginning.

```
package controlStatements;
public class ContinueExample1 {

        public static void main(String[] args) {
                int array[] = { 120, 230, 404, 560, 708, 90, 10, 20 };
                for (int i = 0; i < array.length; i++) {
                        if (array[i] < 560) {
                                continue;
                        }
                        System.out.println("number is=" + array[i]);
                }
        }
}

package controlStatements;
public class ContinueExample2 {
```

```
	public static void main(String[] args) {
		for (int i = 0; i < 100; i++) {
			if (i > 10) {
				System.out.println("Skipped by continue value is=" + i);
				continue;
			}
			System.out.println("value of i is=" + i);
		}
	}
}
```

Java Variables

Java Variable is place holder which holds the data of variable in memory. There are three types of variables.

- Local
- Instance
- Static

```
package VariableDataType;
public class Example1 {
	public int a = 90;

	public String s1 = "Test";

	public double d1 = 90.80;

	public float f1 = 90.80f;

	public long l1 = 90l;

	public static int b = 80;

	public static String s2 = "Test1";

	public void test(){
		int a = 80;
		System.out.println(a);
	}

	public void test1(int a){
		System.out.println(a);
	}
}
```

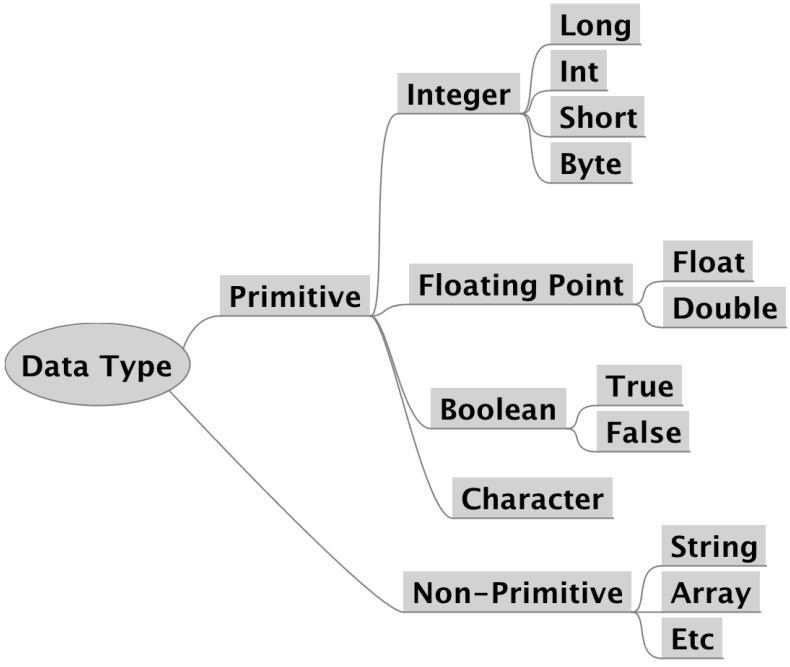
By Bhanu Pratap Singh
https://www.youtube.com/user/MrBhanupratap29/playlists
https://www.facebook.com/learnbybhanupratap/
https://www.udemy.com/seleniumbybhanu/

Data Type

Data types specify the different sizes and values that can be stored in the variable. There are two types of data types in Java:

1. Primitive data types: The primitive data types include boolean, char, byte, short, int, long, float and double.
2. Non-primitive data types: The non-primitive data types include Classes, Interfaces, and Arrays.

Data Type	Default Value	Default size
boolean	FALSE	1 bit
char	'\u0000'	2 byte
byte	0	1 byte
short	0	2 byte
int	0	4 byte
long	0L	8 byte
float	0.0f	4 byte
double	0.0d	8 byte



By Bhanu Pratap Singh
https://www.youtube.com/user/MrBhanupratap29/playlists
https://www.facebook.com/learnbybhanupratap/
https://www.udemy.com/seleniumbybhanu/

```
public class DataType {

	public static void main(String[] args) {
		System.out.println(Byte.MIN_VALUE);
		System.out.println(Byte.MAX_VALUE);
		System.out.println(Short.MIN_VALUE);
		System.out.println(Short.MAX_VALUE);
		System.out.println(Integer.MIN_VALUE);
		System.out.println(Integer.MAX_VALUE);
		System.out.println(Long.MIN_VALUE);
		System.out.println(Long.MAX_VALUE);
		System.out.println(Double.MIN_VALUE);
		System.out.println(Double.MAX_EXPONENT);
		System.out.println(Double.MAX_VALUE);
	}
}
```

byte: The byte data type is an 8-bit signed two's complement integer. The byte data type is useful for saving memory in large arrays.

- Size: 8-bit
- Value: -128 to 127

short: The short data type is a 16-bit signed two's complement integer. Similar to byte, use a short to save memory in large arrays, in situations where the memory savings actually matters.

- **Size:** 16 bit
- **Value:** -32,768 to 32,767 (inclusive)

int
It is a 32-bit signed two's complement integer.

- **Size:** 32 bit
- **Value:** -2^{31} to $2^{31}-1$
- -2,147,483,648 to 2,147,483, 647

long: The long data type is a 64-bit two's complement integer.

- Size: 64 bit
- Value: -2^{63} to $2^{63}-1$.

-

Floating point Numbers : float and double
float: The float data type is a single-precision 32-bit **IEEE 754** floating point. Use a float (instead of double) if you need to save memory in large arrays of floating point numbers.

- **Size:** 32 bits
- **Suffix :** F/f Example: 9.8f

double: The double data type is a double-precision 64-bit IEEE 754 floating point. For decimal values, this data type is generally the default choice.
Note: Both float and double data types were designed especially for scientific calculations, where approximation errors are acceptable. If accuracy is the most prior concern then, it is recommended not to use these data types and use **BigDecimal** class instead.

Basically **single precision** floating point arithmetic deals with 32 bit floating point numbers whereas **double precision** deals with 64 bit. The number of bits in **double precision** increases the maximum value that can be stored as well as increasing the **precision**

char
The char data type is a single 16-bit Unicode character. A char is a single character.

- Value: '\u0000' (or 0) to '\uffff' 65535

Java Return Type

Java requires that a method declare the data type of the value that it returns. If a method does not returna value, it must be declared to return void .

```
package ReturnTypeInJava;
public class Example1 {
	private double d = 90.980;
	private String s1 = "Test";
	private int i = 90;
	private float f = 2.80f;
	private long l = 2l;
	private char c = 'a';
	private boolean b = true;

	private int[] array = {20,30,40};

	private Example1 example1;
```

```
	public double getD() {
		return d;
	}

	public String getS1() {
		return s1;
	}

	public int getI() {
		return i;
	}

	public float getF() {
		return f;
	}

	public long getL() {
		return l;
	}

	public char getC() {
		return c;
	}

	public boolean isB() {
		return b;
	}

	public Example1 getExample1() {
		return example1;
	}


	public int[] getArray() {
		return array;
	}
}
```

Constructors in Java

Constructors are used to initialize the object's state. Like methods, a constructor also contains **collection of statements(i.e. instructions)** that are executed at time of Object creation.

By Bhanu Pratap Singh
https://www.youtube.com/user/MrBhanupratap29/playlists
https://www.facebook.com/learnbybhanupratap/
https://www.udemy.com/seleniumbybhanu/

constructors are used to assign values to the class variables at the time of object creation, either explicitly done by the programmer or by Java itself (default constructor).

When is a Constructor called ?
Each time an object is created using **new()** keyword at least one constructor (it could be default constructor) is invoked to assign initial values to the **data members** of the same class.

Rules for creating Java constructor
There are two rules defined for the constructor.

1. Constructor name must be the same as its class name
2. A Constructor must have no explicit return type
3. A Java constructor cannot be abstract, static, final, and synchronized

Types of Java constructors
There are two types of constructors in Java:

- Default constructor (no-arg constructor)
- Parameterized constructor

Default Constructor

```
package constructor;
public class Example1 {
	public String name;
	public int i;

	Example1() {
	}

	public static void main(String[] args) {
		Example1 obj = new Example1();
		System.out.println(obj.name);
		System.out.println(obj.i);
	}
}
```

Parameterized constructor

```
package constructor;
public class Book {
	int length;
	int breadth;
	int height;

	public Book(int length, int breadth, int height) {
		this.length = length;
```

```
		this.breadth = breadth;
		this.height = height;
	}

	public static void main(String[] args) {
		Book obj = new Book(10, 20, 30);
		System.out.println(obj.length);
		System.out.println(obj.breadth);
		System.out.println(obj.height);
	}
}
```

Key point to learn about constructor.

```
package constructor;
public class Exemple2 {
	int i ;
	public Exemple2(int i) {
		this.i = i;
	}

	public static void main(String[] args) {
		Exemple2 obj = new Exemple2();
	}
}
package constructor;
public class Example3 {
	int i;

	public Example3(int i) {
		this.i = i;
		System.out.println("Parameterized");
	}

	Example3() {
   System.out.println("default");
	}

	public static void main(String[] args) {
		Example3 obj = new Example3();
		Example3 obj1 = new Example3(5);
	}
}


package constructor;
```

```
public class Example4 {
	private Example4(){
		System.out.println("default");
	}

	public static void main(String[] args) {
		Example4 obj = new Example4();
	}
}
package constructor;
public class Example5 {
	void Example5(){
		System.out.println("default");
	}

	public static void main(String[] args) {
		Example5 obj = new Example5();
	}
}
```

How constructors are different from methods in Java?

- Constructor(s) must have the same name as the class within which it defined while it is not necessary for the method in java.
- Constructor(s) do not return any type while method(s) have the return type or **void** if does not return any value.
- Constructor is called only once at the time of Object creation while method(s) can be called any numbers of time.

```
package constructor;
public class Example6 {
	Example6() {
		super();
	}

	public static void main(String[] args) {

	}

}
package constructor;
public class Example7 {
	Example7() {
		System.out.println("I am Example7()");
	}

	Example7(int i) {
```

```
		this();
		System.out.println("I am Example7(int i)");
	}

	public static void main(String[] args) {
		Example7 obj = new Example7(3);
	}
}
package constructor;
public class Example8 {
	Example8() {
		System.out.println("I am Example7()");
	}

	Example8(int i) {
		this();
		System.out.println("I am Example7(int i)");
	}

	Example8(int i, int j) {
		this(5);
		System.out.println("I am Example8(int i,int j) ");
	}

	public static void main(String[] args) {
		Example8 obj = new Example8(3,6);
	}
}
package constructor;
public class Example9 {
	private String name;
	private int age;
	private String state;

	public Example9(){

	}
	public Example9(String name, int age, String state) {
		this.name = name;
		this.age = age;
		this.state = state;
	}vg
	public void display(){
```

```
		System.out.println("Example9 [name=" + name + ", age=" + age + ", state=" +
state + "]");
	}

	public static void main(String[] args) {
		Example9 obj2 = new Example9();
		obj2.display();

		Example9 obj = new Example9("test", 23, "B");
		obj.display();

		Example9 obj1 = new Example9("test1", 24, "A");
		obj1.display();
	}
}
package constructor;
public class Example10 {

	private String name;
	private int age;
	private String state;

	public Example10(String name, int age, String state) {
		this.name = name;
		this.age = age;
		this.state = state;
	}

	public Example10(int age, String name, String state) {
		this.name = name;
		this.age = age;
		this.state = state;
	}

	public static void main(String[] args) {

	}
}
```

Static keyword in java

static is a non-access modifier in Java which is applicable for the following:

1. blocks
2. variables

3. methods
4. nested classes

To create a static member(block,variable,method,nested class), precede its declaration with the keyword *static*. When a member is declared static, it can be accessed before any objects of its class are created, and without reference to any object.

```
package staticBlock;
public class Example1 {
	private static int a;
	public static void test1() {

	}
	static {
		System.out.println("this is sttaic block");
	}
}
```

Static Blocks

If you want to do some calculation in order to initialise the static variables, we can declare them in static block, and static block gets executed only once per class, when the class is first loaded.

```
package staticBlock;
public class Example2 {

	static int i = 10;
	static int j;

	static {
		System.out.println("Static block initialized.");
		j = i * 2;
	}

	public static void main(String[] args) {
		System.out.println("Value of i : " + i);
		System.out.println("Value of j : " + j);
	}
}
```

Static Variable

When you declare variable as static, then only one copy of variable is created and shared among the object. Static variable is always class variable or global variable.

- We can create static variables at class-level only.
- static block and static variables are executed in order they are present in a program.

```
package staticBlock;
```

```
public class Example3 {
	static int j = 80;

	public static void test() {
		int i = 90;
		System.out.println(i);
	}

	public static void test1() {
		static int i = 90;
		System.out.println(i);
	}

	public static void test2() {
		System.out.println(j);
	}
}

package staticBlock;
public class Example4 {

	static {
		System.out.println("static block");
	}

	static int i = 0;
	static {
		System.out.println("value of i=" + i);
		i = 90;
	}

	static {
		System.out.println("value of i=" + i);
	}

	public static void main(String[] args) {

	}
}

package staticBlock;
public class Example5 {

	static int k = 90;
```

```
	void display() {
		System.out.println("value of k is:=" + k);
	}

	public static void main(String[] args) {
		Example5 obj = new Example5();

		obj.display();

		Example5 obj1 = new Example5();
		obj1.display();

		Example5 obj2 = new Example5();
		obj2.display();
	}

}
package staticBlock;
public class Example6 {

	int i = 90;
	static int k = 80;
	static{
		k = 100;
		i = 60;
	}

}
```

Java Static Methods

Static Methods can access class variables(static variables) without using object(instance) of the class, however non-static methods and non-static variables can only be accessed using objects.

Static methods can be accessed directly in static and non-static methods.

```
package staticBlock;
public class Example7 {

	static int i = 10;
	int b = 20;

	static void test1() {
		i = 20;
		System.out.println("from m1");
```

```
		b = 10;
		test2();
	}

	void test2() {
		i = 20;
		b = 10;
		System.out.println("from m2");
	}

	public static void main(String[] args) {
	}
}
package staticBlock;

public class Example8 {

	static void test1(){
		System.out.println("test1()");
	}

	public static void main(String[] args) {
		test1();
	}
}
package staticBlock;

public class Example9 {

	static void test1(){
		System.out.println("test1()");
	}

	void test2(){
		System.out.println("test1()");
	}

	public static void main(String[] args) {
		test1();
		test2();
	}
}
package staticBlock;
```

```
public class Example10 {

	public static void main(String[] args) {
		Example9.test1();
	}
}
```

Static Class

A class can be made **static** only if it is a nested class.

1. Nested static class doesn't need reference of Outer class
2. A static class cannot access non-static members of the Outer class

```
package staticBlock;
public class Example11 {

	private static String str = "test";

	static class MyNestedClass {
		public void disp() {
			System.out.println(str);
		}
	}

	public static void main(String[] args) {
		Example11.MyNestedClass obj = new Example11.MyNestedClass();
		obj.disp();
	}
}
```

```
package staticBlock;
public class Example12 {
	private static String str = "test";

	class MyNestedClass {
		public void disp() {
			System.out.println(str);
		}
	}

	public static void main(String[] args) {
		Example12.MyNestedClass obj = new Example12.MyNestedClass();
		obj.disp();
	}
```

```
}


package staticBlock;
public class Example12 {
	private static String str = "test";

	class MyNestedClass {
		public void disp() {
			System.out.println(str);
		}
	}

	public static void main(String[] args) {
		Example12.MyNestedClass obj = new Example12().new MyNestedClass();
		obj.disp();
	}
}
package staticBlock;

public static class Example13 {

	private static String str = "test";

	static class MyNestedClass {
		public void disp() {
			System.out.println(str);
		}
	}
}
```

When we should use static variables and Methods

Use static variable when it is common to all object and change the value of variable by using static method. E.g. in college all student will have same college name

```
package staticBlock;
public class Student {
	private static String collegeName;
	private String studentName;
	private int rollnumber;
	private static int i;

	Student(String studentName) {
		this.studentName = studentName;
		this.rollnumber = increaseRollNumber();
	}
```

```
	public static String getCollegeName() {
		return collegeName;
	}

	public static void setCollegeName(String collegeName) {
		Student.collegeName = collegeName;
	}

	public static int increaseRollNumber() {
		return i++;
	}

	public void getStudentInformation() {
		System.out.print("student Name : " + this.studentName);
		System.out.print(" roll Number : " + this.rollnumber);
		System.out.print(" college Name : " + collegeName);
		System.out.println();
	}

	public static void main(String[] args) {
		Student.setCollegeName("LS college");

		Student obj = new Student("Ram");
		obj.getStudentInformation();
		Student obj1 = new Student("Mohan");
		obj1.getStudentInformation();
	}

}
```

This keyword in java

There can be a lot of usage of **java this keyword**. In java, this is a **reference variable** that refers to the current object.

Usage of java this keyword

1. this can be used to refer current class instance variable.
2. this can be used to invoke current class method (implicitly)
3. this() can be used to invoke current class constructor.
4. this can be passed as an argument in the method call.
5. this can be passed as argument in the constructor call.
6. this can be used to return the current class instance from the method.

```
package ThisExample;
```

```
public class Example1 {
	int rollno;
	String name;
	float fee;

	Example1(int rollno, String name, float fee) {
		rollno = rollno;
		name = name;
		fee = fee;
	}

	void display() {
		System.out.println(rollno + " " + name + " " + fee);
	}

	public static void main(String args[]) {
		Example1 s1 = new Example1(111, "ankit", 5000f);
		Example1 s2 = new Example1(112, "sumit", 6000f);
		s1.display();
		s2.display();
	}
}
package ThisExample;
public class Example2 {
	int rollno;
	String name;
	float fee;

	Example2(int rollno, String name, float fee) {
		this.rollno = rollno;
		this.name = name;
		this.fee = fee;
	}

	void display() {
		System.out.println(rollno + " " + name + " " + fee);
	}

	public static void main(String args[]) {
		Example2 s1 = new Example2(111, "ankit", 5000f);
		Example2 s2 = new Example2(112, "sumit", 6000f);
		s1.display();
		s2.display();
	}
```

```
}

package ThisExample;
public class Example3 {
	void display() {
		this.show();

		System.out.println("Inside display function");
	}

	void show() {
		System.out.println("Inside show funcion");
	}

	public static void main(String args[]) {
		Example3 t1 = new Example3();
		t1.display();
	}
}
package ThisExample;
public class Example4 {
	int a;
	int b;

	Example4() {
		this(10, 20);
		System.out.println("Inside  default constructor \n");
	}

	Example4(int a, int b) {
		this.a = a;
		this.b = b;
		System.out.println("Inside parameterized constructor");
	}

	public static void main(String[] args) {
		Example4 object = new Example4();
	}
}
package ThisExample;
public class Example5 {
	void m(Example5 obj) {
		System.out.println("method is invoked");
	}
```

```
	void p() {
		m(this);
	}

	public static void main(String args[]) {
		Example5 s1 = new Example5();
		s1.p();
	}
}
package ThisExample;
public class Example6 {

	Example7 obj;

	Example6(Example7 obj) {
		this.obj = obj;
		obj.display();
	}
}
package ThisExample;
public class Example7 {

	int i = 90;

	Example7() {
		Example6 obj = new Example6(this);
	}

	void display() {
		System.out.println("Value of i in Class Example7 : " + i);
	}

	public static void main(String[] args) {
		Example7 obj = new Example7();

	}
}

package ThisExample;
public class Example8 {
	int a;
	int b;
```

```
	Example8() {
		a = 10;
		b = 20;
	}

	Example8 get() {
		return this;
	}

	void display() {
		System.out.println("a = " + a + "  b = " + b);
	}

	public static void main(String[] args) {
		Example8 object = new Example8();
		object.get().display();
	}
}
```

Inheritance in Java

As we know, a child inherits the properties from his parents. A similar concept is followed in Java, where we have two classes:

1. Parent class (Super or Base class)
2. Child class (Subclass or Derived class)

A class which inherits the properties is known as Child Class whereas a class whose properties are inherited is known as Parent class.

Inheritance represents the IS-A relationship which is also known as a *parent-child* relationship.

Why use inheritance in java

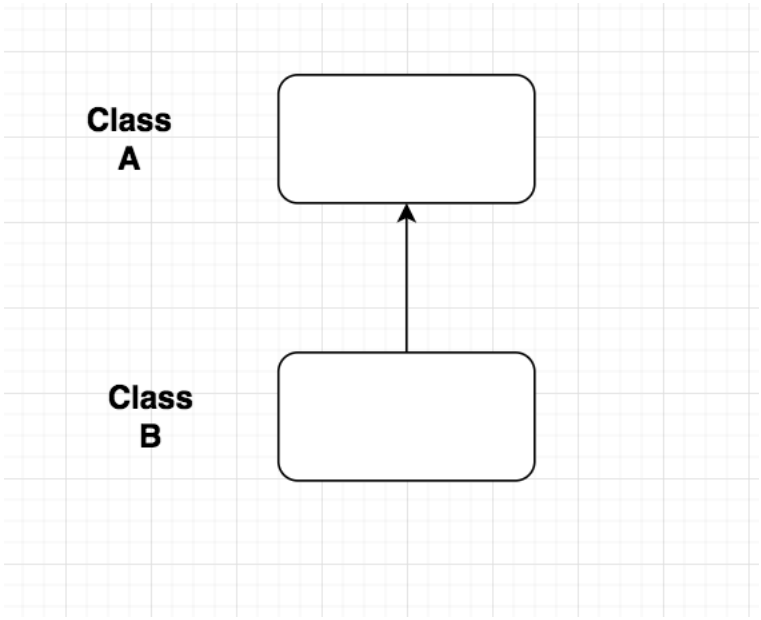
- For Method Overriding (so runtime polymorphism can be achieved).
- For Code Reusability.

Syntax

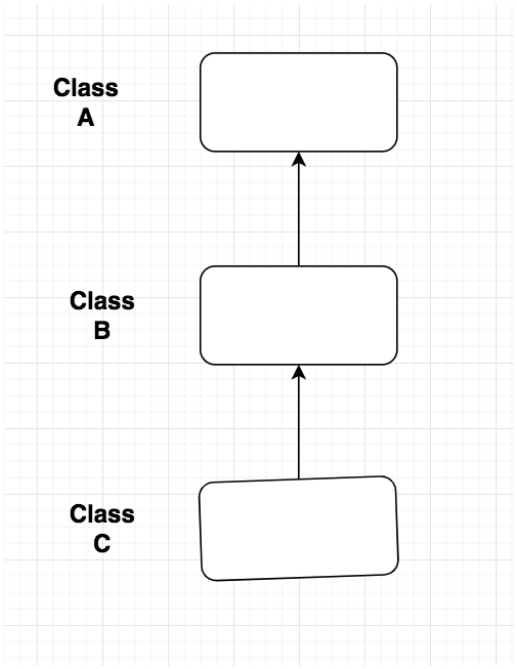
```
package inheritance;
public class Example2 extends Exaple1{

}
```

Types of inheritance in java

1. **Single level Inheritance**

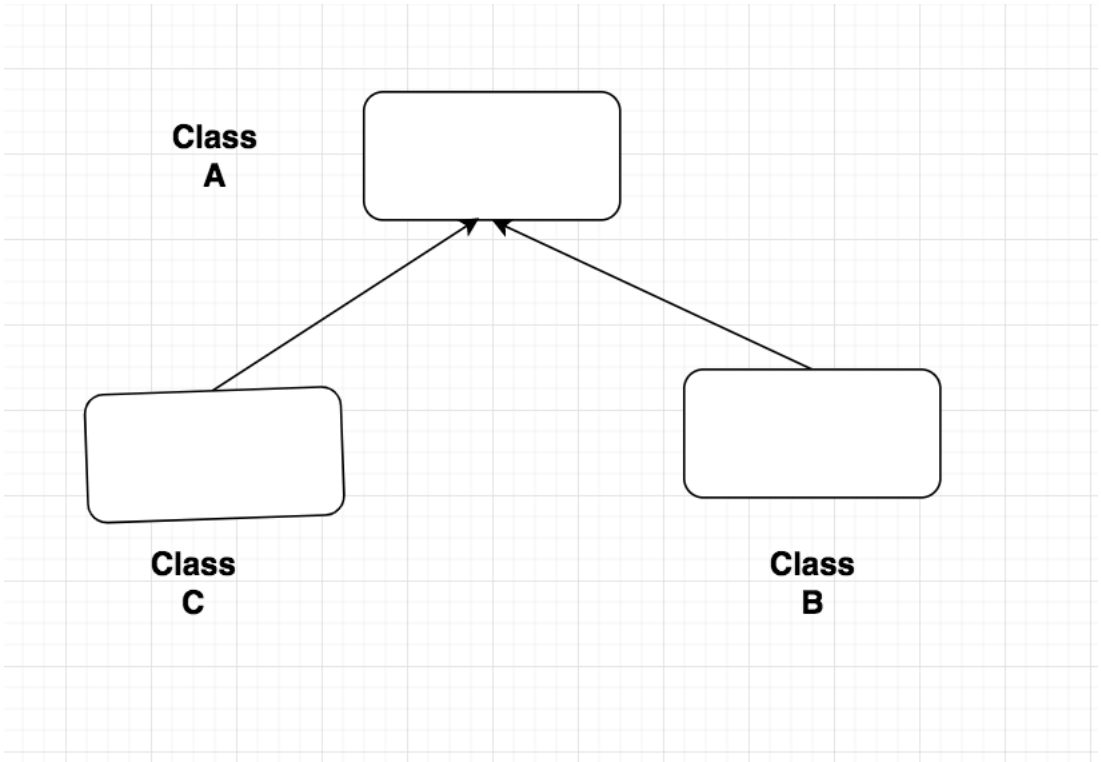


2. **Multilevel Inheritance**



3. **Hierarchical Level Inheritance**

By Bhanu Pratap Singh
https://www.youtube.com/user/MrBhanupratap29/playlists
https://www.facebook.com/learnbybhanupratap/
https://www.udemy.com/seleniumbybhanu/




```
package inheritance;
public class Student {
	private String Schooladdress;

	private String SchoolName;


	public String getSchooladdress() {
		return "Muzaffarpur Bihar 560100";
	}

	public String getSchoolName() {
		return "DAV";
	}
}
package inheritance;
public class Ram extends Student{

	int roolnumber;

	public Ram(int roolnumber) {
		this.roolnumber = roolnumber;
	}

	public void display() {
```

```
		String s1 = "Ram [roolnumber=" + roolnumber + ", getSchooladdress()=" +
getSchooladdress() + ", getSchoolName()="
				+ getSchoolName() + "]";
		System.out.println(s1);
	}

	public static void main(String[] args) {
		Ram ram = new Ram(8);
		ram.display();
	}
}
package inheritance;
public class Teacher {
	String designation = "Teacher";
	String collegeName = "L.S College Muzaffarpur";

	void does() {
		System.out.println("Teaching");
	}
}

package inheritance;

public class JavaTeacher extends Teacher {

	String mainSubject = "Spark";

	public static void main(String args[]) {
		JavaTeacher obj = new JavaTeacher();
		System.out.println(obj.collegeName);
		System.out.println(obj.designation);
		System.out.println(obj.mainSubject);
		obj.does();
	}
}
```

Multilevel Inheritance

```
package inheritance;

public class College {

	public String collegeName = "LS COLLEGE";

}
```

By Bhanu Pratap Singh
https://www.youtube.com/user/MrBhanupratap29/playlists
https://www.facebook.com/learnbybhanupratap/
https://www.udemy.com/seleniumbybhanu/

```
package inheritance;

public class ClassRoom extends College{

	public String classRoom = "10Th";
}
```

```
package inheritance;

public class Students extends ClassRoom {

	int age;
	String name;

	public Students(int age, String name) {
		this.age = age;
		this.name = name;
	}

	public static void main(String[] args) {
		Students students = new Students(10, "Mohan");
		System.out.println(students);
		Students students1 = new Students(10, "RAM");
		System.out.println(students1);
		Students students2 = new Students(10, "SOHAN");
		System.out.println(students2);
	}

	@Override
	public String toString() {
		return "Students [age=" + age + ", name=" + name + ", classRoom=" +
classRoom + ", collegeName=" + collegeName
				+ "]";
	}
}
```

Hierarchical level inheritance

```
package inheritance;

public class Vehicle {

	int speed;
```

```
	public void speed(int speed){
		this.speed = speed;
		System.out.println("Vehicle runts at:="+speed+"KM/H");
	}
}

package inheritance;

public class Car extends Vehicle{

	int price;

	String name;

	public Car(int price, String name) {
		this.price = price;
		this.name = name;
	}

	public static void main(String[] args) {
		Car car = new Car(2000,"Maruti");
		car.speed(10);
		System.out.println(car);
	}

	@Override
	public String toString() {
		return "Car [price=" + price + ", name=" + name + ", speed=" + speed + "]";
	}
}

package inheritance;

public class Jeep extends Vehicle{

	int price;

	String name;

	public Jeep(int price, String name) {
		this.price = price;
		this.name = name;
	}
```

```
	public static void main(String[] args) {
		Jeep jeep = new Jeep(1000,"Jeep");
		jeep.speed(100);
		System.out.println(jeep);
	}

	@Override
	public String toString() {
		return "Jeep [price=" + price + ", name=" + name + ", speed=" + speed + "]";
	}

}
```

Why multiple inheritance is not supported in java?

```
package inheritance;

public class Example3 {

	void msg(){

	}
}

package inheritance;

public class Example4 {

	void msg() {

	}
}

package inheritance;

public class Example6 extends Example3,Example3{

}
```

- **Private members do NOT get inherited.**
- **Constructors cannot be Inherited in Java.**

```
package inheritance;
public class Conts1 {
```

```
	int a;

	public Conts1(int a) {
		this.a = a;
	}

	Conts1() {

	}
}

package inheritance;
public class Conts2 extends Conts1{

	public static void main(String[] args) {
		Conts2 conts2 = new Conts2(4);
	}
}
```

Aggregation in Java

Aggregation in Java is a relationship between two classes that is best described as a "has-a" and "whole/part" relationship. ... The **aggregate** class contains a reference to another class and is said to have ownership of that class. Each class referenced is considered to be part-of the **aggregate** class.

Aggregation represents HAS-A relationship.

```
package inheritance;

public class Operation {

	int square(int n) {
		return n * n;
	}

	int circle(int radious) {
		return (22 / 7) * radious;
	}

	int rectangle(int length,int width) {
		return width*length;
	}

}
package inheritance;
```

```
public class Circle {

	Operation operation;

	public void calculateArea(int area){
		operation = new Operation();
		int a = operation.square(area);
		System.out.println("Circle area="+a);
	}

	public static void main(String[] args) {

		Circle circle = new Circle();
		circle.calculateArea(5);
	}

}
```

When to use Aggregation?

- Code reuse is also best achieved by aggregation when there is no is-a relationship.
- Inheritance should be used only if the relationship is-a is maintained throughout the lifetime of the objects involved; otherwise, aggregation is the best choice.

Understanding meaningful example of Aggregation

Employee has an object of Address, address object contains its own informations such as city, state, country etc. In such case relationship is Employee HAS-A address.

```
package inheritance;
public class Address {

	String city, state, country;

	public Address(String city, String state, String country) {
		this.city = city;
		this.state = state;
		this.country = country;
	}
}
```

```
package inheritance;

public class Employee {

	int id;
	String name;
```

```
	Address address;

	public Employee(int id, String name, Address address) {
		this.id = id;
		this.name = name;
		this.address = address;
	}

	void display() {
		System.out.println(id + " " + name);
		System.out.println(address.city + " " + address.state + " " + address.country);
	}

	public static void main(String[] args) {
		Address address1 = new Address("test1", "BIHAR", "india");
		Address address2 = new Address("test2", "BIHAR", "india");

		Employee e = new Employee(11, "Ram", address1);
		Employee e2 = new Employee(12, "Mohan", address2);

		e.display();
		e2.display();

	}
}
```

Java Polymorphism

Method Overloading is a feature that allows a class to have more than one method having the same name, if their argument lists are different. It is similar to constructor overloading in Java, that allows a class to have more than one constructor having different argument lists.

How to perform method overloading in Java?

```
package methodOverloading;
public class Overloading1 {

	private static void display(int a) {
		System.out.println("Arguments: " + a);
	}

	private static void display(int a, int b) {
		System.out.println("Arguments: " + a + " and " + b);
	}

	public static void main(String[] args) {
```

```
		display(1);
		display(1, 4);
	}
}
```

By changing the datatype of parameters

```
package methodOverloading;
public class Overloading2 {

	private static void display(int a) {
		System.out.println("Got Integer data.");
	}

	private static void display(String a) {
		System.out.println("Got String object.");
	}

	public static void main(String[] args) {
		display(1);
		display("Hello");
	}
}
```

```
package methodOverloading;
public class Overloading3 {
	private String formatNumber(int value) {
		return String.format("%d", value);
	}

	private String formatNumber(double value) {
		return String.format("%.3f", value);
	}

	private String formatNumber(String value) {
		return String.format("%.2f", Double.parseDouble(value));
	}

	public static void main(String[] args) {
		Overloading3 hs = new Overloading3();
		System.out.println(hs.formatNumber(500));
		System.out.println(hs.formatNumber(89.9934));
		System.out.println(hs.formatNumber("550"));
```

```
    }
}
```

%f	**Decimal floating-point**
%d	**Decimal integer**

Important Points

- Two or more methods can have same name inside the same class if they accept different arguments. This feature is known as method overloading.
- Method overloading is achieved by either:
  - changing the number of arguments.
  - or changing the datatype of arguments.
- Method overloading is not possible by changing the return type of methods.

Why method overloading?

Suppose, you have to perform addition of the given numbers but there can be any number of arguments (let's say either 2 or 3 arguments for simplicity).
In order to accomplish the task, you can create two methods `sum1num(int, int)` and `sum2num(int, int, int)` for two and three parameters respectively. However, other programmers as well as you in future may get confused as the behaviour of both methods is same but they differ by name.
The better way to accomplish this task is by overloading methods. And, depending upon the argument passed, one of the overloaded methods is called. This helps to increase readability of the program

Why Method Overloading is not possible by changing the return type of method only?

```
package methodOverloading;
public class Overloading4 {

	static int add(int a, int b) {
		return a + b;
	}

	static double add(int a, int b) {
		return a + b;
	}
```

```
}

class TestOverloading3 {
	public static void main(String[] args) {
		System.out.println(Adder.add(11, 11));// ambiguity
	}
}
```

Compile Time Error: method add(int,int) is already defined in class Overloading4

Can we overload java main() method?

Yes, by method overloading. You can have any number of main methods in a class by method overloading. But JVM calls main() method which receives string array as arguments only

```
package methodOverloading;

public class Overloading5 {

	public static void main(String[] args) {
		System.out.println("main with String[]");
	}

	public static void main(String args) {
		System.out.println("main with String");
	}

	public static void main() {
		System.out.println("main without args");
	}
}

package methodOverloading;

public class Overloading6 {

	void sum(int a, long b) {
		System.out.println(a + b);
	}

	void sum(int a, int b, int c) {
		System.out.println(a + b + c);
	}

}
```

Method Overloading with Type Promotion in case of ambiguity

```
package methodOverloading;

public class Overloading7 {

	void sum(int a, long b) {
		System.out.println("a method invoked");
	}

	void sum(long a, int b) {
		System.out.println("b method invoked");
	}

	public static void main(String args[]) {
		Overloading7 obj = new Overloading7();
		obj.sum(20, 20);
	}
}
```

When parent class has only parameterised constructor Then child class has to explicitly call the parent class parameterised constructor.

```
package methodOverloading;

public class Overloading8 {

	Overloading8(int i) {

	}
}
```

```
package methodOverloading;

public class Overloading9 extends Overloading8 {

	Overloading9(int i) {
		super(i);
	}

	public static void main(String[] args) {

	}
}
```

By Bhanu Pratap Singh
https://www.youtube.com/user/MrBhanupratap29/playlists
https://www.facebook.com/learnbybhanupratap/
https://www.udemy.com/seleniumbybhanu/

When parent class has parameterised and default constructor Then child class has not explicitly call to the parent class constructor. Since Java compiler will keep spuer() as first line of constructor.

```
package methodOverloading;

public class Overloading10 {

	Overloading10(int i) {

	}

	Overloading10() {

	}
}
package methodOverloading;

public class Overloading11 extends Overloading10{

	Overloading11(){

	}

}
```

Method Overriding in Java

If subclass (child class) has the same method as declared in the parent class, it is known as method overriding in Java.

Usage of Java Method Overriding

1. Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.
2. Method overriding is used for runtime polymorphism

Rules for Java Method Overriding

1. The method must have the same parameter as in the parent class.
2. The method must have the same name as in the parent class
3. There must be an IS-A relationship (inheritance).

```
package methodOverriding;

public class Overriding1 {
```

```
	public void test(){
		System.out.println("This is test implementation");
	}

	public void test1(){
		System.out.println("This is test implementation test1");
	}

}
package methodOverriding;
public class Overriding2 extends Overriding1{

	public void test(){
		System.out.println("This is test implementation-1");
	}

	public static void main(String[] args) {
		Overriding2 Overriding2 = new Overriding2();
		Overriding2.test();

		Overriding2.test1();
	}
}

package methodOverriding;
public class Vehicle {

	public void run() {
		System.out.println("runs at 80KM/H");
	}
}
package methodOverriding;
public class Car extends Vehicle{

	public void run(){
		System.out.println("runs at 180KM/H");
	}
}
package methodOverriding;

public class Jeep  extends Vehicle{

}
```

By Bhanu Pratap Singh
https://www.youtube.com/user/MrBhanupratap29/playlists
https://www.facebook.com/learnbybhanupratap/
https://www.udemy.com/seleniumbybhanu/

```
package methodOverriding;
public class Bike  extends Vehicle{

}
package methodOverriding;
public class Bank {

	public int rateOfInterest(){
		System.out.println("rateOfInterest= 0%");
		return 0;
	}
}
package methodOverriding;

public class HDFCBank extends Bank {

	public int rateOfInterest() {
		System.out.println("rateOfInterest= 8.5%");
		return 0;
	}
}
package methodOverriding;

public class ICICI extends Bank{

}
package methodOverriding;

public class SBI extends Bank {

	public int rateOfInterest() {
		System.out.println("rateOfInterest= 7.5%");
		return 0;
	}
}
```

Can we override static method?

No, Since static members are class members.

It is because the static method is bound with class whereas instance method is bound with an object. Static belongs to the class area, and an instance belongs to the heap area.

```
package methodOverriding;

public class Example1 {

	private static void test1() {
```

```
		System.out.println("Example1");
	}

	private static void test2() {
		System.out.println("Example1 test2");
	}
}

package methodOverriding;

public class Example2 extends Example1 {

	private static void test1() {
		System.out.println("Example2");
	}

	public static void main(String[] args) {
		Example2.test1();

		// Example2.test2();
	}
}
```

Can we override java main method?

No, because the main is a static method.

Cam we Override private methods?

No, Because private is only to class.

Interface in Java

Points to remember

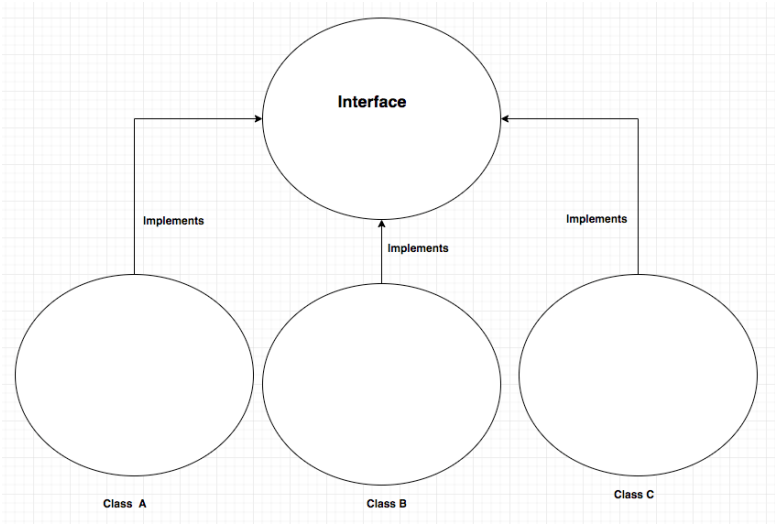
- You cannot instantiate an interface.
- An interface does not contain any constructors.
- All of the methods in an interface are abstract.
- An interface fields that can appear in an interface must be declared both static and final.

```java
package interfaceInjava;

public interface A {

    int i = 90;
    public static final int j = 80;

    void test1();

    void test2();
}
```

- An interface is not extended by a class; it is implemented by a class.
- An interface can extend multiple interfaces.
- All the methods are public and abstract. And all the fields are public, static, and final.
- It is used to achieve multiple inheritance.

```java
package interfaceInjava;

public interface A {

}

public class B implements A {

}
```

By Bhanu Pratap Singh
https://www.youtube.com/user/MrBhanupratap29/playlists
https://www.facebook.com/learnbybhanupratap/
https://www.udemy.com/seleniumbybhanu/



New features added in interfaces in JDK 8

1. Prior to JDK 8, interface could not define implementation. We can now add default implementation for interface methods. This default implementation has special use and does not affect the intention behind interfaces.

   Suppose we need to add a new function in an existing interface. Obviously the old code will not work as the classes have not implemented those new functions. So with the help of default implementation, we will give a default body for the newly added functions. Then the old codes will still work.

2. Another feature that was added in JDK 8 is that we can now define static methods in interfaces which can be called independently without an object. Note: these methods are not inherited.

```java
package interfaceInjava;

public interface A {
    // in java 8

    default int test3() {
        return 0;
    }

    default void test4() {
    }

    static void test5() {

    }
}
```

New features added in interfaces in JDK 9

From Java 9 onwards, interfaces can contain following also

1. Static methods
2. Private methods
3. Private Static methods

Syntax

```java
package interfaceInjava;

public interface Example1 {

        int i = 90;

        void test1();

}
```

To implement an interface we use keyword: implement

By Bhanu Pratap Singh
https://www.youtube.com/user/MrBhanupratap29/playlists
https://www.facebook.com/learnbybhanupratap/
https://www.udemy.com/seleniumbybhanu/

```java
package interfaceInjava;

public interface Example2 {

	int i = 90;

	void test1();

	void test2();
}
```

```java
package interfaceInjava;
public class ImplementExample2 implements Example2{

	@Override
	public void test1() {
		// TODO Auto-generated method stub

	}

	@Override
	public void test2() {
		// TODO Auto-generated method stub

	}

}
```

An interface does not contain any constructors.

```java
package interfaceInjava;
public interface Example3 {


	Example3(){

	}
}
```

An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.

```java
package interfaceInjava;
```

```
public interface Example4 {

	Example4 example4;

	int i;

}
```

An interface can extend multiple interfaces.

```
package interfaceInjava;
public interface Example5 {

	void method5();
}

package interfaceInjava;
public interface Example6 {

	void method6();
}

package interfaceInjava;
public interface Example7 extends Example5, Example6{

}
```

Default and static method in interface

```
package interfaceInjava;
public interface Example8 {

	void method6();

	default void test1() {
		System.out.println("I am test1");
	}

	static void test2() {
		System.out.println("I am test2");
	}
}
```

When class implements interface which extends more than one interface.

```
package interfaceInjava;
```

```java
public interface Example5 {

	void method5();
}
```

```java
package interfaceInjava;
public interface Example6 {

	void method6();
}
```

```java
package interfaceInjava;
public interface Example7 extends Example5, Example6{

}
```

```java
package interfaceInjava;
public class TestExample7 implements Example7{

	@Override
	public void method5() {
		// TODO Auto-generated method stub

	}

	@Override
	public void method6() {
		// TODO Auto-generated method stub

	}

}
```

Interface Real Time Example

```java
package interfaceInjava;
public interface Vehicle {

	void changeGear(int a);

	void speedUp(int a);

	void applyBrakes(int a);
}
```

```java
package interfaceInjava;
```

```
public class Car implements Vehicle {

	int speed;
	int gear;

	@Override
	public void changeGear(int a) {
		gear = a;
	}

	@Override
	public void speedUp(int a) {

		speed = a;
	}

	@Override
	public void applyBrakes(int a) {

		speed = speed - a;
	}

	public void display() {
		System.out.println("speed: " + speed + " gear: " + gear);
	}
}

package interfaceInjava;
public class Bike implements Vehicle {

	int speed;
	int gear;

	@Override
	public void changeGear(int a) {
		gear = a;
	}

	@Override
	public void speedUp(int a) {

		speed = a;
	}
```

```
	@Override
	public void applyBrakes(int a) {

		speed = speed - a;
	}

	public void display() {
		System.out.println("speed: " + speed + " gear: " + gear);
	}

}
package interfaceInjava;
public class TestVehicle {

	public static void main(String[] args) {

		Car car = new Car();
		car.changeGear(3);
		car.speedUp(4);
		car.applyBrakes(1);

		System.out.println("Bicycle present state:");
		car.display();

		Bike bike = new Bike();
		bike.changeGear(2);
		bike.speedUp(4);
		bike.applyBrakes(1);

		System.out.println("Bike present state:");
		bike.display();
	}

}
```

Java 8 Interface Features

From Java 8, apart from public abstract methods, you can have public static methods and public default methods.

```
package interfaceInjava;
public interface Exampl3 {

	default void test1(){
```

```
		System.out.println("I am default");
	}

	static void test2(){
		System.out.println("I am static");
	}

}
package interfaceInjava;
public class Exampl3Test implements Exampl3{

	public static void main(String[] args) {
		Exampl3.test2();

		Exampl3Test obj = new Exampl3Test();
		obj.test1();
	}

}
```

Private methods since java 9

Since java 9, you will be able to add private methods and private static method in interfaces.

These private methods will improve code re-usability inside interfaces. Foe example, if two default methods needed to share code, a private interface method would allow them to do so, but without exposing that private method to it's implementing classes.

Using private methods in interfaces have four rules :

1. Private interface method cannot be abstract.
2. Private method can be used only inside interface.
3. Private static method can be used inside other static and non-static interface methods.
4. Private non-static methods cannot be used inside private static methods.

```
public interface CustomInterface {

  public abstract void method1();

  public default void method2() {
    method4();  //private method inside default method
    method5();  //static method inside other non-static method
    System.out.println("default method method2()");
```

```
    }

    public static void method3() {
        method5(); //static method inside other static method
        System.out.println("static method method3()");
    }

    private void method4(){
        System.out.println("private method method4()");
    }

    private static void method5(){
        System.out.println("private static method method5()");
    }
}

public class CustomClass implements CustomInterface {

    @Override
    public void method1() {
        System.out.println("abstract method");
    }

    public static void main(String[] args){
        CustomInterface instance = new CustomClass();
        instance.method1();
        instance.method2();
        CustomInterface.method3();
    }
}
```

Abstract classes

1. An abstract class is a class that is declared with abstract keyword.
2. An abstract method is a method that is declared without an implementation.
3. An abstract class may or may not have all abstract methods. Some of them can be concrete methods
4. A method defined abstract must always be redefined in the subclass, thus making overriding compulsory OR either make subclass itself abstract.
5. Any class that contains one or more abstract methods must also be declared with abstract keyword.
6. There can be no object of an abstract class. That is, an abstract class cannot be directly instantiated with the *new operator*.
7. An abstract class can have parametrized constructors and default constructor is always present in an abstract class.

Syntax

```
package abstractInJava;
public abstract class Example1 {

	void test1() {

	}

	abstract void test2();

}
```

Abstract members are not by default public static final

```
package abstractInJava;
public abstract class Example2 {

	int i = 80;

	public static final int j = 40;

	abstract void test1();

	public abstract void test2();

}
```

Use of abstract class

```
package abstractInJava;
public abstract class Person {

	private String name;
	private String gender;

	public Person(String nm, String gen) {
		this.name = nm;
		this.gender = gen;
	}

	public abstract void work();

	@Override
	public String toString() {
		return "Name=" + this.name + "::Gender=" + this.gender;
	}
```

```
	public void changeName(String newName) {
		this.name = newName;
	}
}

package abstractInJava;

public class Employee extends Person {

	private int empId;

	public Employee(String nm, String gen, int id) {
		super(nm, gen);
		this.empId = id;
	}

	@Override
	public void work() {
		if (empId == 0) {
			System.out.println("Not working");
		} else {
			System.out.println("Working as employee");
		}
	}

	public static void main(String args[]) {
		Person student = new Employee("Test1", "Female", 0);
		Person employee = new Employee("Test2", "Male", 12);
		student.work();
		employee.work();
		employee.changeName("Test3");
		System.out.println(employee.toString());
	}

}
```

Can we extend more than one abstract class?
Ans: NO

Why can't we create the object of an abstract class?

Because these classes are incomplete, they have abstract methods that have no body so if java allows you to create object of this class then if someone calls the abstract method using that object then What would happen? There would be no actual implementation of the

method to invoke.
Also because an object is concrete. An abstract class is like a template, so you have to extend it and build on it before you can use it.

Can we create constructor in abstract class

Ans: Yes

```
package abstractInJava;
public abstract class Example3 {
	int i;

	String name;

	public Example3(int i, String name) {
		this.i = i;
		this.name = name;
	}

	Example3() {

	}
}
```

Abstract Vs Interface

Abstract class	Interface
1) Abstract class can **have abstract and non-abstract**methods.	Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also.
2) Abstract class **doesn't support multiple inheritance**.	Interface **supports multiple inheritance**.
3) Abstract class **can have final, non-final, static and non-static variables**.	Interface has **only static and final variables**.
4) Abstract class **can provide the implementation of interface**.	Interface **can't provide the implementation of abstract class**.
5) The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.
6) An **abstract class** can extend another Java class and implement multiple Java interfaces.	An **interface** can extend another Java interface only.

7) An **abstract class** can be extended using keyword "extends".	An **interface** can be implemented using keyword "implements".
8) A Java **abstract class** can have class members like private, protected, etc.	Members of a Java interface are public by default.

9)**Example:**
public abstract class Shape{
public abstract void draw();
}
	Example:
	public interface Drawable{
	void draw();
	}

Abstract Classes Compared to Interfaces

Abstract classes are similar to interfaces. You cannot instantiate them, and they may contain a mix of methods declared with or without an implementation. However, with abstract classes, you can declare fields that are not static and final, and define public, protected, and private concrete methods. With interfaces, all fields are automatically public, static, and final, and all methods that you declare or define (as default methods) are public. In addition, you can extend only one class, whether or not it is abstract, whereas you can implement any number of interfaces.

Which should you use, abstract classes or interfaces?

- Consider using abstract classes if any of these statements apply to your situation:
  - You want to share code among several closely related classes.
  - You expect that classes that extend your abstract class have many common methods or fields, or require access modifiers other than public (such as protected and private).
  - You want to declare non-static or non-final fields. This enables you to define methods that can access and modify the state of the object to which they belong.
- Consider using interfaces if any of these statements apply to your situation:
  - You expect that unrelated classes would implement your interface. For example, the interfaces Comparable and Cloneable are implemented by many unrelated classes.
  - You want to specify the behaviour of a particular data type, but not concerned about who implements its behaviour.
  - You want to take advantage of multiple inheritance of type.

Example

```
package interfaceInjava;
public class Eample100 implements Example101 {
```

```
	@Override
	public void test1() {

	}

	public void test5() {

	}

	public static void main(String[] args) {
		Example101 obj = new Eample100();

		obj.test1();
		obj.test2();
		obj.test3();

		Example102 obj1 = new Eample100();
		obj1.test3();

		Eample100 obj2 = new Eample100();
		obj2.test5();

	}

	@Override
	public void test3() {
		// TODO Auto-generated method stub

	}

}
package interfaceInjava;
public interface Example101 extends Example102{

	void test1();

	default void test2(){
		System.out.println("default method");
	}
}
package interfaceInjava;

public interface Example102 {
```

```
        void test3();
}
```

Access Modifier

	With In Class Same package	Outside Class Same package	Outside Class Different package Without Inheritance	Outside Class Different package Using Inheritance
Private	**YES**	**NO**	**NO**	**NO**
public	**YES**	**YES**	**YES**	**YES**
protected	**YES**	**YES**	**NO**	**YES**
default	**YES**	**YES**	**NO**	**NO**

```
package accessmodifier;
public class Example1 {

        public String s1;

        private String s2;

        protected String s3;

        String s4;

        protected void name1() {
                System.out.println("protected method");
        }

        public void name2() {
                System.out.println("public method");
        }

        private void name3() {
                System.out.println("private method");
        }

        void name4() {
                System.out.println("default method");
        }

}
```

What are the members we can access in different class, within same package?

By Bhanu Pratap Singh
https://www.youtube.com/user/MrBhanupratap29/playlists
https://www.facebook.com/learnbybhanupratap/
https://www.udemy.com/seleniumbybhanu/

```
package accessmodifier;
public class Example2 {

	public static void main(String[] args) {

		Example1 example1 = new Example1();

		example1.name1();
		example1.name2();
		example1.name4();

		System.out.println(example1.s1);
		System.out.println(example1.s3);
		System.out.println(example1.s4);
	}

}
```

What are the members we can access in different class, in different package?

```
package access.modifier.test;
import accessmodifier.Example1;
public class Example3 {

	public static void main(String[] args) {

		Example1 example1 = new Example1();

		example1.name2();

		System.out.println(example1.s1);
	}

}
```

What are the members we can access in different class, in different package through inheritance?

```
package access.modifier.test;
import accessmodifier.Example1;
public class Example4 extends Example1{

	public static void main(String[] args) {
		Example4 example4 = new Example4();
		example4.name1();
```

```
        example4.name2();

        System.out.println(example4.s1);
        System.out.println(example4.s3);
    }

}
```

Java Enum

Enum in java is a data type that contains fixed set of constants.
It can be used for days of the week (SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY and SATURDAY) , directions (NORTH, SOUTH, EAST and WEST) etc. The java enum constants are static and final implicitly.
A *Java Enum* is a special Java type used to define collections of constants. More precisely, a Java enum type is a special kind of Java class. An enum can contain constants, methods etc. Java enums were added in Java 5.

Points to remember for Java Enum

- enum improves type safety
- enum can be easily used in switch
- enum can be traversed
- enum can have fields, constructors and methods
- enum may implement many interfaces but cannot extend any class because it internally extends Enum class

Enum can be transversed

```
package enumjava;
public class Example1 {

    public enum Season {
        WINTER, SPRING, SUMMER, FALL
    }

    public static void main(String[] args) {
        for (Season s : Season.values())
            System.out.println(s);

    }
}
```

By Bhanu Pratap Singh
https://www.youtube.com/user/MrBhanupratap29/playlists
https://www.facebook.com/learnbybhanupratap/
https://www.udemy.com/seleniumbybhanu/

Enum Fields

You can add fields to a Java enum. Thus, each constant enum value gets these fields. The field values must be supplied to the constructor of the enum when defining the constants.

```
package enumjava;
public class Example2 {

	enum Season {
		WINTER(5), SPRING(10), SUMMER(15), FALL(20);

		private int value;

		private Season(int value) {
			this.value = value;
		}
	}

	public static void main(String args[]) {
		for (Season s : Season.values())
			System.out.println(s + " " + s.value);

	}
}
package enumjava;
public enum Test1 {
	HIGH(5), // calls constructor with value 3
	MEDIUM(6), // calls constructor with value 2
	LOW(2) // calls constructor with value 1
	; // semicolon needed when fields / methods follow

	private final int levelCode;

	private Test1(int levelCode) {
		this.levelCode = levelCode;
	}

	public int getLevelCode() {
		return levelCode;
	}
}
```

in the example above has a constructor which takes an int. The enum constructor sets the int field. When the constant enum values are defined, an int value is passed to the enum constructor.

By Bhanu Pratap Singh
https://www.youtube.com/user/MrBhanupratap29/playlists
https://www.facebook.com/learnbybhanupratap/
https://www.udemy.com/seleniumbybhanu/

The enum constructor must be either private or package scope (default). You cannot use public or protected constructors for a Java enum.

Switch Case call based on enum constant

```
package enumjava;

public enum Operation {
	PLUS, MINUS, TIMES, DIVIDE;

	double calculate(double x, double y) {
		switch (this) {
		case PLUS:
			return x + y;
		case MINUS:
			return x - y;
		case TIMES:
			return x * y;
		case DIVIDE:
			return x / y;
		default:
			throw new AssertionError("Unknown operations " + this);
		}
	}
}
package enumjava;

public class TestOperation {
	public static void main(String[] args) {

		double result = Operation.PLUS.calculate(1, 3);
		System.out.println(result);

		result = Operation.MINUS.calculate(1, 3);
		System.out.println(result);
	}
}
```

How to get single Data from enum constant

```
package enumjava;
public enum State {

	BIHAR(100, "Hindi", "Bjp"), UP(1000, "Hindi", "Bjp"), MP(100, "Hindi", "Congress"),
GOA(100, "Hindi", "Test");

	private State(int population, String language, String party) {
```

```
                this.population = population;
                this.language = language;
                this.party = party;
        }

        private int population;
        private String language;
        private String party;

        public int getPopulation() {
                return population;
        }

        public String getLanguage() {
                return language;
        }

        public String getParty() {
                return party;
        }

}
package enumjava;
public class TestState {
        public static void main(String[] args) {
                String data = State.BIHAR.getLanguage();
                System.out.println(data);
        }
}
```

Type conversion in Java with Examples

When you assign value of one data type to another, the two types might not be compatible with each other. If the data types are compatible, then Java will perform the conversion automatically known as Automatic Type Conversion and if not then they need to be casted or converted explicitly. For example, assigning an int value to a long variable.

Widening or Automatic Type Conversion

Widening conversion takes place when two data types are automatically converted. This happens when:

- The two data types are compatible.
- When we assign value of a smaller data type to a bigger data type.

For Example, in java the numeric data types are compatible with each other but no automatic conversion is supported from numeric type to char or Boolean. Also, char and Boolean are not compatible with each other.
Byte--→Short--→Int--→Long--→Float--→Double (**Widening or Automatic Type Conversion**)

```
package enumjava;
public class TypeCastingExample1 {

	public static void main(String[] args) {
		int i = 200;

		long l = i;

		float f = l;

		double d = i;

		System.out.println("Int value " + i);
		System.out.println("Long value " + l);
		System.out.println("Float value " + f);
		System.out.println("Double value " + d);
	}
}
```

Explicit Conversion

If we want to assign a value of larger data type to a smaller data type we perform explicit type casting or narrowing.

- This is useful for incompatible data types where automatic conversion cannot be done.
- Here, target-type specifies the desired type to convert the specified value to.

Double--→Float--→Long--→Int--→Short--→Byte (**Narrowing or Explicit Conversion**)

```
package typeCasting;

public class TypeCastingExample2 {

	public static void main(String[] args) {
		double d = 300.09;
```

```
		long l = (long) d;

		int i = (int) l;

		short b = (short)d;

		System.out.println("Double value " + d);

		System.out.println("Long value " + l);

		System.out.println("Int value " + i);

		System.out.println("Short value " + b);
	}
}
```

Convert Int and double to byte

Byte Class which is used to return number of bits required to represent a byte value in binary representation (two's complement).

When we do Byte. SIZE It returns a int value equal to 8.

```
package typeCasting;
public class TypeCastingExample3 {
	public static void main(String args[]) {
		byte b;
		int i = 259;
		double d = 412.142;

		System.out.println("Convert int to byte.");

		b = (byte) i;
		System.out.println("i = " + i + " b = " + b);

		System.out.println("Convert double to byte.");
		b = (byte) d;
		System.out.println("d = " + d + " b= " + b);
	}
}
```

```
package typeCasting;
public class A {

	public void test2(){
```

```
		System.out.println("test1");
	}

}
package typeCasting;
public class B extends A {

	public void test1() {
		System.out.println("test1");
	}

	public static void main(String[] args) {
		A obj = new B();

		obj.test2();

		A obj1 = new B();

		((B) obj1).test1();
	}

}
```

Type promotion in Expressions

While evaluating expressions, the intermediate value may exceed the range of operands and hence the expression value will be promoted. Some conditions for type promotion are:

1. Java automatically promotes each byte, short, or char operand to int when evaluating an expression.
2. If one operand is a long, float or double the whole expression is promoted to long, float or double respectively.

```
package typeCasting;
public class TypeCastingExample4 {

	public static void main(String args[]) {
		byte b = 47;
		char c = 'b';
		short s = 1026;
		int i = 80000;
		float f = 8.67f;
		double d = .923;

		double result = (f * b) + (i / c) - (d * s);
```

```
            System.out.println("result = " + result);
        }
}
```

Final Keyword In Java

The **final keyword** in java is used to restrict the user. The java final keyword can be used in many context. Final can be:

1. variable
2. method
3. class

The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable. It can be initialized in the constructor only. The blank final variable can be static also which will be initialized in the static block only.

1) Java final variable

If you make any variable as final, you cannot change the value of final variable(It will be constant).

```
package finalInJava;

public class Example1 {

        final int speedlimit = 100;

        void run() {
            speedlimit = 400; // Compile Time error
        }

        public static void main(String args[]) {
            Example1 obj = new Example1();
            obj.run();
        }
}
```

2) Java final method

If you make method as final , we can't override

```
package finalInJava;
```

```
public class Example2 {

	final void test1(){

	}
}
```

```
package finalInJava;

public class Example3 extends Example2{

	final void test1() {

	}

}
```

3) Java Final Class
When we make class as final class we can't extend

```
package finalInJava;

public final class Example4 {

}
```

```
package finalInJava;

public class Example5 extends Example4{

}
```

4) Can we Inherit final method

Yes, But we can't override

What is blank or uninitialized final variable?

A final variable that is not initialized at the time of declaration is known as blank final variable.
If you want to create a variable that is initialized at the time of creating object and once initialized may not be changed, it is useful. For example PAN CARD number of an employee. It can be initialized only in constructor.

```
package finalInJava;

public class Example6 {

	int id;
	String name;
	final String PAN_CARD_NUMBER;

	Example6(String panNumber) {
		this.PAN_CARD_NUMBER = panNumber;
	}

}
```

Can we re-initialize the final variables
Ans:- No

```
package finalInJava;

public class Example6 {

	int id;
	String name;
	final String PAN_CARD_NUMBER="123";

	Example6(String panNumber) {
		this.PAN_CARD_NUMBER = panNumber;
	}
}
```

Can we initialize blank final variable?
Yes, but only in constructor. For example:

```
package finalInJava;

public class Example7 {

	final int speedlimit;

	Example7() {
		speedlimit = 70;
		System.out.println(speedlimit);
	}

	public static void main(String args[]) {
		new Example7();
```

```
	}
}
```

Static blank final variable

A static final variable that is not initialized at the time of declaration is known as static blank final variable. It can be initialized only in static block.

```
package finalInJava;
public class Example8 {

	static final int data;
	static {
		data = 50;
	}

	public static void main(String args[]) {
		System.out.println(Example8.data);
	}
}
```

What is final parameter?

```
package finalInJava;
public class Example9 {

	int cube(final int n) {
		n = n + 2;
		return n * n * n;
	}

	public static void main(String args[]) {
		Example9 b = new Example9();
		b.cube(5);
	}
}
```

Can we declare a constructor final?

Runtime Polymorphism in Java

Runtime polymorphism or **Dynamic Method Dispatch** is a process in which a call to an overridden method is resolved at runtime rather than compile-time.

In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.

Example of Java Runtime Polymorphism

```
package RuntimePolymorphism;

public class Example1 {

	void run() {
		System.out.println("running");
	}
}
```

```
package RuntimePolymorphism;
public class Example2 extends Example1{

	void run() {
		System.out.println("running safely with 60km");
	}

	public static void main(String args[]) {
		Example1 b = new Example2();// upcasting
		b.run();
	}
}
```

```
package RuntimePolymorphism;
public class Example3 {

	float getRateOfInterest() {
		return 0;
	}
}

class SBI extends Example3 {
	float getRateOfInterest() {
		return 7.4f;
```

```
        }
}

class ICICI extends Example3 {
        float getRateOfInterest() {
                return 7.3f;
        }
}

class HDFC extends Example3 {
        float getRateOfInterest() {
                return 8.7f;
        }
}

class TestPolymorphism {
        public static void main(String args[]) {
                Example3 b;
                b = new SBI();
                System.out.println("SBI Rate of Interest: " + b.getRateOfInterest());
                b = new ICICI();
                System.out.println("ICICI Rate of Interest: " + b.getRateOfInterest());
                b = new HDFC();
                System.out.println("HDFC Rate of Interest: " + b.getRateOfInterest());
        }
}

package RuntimePolymorphism;
public class Animal {

        void eat() {
                System.out.println("eating...");
        }
}


package RuntimePolymorphism;
public class Cat extends Animal{

        void eat() {
                System.out.println("eats rat...");
        }
}
```

```
package RuntimePolymorphism;
public class Dog extends Animal{

	void eat() {
		System.out.println("eats bread...");
	}

}
```

```
package RuntimePolymorphism;
public class Lion extends Animal{

	void eat() {
		System.out.println("eats meat...");
	}
}
```

Static Binding and Dynamic Binding

When type of the object is determined at compiled time(by the compiler), it is known as static binding.
If there is any private, final or static method in a class, there is static binding.

Static Binding or Early Binding

The binding which can be resolved at compile time by compiler is known as static or early binding. The binding of static, private and final methods is compile-time. **Why?** The reason is that the these method cannot be overridden and the type of the class is determined at the compile time.

Dynamic Binding or Late Binding

When compiler is not able to resolve the call/binding at compile time, such binding is known as Dynamic or late Binding. Method Overriding is a perfect example of dynamic binding as in overriding both parent and child classes have same method and in this case the **type of the object** determines which method is to be executed. The type of object is determined at the run time so this is known as dynamic binding.

```
package staticAndDynamicBinding;

public class Example1 {
	private void eat() {
		System.out.println("dog is eating...");
	}

	public static void main(String args[]) {
		Example1 d1 = new Example1();
```

```
		d1.eat();
	}
}
```

```
package staticAndDynamicBinding;

public class Example4 {
	public static void test() {
		System.out.println("Example4");
	}
}
```

```
package staticAndDynamicBinding;

public class Example5 extends Example4 {
	public static void test() {
		System.out.println("Example5");
	}

	public static void main(String args[]) {
		Example4 obj = new Example5();

		Example4 obj2 = new Example5();
		obj.test();
		obj2.test();
	}
}
```

```
package staticAndDynamicBinding;

public class Human {
	public static void walk() {
		System.out.println("Human walks");
	}
}

public class Boy extends Human {
	public static void walk() {
		System.out.println("Boy walks");
	}

	public static void main(String args[]) {
		Human obj = new Boy();
		Human obj2 = new Human();
```

```
                obj.walk();
                obj2.walk();
        }
}
```

Dynamic binding

When type of the object is determined at run-time, it is known as dynamic binding.

```
package staticAndDynamicBinding;

public class Example3 extends Example2 {
        void eat() {
                System.out.println("dog is eating...");
        }
        public static void main(String args[]) {
                Example2 a = new Example3();
                a.eat();
        }
}
```

```
package staticAndDynamicBinding;

public class Human {
        public void walk() {
                System.out.println("Human walks");
        }
}

class Boy extends Human {
        public void walk() {
                System.out.println("Boy walks");
        }

        public static void main(String args[]) {
                Human obj = new Boy();
                Human obj2 = new Human();
                obj.walk();
                obj2.walk();
        }
}
```

Static Binding vs Dynamic Binding

Let's discuss the difference between static and dynamic binding in Java.

1. Static binding happens at compile-time while dynamic binding happens at runtime.

2. Binding of private, static and final methods always happen at compile time since these methods cannot be overridden. When the method overriding is actually happening and the reference of parent type is assigned to the object of child class type then such binding is resolved during runtime.
3. The binding of overloaded methods is static and the binding of overridden methods is dynamic.

Java instanceof

The **java instanceof operator** is used to test whether the object is an instance of the specified type (class or subclass or interface).

```
package instanceOfInJava;
public class Example1 {
	public static void main(String args[]) {
		Example1 s = new Example1();
		System.out.println(s instanceof Example1);
	}
}
```

```
package instanceOfInJava;
public class Animal {

}
```

```
package instanceOfInJava;
public class Dog extends Animal {

	public static void main(String args[]) {
		Dog d = new Dog();
		System.out.println(d instanceof Animal);
	}
}
```

```
package instanceOfInJava;
public class Example2 extends Example1 {

	public static void main(String[] args) {
		Example2 obj = new Example2();

		if (obj instanceof Example1) {
			System.out.println("Obj instanceof Example1");
		} else {
			System.out.println("Obj NOT instanceof Example1");
```

```
		}

		if (obj instanceof Example2) {
			System.out.println("Obj instanceof Example2");
		} else {
			System.out.println("Obj NOT instanceof Example2");
		}

		if (obj instanceof Object) {
			System.out.println("Obj instanceof Object");
		} else {
			System.out.println("Obj NOT instanceof Object");
		}

		Example1 obj1 = new Example1();

		if (obj1 instanceof Example1) {
			System.out.println("Obj instanceof Example1");
		} else {
			System.out.println("Obj NOT instanceof Example1");
		}

		if (obj1 instanceof Example2) {
			System.out.println("Obj instanceof Example2");
		} else {
			System.out.println("Obj NOT instanceof Example2");
		}

		if (obj1 instanceof Object) {
			System.out.println("Obj instanceof Object");
		} else {
			System.out.println("Obj NOT instanceof Object");
		}
	}

}

package instanceOfInJava;
public class Example3 {
	static int i = 0;
	static boolean b = false;
	static double d = 9.0;
	static char c = 'v';
	static float f = 9.0f;
```

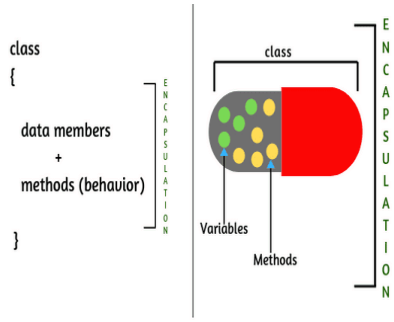
```
    public static void main(String[] args) {

        // As the name suggests, instanceof means an instance (object) of a class.
        // Primitive data types are not instances.
        if (i instanceof Integer) {

        }
        if(Integer.class.isInstance(i)) {
            System.out.println("True");
        }
        if(Boolean.class.isInstance(b)) {
            System.out.println("True");
        }
        if(Double.class.isInstance(d)) {
            System.out.println("True");
        }
        if(Character.class.isInstance(c)) {
            System.out.println("True");
        }
    }

}
```

Encapsulation in Java

Suppose you have an account in the bank. If your balance variable is declared as a public variable in the bank software, your account balance will be known as public, In this case, anyone can know your account balance. So, would you like it? Obviously No.

So, they declare balance variable as private for making your account safe, so that anyone cannot see your account balance. The person who has to see his account balance, he will have to access private members only through methods defined inside that class and this method will ask your account holder name or user Id, and password for authentication. Thus, We can achieve security by utilizing the concept of data hiding. This is called Encapsulation.

```
class Account{
private int account_number;
private int account_balance;
    // getter method
        public int getBalance() {
        return this.account_balance;
    }
    // setter method
        public void setNumber(int num) {
        this.account_number = num;
    }
}
```



Key point:

By Bhanu Pratap Singh
https://www.youtube.com/user/MrBhanupratap29/playlists
https://www.facebook.com/learnbybhanupratap/
https://www.udemy.com/seleniumbybhanu/

In Java, Encapsulation is one of the four principles of OOPs concepts and the other three are Abstraction, Inheritance, and Polymorphism.

How to achieve or implement Encapsulation in Java

We can achieve encapsulation in Java in the following ways.

1. Declaring the instance variable of the class as private. so that it cannot be accessed directly by anyone from outside the class.
2. Provide the public setter and getter methods in the class to set/modify the values of the variable/fields.

Advantages of Encapsulation in Java

There are following advantages of encapsulation in Java.

1. The encapsulated code is more flexible and easy to change with new requirements.
2. It prevents the other classes to access the private fields.
3. Encapsulation allows modifying implemented code without breaking others code who have implemented the code.
4. It keeps the data and codes safe from external inheritance. Thus, Encapsulation helps to achieve security.
5. It improves the maintainability of the application.
6. If you don't define the setter method in the class then the fields can be made read-only.
7. If you don't define the getter method in the class then the fields can be made write-only.

Disadvantages of Encapsulation in Java

The main disadvantage of the encapsulation in Java is it increases the length of the code and slow shutdown execution.

Real-time Example 1:

When you log into your email accounts such as Gmail, Yahoo mail, or Rediff mail, there is a lot of internal processes taking place in the backend and you have no control over it.

When you enter the password for logging, they are retrieved in an encrypted form and verified and then you are given the access to your account. You do not have control over it that how the password has been verified. Thus, it keeps our account safe from being misused.

Real-time Example 2:

Suppose you have an account in the bank. If your balance variable is declared as a public variable in the bank software, your account balance will be known as public, In this case, anyone can know your account balance. So, would you like it? Obviously No.

So, they declare balance variable as private for making your account safe, so that anyone cannot see your account balance. The person who has to see his account balance, he will have to access private members only through methods defined inside that class and this method will ask your account holder name or user Id, and password for authentication.

By Bhanu Pratap Singh
https://www.youtube.com/user/MrBhanupratap29/playlists
https://www.facebook.com/learnbybhanupratap/
https://www.udemy.com/seleniumbybhanu/

Thus, We can achieve security by utilizing the concept of data hiding. This is called Encapsulation.

```
package encapsulationInJava;
public class Example2 {
	private String name;

	private String age;

	public String getName() {
		return name;
	}

	public void setName(String name) {
		this.name = name;
	}

	public String getAge() {
		return age;
	}

	public void setAge(String age) {
		this.age = age;
	}

}
```

```
package encapsulationInJava;

public class TestExample2 {

	public static void main(String[] args) {
		Example2 obj = new Example2();
		obj.setAge("10");
		obj.setName("Test");

		System.out.println("name="+obj.getName()+" age="+obj.getAge());
	}
}
```

While setting data to variables we can provide validation logic

```
package encapsulationInJava;

public class Example3 {
```

```
	private int age;
	private String name;

	public int getAge() {
		return age;
	}

	public void setAge(int age) throws Exception {

		if (age < 10) {
			throw new Exception("age should not be less than 10");
		}
		this.age = age;
	}

	public String getName() {
		return name;
	}

	public void setName(String name) throws Exception {
		if (name == null || name == "") {
			throw new Exception("name should not be empty or null");

		}
		this.name = name;
	}

}
package encapsulationInJava;
public class TestExample3 {

	public static void main(String[] args) throws Exception {
		Example3 obj = new Example3();
		obj.setAge(11);
		System.out.println(obj.getAge());

		obj.setAge(9);
		System.out.println(obj.getAge());
	}

}
```

It help us to provide read only access

```
package encapsulationInJava;
```

```
public class Example4 {

	private String schoolName = "ABC";

	private String studentName;

	public String getSchoolName() {
		return schoolName;
	}

	public String getStudentName() {
		return studentName;
	}

	public void setStudentName(String studentName) {
		this.studentName = studentName;
	}

	@Override
	public String toString() {
		return "Example4 [schoolName=" + schoolName + ", studentName=" +
studentName + "]";
	}

}
package encapsulationInJava;

public class TestExample4 {

	public static void main(String[] args) {
		Example4 obj = new Example4();
		obj.setStudentName("Ram");

		System.out.println(obj);
	}
}
```

Let's understand how Encapsulation allows modifying implemented Java code without breaking others code who have implemented the code? Since data type of Id has been changed from String to Integer. So, I will only change in getter and setter method to avoid breaking of other codes.

```
package encapsulationInJava;
public class Student {
```

```
	String id; // Here, No encapsulation is used. Since the field is not
				// private.
}
package encapsulationInJava;

public class EncapsulationTest1 {

	public static void main(String[][] args) {
		Student st = new Student();
		st.id = "5"; // As the field is not private. So, it can be accessed by
					// anyone from outside the class.
	}
}
```

Suppose in above program 3, anyone changes the data type of id from String to Integer like this:

```
package encapsulationInJava;
public class Student {

	int id; // Here, No encapsulation is used. Since the field is not
				// private.
}

package encapsulationInJava;
public class EncapsulationTest1 {

	public static void main(String[][] args) {
		Student st = new Student();
		st.id = "5";
	}
}
```

Now, what will happen? Whenever Id has been used then the compilation time error will be generated.
When you have encapsulation , this issue can be solved without breaking the implementation.

```
public class Student {
	Integer id;

	public String getId() {
```

```
		return String.valueOf(id);
	}

	public void setId(String id) {
		this.id = Integer.parseInt(id);
	}
}
```

What you should not do in Encapsulation.

```
package encapsulationInJava;
import java.util.Arrays;

public class TestExample5 {

	private int[] a = { 1, 2, 3, 4 };

	public int[] getA() {
		return a;
	}

	public void setA(int[] a) {
		this.a = a;
	}
}
package encapsulationInJava;
import java.util.Arrays;
public class TestTestExample5 {
	public static void main(String[] args) {
		TestExample5 example5 = new TestExample5();
		int[] data = example5.getA();
		data[2] = 0;
		System.out.println(Arrays.toString(example5.getA()));

		TestExample5 example6 = new TestExample5();
		// Returns a shallow copy
		int[] data1 = example5.getA().clone();
		data1[2] = 0;
		System.out.println(Arrays.toString(example6.getA()));
	}
}
```

Output

[1, 2, 0, 4]

[1, 2, 3, 4]

```
package encapsulationInJava;
import java.util.ArrayList;
import java.util.List;
public class TestExample6 {
	private List<String> books;
	public TestExample6() {
		this.books = new ArrayList<String>();
		books.add("Test1");
		books.add("Test2");
		books.add("Test3");
	}

	public List<String> getBooks() {
		return new ArrayList<String>(books);
		//return books;
	}

	public void setBooks(List<String> books) {
		this.books = books;
	}

}
package encapsulationInJava;

import java.util.List;

public class TestTestExample6 {

	public static void main(String[] args) {
		TestExample6 example6 = new TestExample6();
		System.out.println(example6.getBooks());
		List<String> data = example6.getBooks();
		data.remove(1);
		System.out.println(data);
		List<String> data1 = example6.getBooks();
		data1 = null;
		System.out.println(data1);
	}

}
```

Output

[Test1, Test2, Test3]

[Test1, Test3]
Null

```
package encapsulationInJava;

import java.util.*;

public class CollectionGetterSetter {
	private List<String> listTitles;

	public void setListTitles(List<String> titles) {
		this.listTitles = titles;
	}

	public List<String> getListTitles() {
		return this.listTitles;
	}

	public static void main(String[] args) {
		CollectionGetterSetter app = new CollectionGetterSetter();
		List<String> titles1 = new ArrayList();
		titles1.add("Name");
		titles1.add("Address");
		titles1.add("Email");
		titles1.add("Job");
		app.setListTitles(titles1);
		System.out.println("Titles 1: " + titles1);
		titles1.set(2, "Habilitation");
		List<String> titles2 = app.getListTitles();
		System.out.println("Titles 2: " + titles2);
		titles2.set(0, "Full name");
		List<String> titles3 = app.getListTitles();
		System.out.println("Titles 3: " + titles3);
	}
}
```

Output:

Titles 1: [Name, Address, Email, Job]
Titles 2: [Name, Address, Habilitation, Job]
Titles 3: [Full name, Address, Habilitation, Job]

```
package encapsulationInJava;

import java.util.*;

public class CollectionGetterSetter2 {
```

```
	private List<String> listTitles;

	public void setListTitles(List<String> titles) {
		this.listTitles = new ArrayList<String>(titles);
	}

	public List<String> getListTitles() {
		return new ArrayList<String>(this.listTitles);
	}

	public static void main(String[] args) {
		CollectionGetterSetter2 app = new CollectionGetterSetter2();
		List<String> titles1 = new ArrayList<String>();
		titles1.add("Name");
		titles1.add("Address");
		titles1.add("Email");
		titles1.add("Job");
		app.setListTitles(titles1);
		System.out.println("Titles 1: " + titles1);
		titles1.set(2, "Habilitation");
		List<String> titles2 = app.getListTitles();
		System.out.println("Titles 2: " + titles2);
		titles2.set(0, "Full name");
		List<String> titles3 = app.getListTitles();
		System.out.println("Titles 3: " + titles3);
	}
}
```

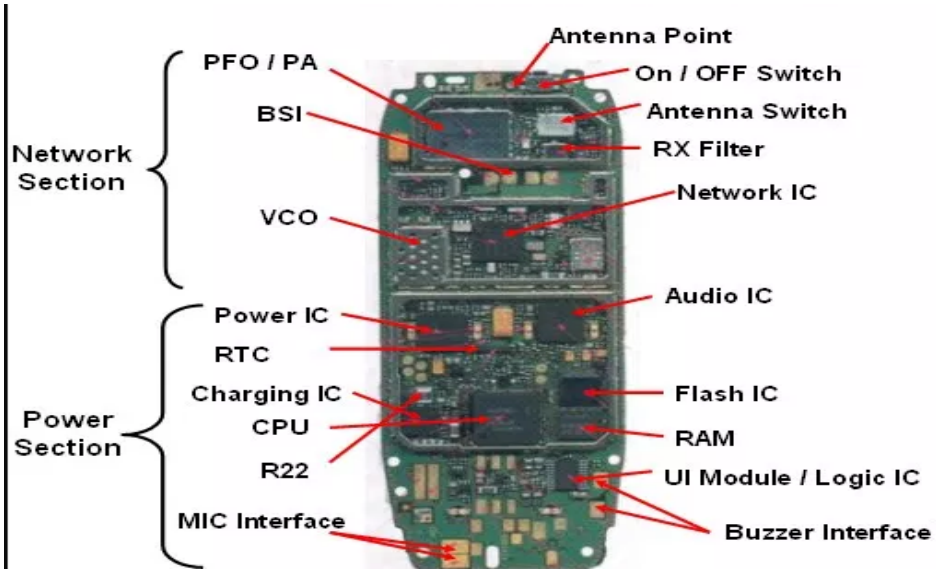
Output

Titles 1: [Name, Address, Email, Job]
Titles 2: [Name, Address, Email, Job]
Titles 3: [Name, Address, Email, Job]

Everything has many properties and behaviours so take whatever object you want TV, Mobile, Car, Human or anything.

Abstraction:

The basic components of a coffee machine



1. Process of picking the essence of an object you really need
2. In other words, pick the properties you need from the object Example:
   a. TV - Sound, Visuals, Power Input, Channels Input.
   b. Mobile - Button/Touch screen, power button, volume button, sim port.
   c. Car - Steering, Break, Clutch, Accelerator, Key Hole.
   d. Human - Voice, Body, Eye Sight, Hearing, Emotions.

By Bhanu Pratap Singh
https://www.youtube.com/user/MrBhanupratap29/playlists
https://www.facebook.com/learnbybhanupratap/
https://www.udemy.com/seleniumbybhanu/

Encapsulation:

1. Process of hiding the details of an object you don't need
2. In other words, hide the properties and operations you don't need from the object but are required for the object to work properly Example:
   a. TV - Internal and connections of Speaker, Display, Power distribution b/w components, Channel mechanism.
   b. Mobile - How the input is parsed and processed, How pressing a button on/off or changes volumes, how sim will connect to service providers.
   c. Car - How turning steering turns the car, How break slow or stops the car, How clutch works, How accelerator increases speed, How key hole switch on/off the car.
   d. Human - How voice is produced, What's inside the body, How eye sight works, How hearing works, How emotions generate and affect us.

ABSTRACT everything you need and ENCAPSU

What is Abstraction

Abstraction is a process of hiding the implementation details from the user. Only the functionality will be provided to the user. In Java, abstraction is achieved using abstract classes and interfaces.

Java Abstraction Example

To give an example of abstraction we will create one superclass called Employee and two subclasses – Contractor and Fulltime Employee. Both subclasses have common properties to share, like the name of the employee and the amount of money the person will be paid per hour. There is one major difference between contractors and full-time employees – the time they work for the company. Full-time employees work constantly 8 hours per day and the working time of contractors may vary.

```
package abstractionInJava;
public abstract class Employee {
	private String name;
	private int paymentPerHour;

	public Employee(String name, int paymentPerHour) {
		this.name = name;
		this.paymentPerHour = paymentPerHour;
	}

	public abstract int calculateSalary();

	public String getName() {
		return name;
	}
```

```
	public void setName(String name) {
		this.name = name;
	}

	public int getPaymentPerHour() {
		return paymentPerHour;
	}

	public void setPaymentPerHour(int paymentPerHour) {
		this.paymentPerHour = paymentPerHour;
	}
}
```

The Contractor class inherits all properties from its parent Employee but have to provide it's own implementation of calculateSalary() method. In this case we multiply the value of payment per hour with given working hours.

```
package abstractionInJava;
public class Contractor extends Employee {

	private int workingHours;

	public Contractor(String name, int paymentPerHour, int workingHours) {
		super(name, paymentPerHour);
		this.workingHours = workingHours;
	}

	@Override
	public int calculateSalary() {
		return getPaymentPerHour() * workingHours;
	}
}
```

The FullTimeEmployee also has it's own implementation ofcalculateSalary()method. In this case we just multiply by constant 8 hours.

```
package abstractionInJava;

public class FullTimeEmployee extends Employee {

	public FullTimeEmployee(String name, int paymentPerHour) {
		super(name, paymentPerHour);
	}

	@Override
```

```
	public int calculateSalary() {
		return getPaymentPerHour() * 8;
	}
}

package abstractionInJava
public class TestAbstraction {

	public static void main(String[] args) {

		Employee obj = new Contractor("Test1", 300, 4);

		System.out.println("Contractor Salary "+obj.calculateSalary());

		Employee obj1 = new FullTimeEmployee("Test1", 500);

		System.out.println("FullTime Salary "+obj1.calculateSalary());
	}
}
```

Output

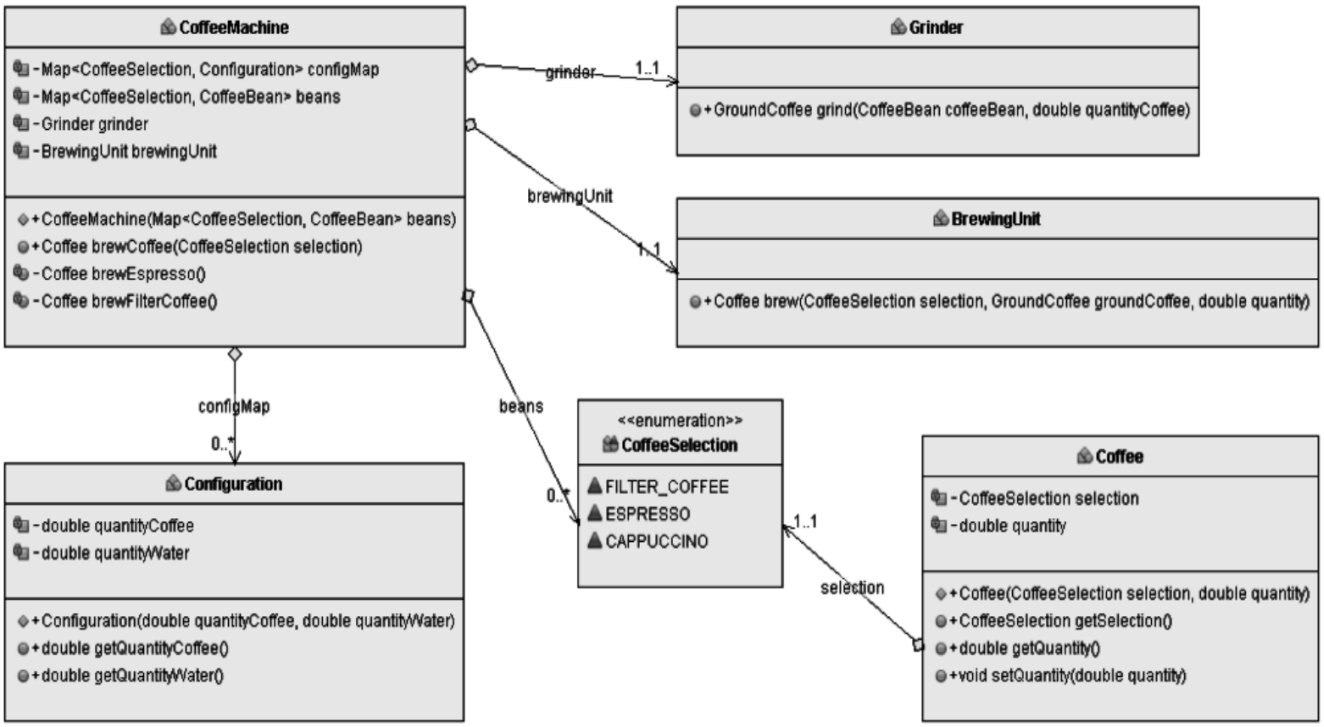
Contractor Salary 1200
FullTime Salary 4000

Abstraction in the real world

when I wake up in the morning, I go into my kitchen, switch on the coffee machine and make coffee. Sounds familiar?
Making coffee with a coffee machine is a good example of abstraction.
You need to know how to use your coffee machine to make coffee. You need to provide water and coffee beans, switch it on and select the kind of coffee you want to get.

The thing you don't need to know is how the coffee machine is working internally to brew a fresh cup of delicious coffee. You don't need to know the ideal temperature of the water or the amount of ground coffee you need to use.

Someone else worried about that and created a coffee machine that now acts as an abstraction and hides all these details. You just interact with a simple interface that doesn't require any knowledge about the internal implementation.




```
package encapsulationInJava;

//CoffeeSelection is a simple enum providing a set of predefined values
// for the different kinds of coffees.

public enum CoffeeSelection {
	ESPRESSO, CAPPUCCINO, FILTER_COFFEE;
}
package encapsulationInJava;

public class Configuration {

	private double quantityCoffee;
	private double quantityWater;

	public Configuration(double quantityCoffee, double quantityWater) {
		this.quantityCoffee = quantityCoffee;
		this.quantityWater = quantityWater;
	}
```

By Bhanu Pratap Singh
https://www.youtube.com/user/MrBhanupratap29/playlists
https://www.facebook.com/learnbybhanupratap/
https://www.udemy.com/seleniumbybhanu/

```
	public double getQuantityCoffee() {
		return quantityCoffee;
	}

	public double getQuantityWater() {
		return quantityWater;
	}
}
```

```
package encapsulationInJava;

public class CoffeeBean {

	private String name;

	private double quantity;

	public CoffeeBean(String name, double quantity) {
		this.name = name;
		this.quantity = quantity;
	}

	public String getName() {
		return name;
	}

	public void setName(String name) {
		this.name = name;
	}

	public double getQuantity() {
		return quantity;
	}

	public void setQuantity(double quantity) {
		this.quantity = quantity;
	}
}
```

```
package encapsulationInJava;

public class BrewingUnit {
```

```
	public Coffee brew(CoffeeSelection selection, GroundCoffee groundCoffee, double
quantity) {
		return new Coffee(selection, quantity);
	}
}

package encapsulationInJava;
public class Grinder {
	public GroundCoffee grind(CoffeeBean coffeeBean, double quantityCoffee) {
		return new GroundCoffee();
	}
}
package encapsulationInJava;
public class GroundCoffee {

}
package encapsulationInJava;
import java.util.HashMap;
import java.util.Map;

public class CoffeeMachine implements CoffeeMachineIf {

	private Map<CoffeeSelection, Configuration> configMap;

	private Map<CoffeeSelection, CoffeeBean> beans;

	private Grinder grinder;

	private BrewingUnit brewingUnit;

	public CoffeeMachine(Map<CoffeeSelection, CoffeeBean> beans) {
		this.beans = beans;
		this.grinder = new Grinder();
		this.brewingUnit = new BrewingUnit();
		this.configMap = new HashMap<CoffeeSelection, Configuration>();
		this.configMap.put(CoffeeSelection.ESPRESSO, new Configuration(80, 2800));
		this.configMap.put(CoffeeSelection.FILTER_COFFEE, new Configuration(300,
4800));
	}

	@Override
	public Coffee brewCoffee(CoffeeSelection selection) throws Exception {
		switch (selection) {
		case FILTER_COFFEE:
```

```
                return brewFilterCoffee();

            case ESPRESSO:
                return brewEspresso();

            default:
                throw new Exception("CoffeeSelection [" + selection + "] not
supported!");
            }
        }

        private Coffee brewEspresso() {
            Configuration config = configMap.get(CoffeeSelection.ESPRESSO);

            GroundCoffee groundCoffee =
this.grinder.grind(this.beans.get(CoffeeSelection.ESPRESSO),
                    this.beans.get(CoffeeSelection.ESPRESSO).getQuantity());

            // brew an espresso
            return this.brewingUnit.brew(CoffeeSelection.ESPRESSO, groundCoffee,
config.getQuantityWater());
        }

        private Coffee brewFilterCoffee() {
            Configuration config = configMap.get(CoffeeSelection.FILTER_COFFEE);

            // grind the coffee beans
            GroundCoffee groundCoffee =
this.grinder.grind(this.beans.get(CoffeeSelection.FILTER_COFFEE),
                    this.beans.get(CoffeeSelection.FILTER_COFFEE).getQuantity());

            // brew a filter coffee
            return this.brewingUnit.brew(CoffeeSelection.FILTER_COFFEE, groundCoffee,
config.getQuantityWater());
        }

        public void addBeans(CoffeeSelection sel, CoffeeBean newBeans) throws Exception {
            CoffeeBean existingBeans = this.beans.get(sel);

            if (existingBeans != null) {
                if (existingBeans.getName().equals(newBeans.getName())) {
                    existingBeans.setQuantity(existingBeans.getQuantity() +
newBeans.getQuantity());
                } else {
```

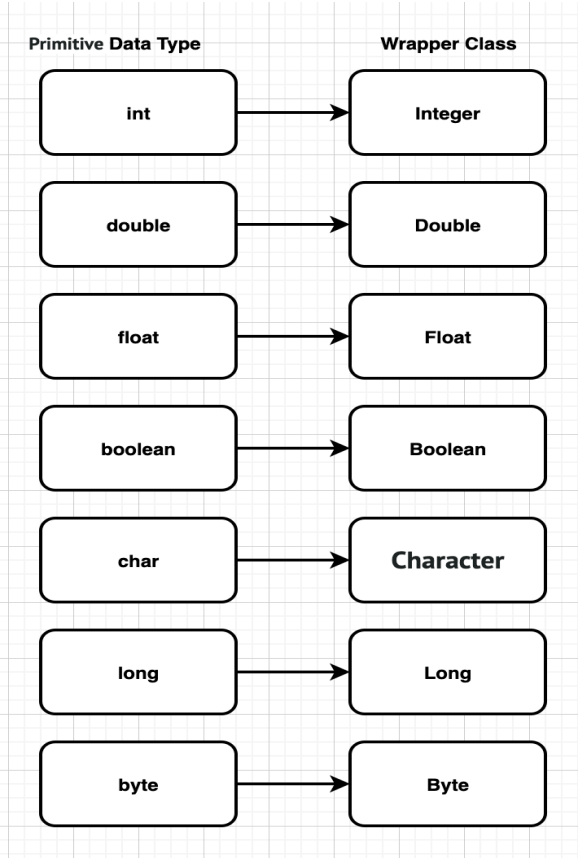
```
                                throw new Exception("Only one kind of beans supported for
each CoffeeSelection.");
                        }
                } else {
                        this.beans.put(sel, newBeans);
                }
        }
}

package encapsulationInJava;

import java.util.HashMap;
import java.util.Map;

public class CoffeeApp {
        public static void main(String[] args) {
                // create a Map of available coffee beans
                Map<CoffeeSelection, CoffeeBean> beans = new HashMap<CoffeeSelection,
CoffeeBean>();
                beans.put(CoffeeSelection.ESPRESSO, new CoffeeBean("My favorite espresso
bean", 1000));
                beans.put(CoffeeSelection.FILTER_COFFEE, new CoffeeBean("My favorite
filter coffee bean", 1000));

                // get a new CoffeeMachine object
                CoffeeMachine machine = new CoffeeMachine(beans);

                // brew a fresh coffee
                try {
                        Coffee espresso = machine.brewCoffee(CoffeeSelection.ESPRESSO);
                        System.out.println("Your "+espresso.getSelection()+" is ready");
                } catch (Exception e) {
                        e.printStackTrace();
                }
        } // end main
} // end CoffeeApp
```

Wrapper Classes in Java

We can wrap a primitive value into a wrapper class object. In other words, wrapper classes provide a way to use **primitive data types (int, char, short, byte, etc) as objects**. These wrapper classes come under **java.util package**.

https://docs.oracle.com/javase/7/docs/api/index.html (Go to java.lang package)

By Bhanu Pratap Singh
https://www.youtube.com/user/MrBhanupratap29/playlists
https://www.facebook.com/learnbybhanupratap/
https://www.udemy.com/seleniumbybhanu/



Why we should wrapper class

1. Wrapper class convert primitive data types into objects. Objects are needed if we wish to modify the arguments passed into a method (because primitive types are passed by value).
2. java.util package classes consumes only objects and hence we need wrapper classes for this.
3. Data structures in the Collection framework, such as ArrayList , LinkedList and Vector, store only objects (reference types) and not primitive types.
4. An object is Required to support synchronization in multithreading.

Class Integer

Constructor Summary

Constructors	
Constructor and Description	
`Integer(int value)`	Constructs a newly allocated `Integer` object that represents the specified `int` value.
`Integer(String s)`	Constructs a newly allocated `Integer` object that represents the `int` value indicated by the `String` parameter.

By Bhanu Pratap Singh
https://www.youtube.com/user/MrBhanupratap29/playlists
https://www.facebook.com/learnbybhanupratap/
https://www.udemy.com/seleniumbybhanu/

Method Summary

Methods	
Modifier and Type	**Method and Description**
static int	**bitCount**(int i) Returns the number of one-bits in the two's complement binary representation of the specified int value.
byte	**byteValue**() Returns the value of this Integer as a byte.
static int	**compare**(int x, int y) Compares two int values numerically.
int	**compareTo**(**Integer** anotherInteger) Compares two Integer objects numerically.
static **Integer**	**decode**(**String** nm) Decodes a String into an Integer.
double	**doubleValue**() Returns the value of this Integer as a double.
boolean	**equals**(**Object** obj) Compares this object to the specified object.
float	**floatValue**() Returns the value of this Integer as a float.
static **Integer**	**getInteger**(**String** nm) Determines the integer value of the system property with the specified name.
static **Integer**	**getInteger**(**String** nm, int val) Determines the integer value of the system property with the specified name.
static **Integer**	**getInteger**(**String** nm, **Integer** val) Returns the integer value of the system property with the specified name.
int	**hashCode**() Returns a hash code for this Integer.
static int	**highestOneBit**(int i) Returns an int value with at most a single one-bit, in the position of the highest-order ("leftmost") one-bit in the specified int value.
int	**intValue**() Returns the value of this Integer as an int.
long	**longValue**() Returns the value of this Integer as a long.

By Bhanu Pratap Singh
https://www.youtube.com/user/MrBhanupratap29/playlists
https://www.facebook.com/learnbybhanupratap/
https://www.udemy.com/seleniumbybhanu/

static int	**lowestOneBit**(int i) Returns an int value with at most a single one-bit, in the position of the lowest-order ("rightmost") one-bit in the specified int value.
static int	**numberOfLeadingZeros**(int i) Returns the number of zero bits preceding the highest-order ("leftmost") one-bit in the two's complement binary representation of the specified int value.
static int	**numberOfTrailingZeros**(int i) Returns the number of zero bits following the lowest-order ("rightmost") one-bit in the two's complement binary representation of the specified int value.
static int	**parseInt**(**String** s) Parses the string argument as a signed decimal integer.
static int	**parseInt**(**String** s, int radix) Parses the string argument as a signed integer in the radix specified by the second argument.
static int	**reverse**(int i) Returns the value obtained by reversing the order of the bits in the two's complement binary representation of the specified int value.
static int	**reverseBytes**(int i) Returns the value obtained by reversing the order of the bytes in the two's complement representation of the specified int value.
static int	**rotateLeft**(int i, int distance) Returns the value obtained by rotating the two's complement binary representation of the specified int value left by the specified number of bits.
static int	**rotateRight**(int i, int distance) Returns the value obtained by rotating the two's complement binary representation of the specified int value right by the specified number of bits.
short	**shortValue**() Returns the value of this Integer as a short.
static int	**signum**(int i) Returns the signum function of the specified int value.
static **String**	**toBinaryString**(int i) Returns a string representation of the integer argument as an unsigned integer in base 2.
static **String**	**toHexString**(int i) Returns a string representation of the integer argument as an unsigned integer in base 16.

By Bhanu Pratap Singh
https://www.youtube.com/user/MrBhanupratap29/playlists
https://www.facebook.com/learnbybhanupratap/
https://www.udemy.com/seleniumbybhanu/

static String	toOctalString(int i) Returns a string representation of the integer argument as an unsigned integer in base 8.
String	toString() Returns a String object representing this Integer's value.
static String	toString(int i) Returns a String object representing the specified integer.
static String	toString(int i, int radix) Returns a string representation of the first argument in the radix specified by the second argument.
static Integer	valueOf(int i) Returns an Integer instance representing the specified int value.
static Integer	valueOf(String s) Returns an Integer object holding the value of the specified String.
static Integer	valueOf(String s, int radix) Returns an Integer object holding the value extracted from the specified String when parsed with the radix given by the second argument.

```
package wrapperClassInjava;

public class Example1 {

	public static void main(String[] args) {

		Integer integer1 = new Integer(10);
		Integer integer2 = new Integer("100");

		System.out.println(integer1.compareTo(integer2));
		System.out.println(integer1.doubleValue());
		System.out.println(integer1.intValue());
		System.out.println(integer1.longValue());
		System.out.println("====================");
		System.out.println(integer1.compareTo(integer2));
		System.out.println(integer2.doubleValue());
		System.out.println(integer2.intValue());
		System.out.println(integer2.longValue());
		System.out.println("====================");

		/**
		 * @return the integer value represented by the argument in
decimal.@exception
```

```
		 *	NumberFormatException if the string does not contain aparsable
		 *	integer.
		 */
		int s2 = Integer.parseInt("1000");
		System.out.println(s2);

		/**
		 * Both valueOf and parseInt methods are used to convert String to Integer in
Java,
		 * but there are subtle difference between them. ... valueOf() of java. lang.
		 * Integer returns an Integer object, while parseInt() method returns an int
primitive
		 */
		int s3 = Integer.valueOf(1000);
	}
}
```

Integer.valueOf() returns an Integer object while Integer.parseInt() returns a primitive int.

```
package wrapperClassInjava;
public class Test1 {
	public static void main(String args[]) {
		String s = "77";
		int str = Integer.parseInt(s);
		System.out.print(str);
		Integer str1 = Integer.valueOf(s);
		System.out.print(str1);
	}
}
```

Both String and integer can be passed a parameter to Integer.valueOf() whereas only a String can be passed as parameter to Integer.parseInt().

```
package wrapperClassInjava;

class Test3 {
	public static void main(String args[]) {
		int val = 99;
		int str1 = Integer.valueOf(val);
		System.out.print(str1);
		int str = Integer.parseInt(val);
		System.out.print(str);
	}
}
```

By Bhanu Pratap Singh
https://www.youtube.com/user/MrBhanupratap29/playlists
https://www.facebook.com/learnbybhanupratap/
https://www.udemy.com/seleniumbybhanu/

Integer.valueOf() can take a character as parameter and will return its corresponding unicode value whereas Integer.parseInt() will produce an error on passing a character as parameter.

```
package wrapperClassInjava;
public class Example2 {
	public static void main(String args[]) {
		char val = 'A';
		int str1 = Integer.valueOf(val);
		System.out.print(str1);
		int str = Integer.parseInt(val);
		System.out.print(str);
	}
}
```

INTEGER.PARSEINT()	INTEGER.VALUEOF()
It can only take a String as a parameter.	It can take a String as well as an integer as parameter.
It returns a primitive int value.	It returns an Integer object.
When an integer is passed as parameter, it produces an error due to incompatible types	When an integer is passed as parameter, it returns an Integer object corresponding to the given parameter.
This method produces an error(incompatible types) when a character is passed as parameter.	This method can take a character as parameter and will return the corresponding unicode.
This lags behind in terms of performance since parsing a string takes a lot of time when compared to generating one.	This method is likely to yield significantly better space and time performance by caching frequently requested values.
If we need the primitive int datatype then Integer.parseInt() method is to be used.	If Wrapper Integer object is needed then valueOf() method is to be used.

By Bhanu Pratap Singh
https://www.youtube.com/user/MrBhanupratap29/playlists
https://www.facebook.com/learnbybhanupratap/
https://www.udemy.com/seleniumbybhanu/

Hexadecimal	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Integer.parseInt(String,radix) method is used to parse integer from binary, octal or hexa decimal numbers. It is basically used to convert binary to integer, octal to integer and hexadecimal to integer.

String is alphanumeric value;
Radix is used to specify type of input String.

Value range of radix is **2 to 36**. Get radix range
from **Character.MIN_RADIX** and **Character.MAX_RADIX**.
2 - base-2 numbers (Binary). 2 means input value only contains 0-1. Otherwise will through NumberFormatException
8 - base-8 numbers (Octal). 8 means input string contains only 0-7. Otherwise will through NumberFormatException
10 - base-10 numbers (Decimal). It may contains 0-9 characters.
16 - base-16 numbers (Hexa decimal). It may contains 0-9 and A,B,C,D,E,F characters.
36 - base-36 numbers (Numbers and Letters). It may contains 0-9 (10) numbers and A to Z (26) letters.

```
package wrapperClassInjava;
```

```
public class Snippet {
	public static void main(String args[]) {
		// Binary to decimal conversion
		System.out.println(Integer.parseInt("101010", 2));

		// Octal to decimal conversion
		System.out.println(Integer.parseInt("463", 8));

		// Hexadecimal to decimal conversion
		System.out.println(Integer.parseInt("4AC", 16));

		// String to decimal conversion - throughs NumberFormat Exception, Because
it
		// contains D
		System.out.println(Integer.parseInt("6BDC", 13));
	}

}
```

```
package wrapperClassInjava;


public class Ecample2 {
	public static void main(String[] args) {

		Double double1 = new Double(10);
		Double double2 = new Double("100");

		System.out.println(double1.compareTo(double2));
		System.out.println(double1.doubleValue());
		System.out.println(double1.intValue());
		System.out.println(double1.longValue());

		double s2 = Double.parseDouble("1000");

		//double s1 = Double.parseDouble(1000);

		double s3 = Double.valueOf(1000);

		double s4 = Double.valueOf("1000");

		System.out.println(double1.MAX_VALUE);
		System.out.println(double1.MIN_VALUE);
	}
```

By Bhanu Pratap Singh
https://www.youtube.com/user/MrBhanupratap29/playlists
https://www.facebook.com/learnbybhanupratap/
https://www.udemy.com/seleniumbybhanu/

```
}

package wrapperClassInjava;

public class Ecample2 {
	public static void main(String[] args) {

		Boolean boolean1 = new Boolean(true);
		Boolean boolean2 = new Boolean("test");

		System.out.println(boolean1.compareTo(boolean2));
		System.out.println(boolean1.booleanValue());

		System.out.println(Boolean.parseBoolean("true"));

		System.out.println(Boolean.parseBoolean("ABS"));
	}
}
```

String In Java

String is a sequence of characters, for e.g. “Hello” is a string of 5 characters. In java, string is an immutable object which means it is constant and can cannot be changed once it has been created.

Creating a String

There are two ways to create a String in Java

1. String literal
2. Using new keyword

String literal

In java, Strings can be created like this: Assigning a String literal to a String instance:
String str1 = "Welcome";

String str2 = "Welcome";

if the object already exist in the memory it does not create a new Object rather it assigns the same old object to the new instance, that means even though we have two string instances above(str1 and str2) compiler only created on string object (having the value “Welcome”) and assigned the same to both the instances. For example there are 10 string

instances that have same value, it means that in memory there is only one object having the value and all the 10 string instances would be pointing to the same object.

Using New Keyword

String str1 = new String("Welcome");
String str2 = new String("Welcome");

In this case compiler would create two different object in memory having the same string.

Immutable String
Immutable String means , once String object is created it can't be changed or modified.
Example:

```
package stringInjava;
public class Example1 {

	public static void main(String args[]) {
		String s = "Bhanu";
		s.concat(" Java");
		System.out.println(s);
	}
}
```

Here output remains "Bhanu" , since String is immutable and new object is created in memory "BhanuJava"

String compare
There are three ways to compare string in java:

1. By equals() method
2. By = = operator
3. By compareTo() method

equals() method compares the original content of the string
The = = operator compares references not values.

The String compareTo() method compares values lexicographically and returns an integer value that describes if first string is less than, equal to or greater than second string. Suppose s1 and s2 are two string variables. If:

- **s1 == s2** :0
- **s1 > s2** :positive value
- **s1 < s2** :negative value

```
package stringInjava;
```

```
public class Example2 {
	public static void main(String[] args) {

		String s1 = "Bhanu";
		String s2 = "Bhanu";
		String s3 = new String("Bhanu");
		String s4 = "Test";
		System.out.println(s1.equals(s2));// true
		System.out.println(s1.equals(s3));// true
		System.out.println(s1.equals(s4));// false

		s1 = "Bhanu";
		s2 = "BHANU";

		System.out.println(s1.equals(s2));// false
		System.out.println(s1.equalsIgnoreCase(s2));// true

		s2 = "Bhanu";
		s3 = new String("Bhanu");
		System.out.println(s1 == s2);// true (because both refer to same instance)
		System.out.println(s1 == s3);// false(because s3 refers to instance created in
nonpool)


		s1 = "Bhanu";
		s2 = "Bhanu";
		s3 = "Bist1";
		System.out.println(s1.compareTo(s2));// 0
		System.out.println(s1.compareTo(s3));// -1(because s1>s3)
		System.out.println(s3.compareTo(s1));// 1(because s3 < s1 )
	}

}
```

Concatenation

- By + (string concatenation) operator
- By concat() method

```
package stringInjava;

public class Example3 {

	public static void main(String[] args) {
		String s = 90 + 30 + "Bhanu" + 50 + 50;
```

```
		System.out.println(s);

		String s1 = "Bhanu ";
		String s2 = "Pratap";
		String s3 = s1.concat(s2);
		System.out.println(s3);
	}

}
```

Substring

You can get substring from the given string object by one of the two methods:

1. **public String substring(int startIndex):** This method returns new String object containing the substring of the given string from specified startIndex (inclusive).
2. **public String substring(int startIndex, int endIndex):** This method returns new String object containing the substring of the given string from specified startIndex to endIndex.

In case of string:

- **startIndex:** inclusive
- **endIndex:** exclusive

```
package stringInjava;
public class Example4 {

	public static void main(String[] args) {
		String s = "Bhanupratap";
		System.out.println(s.substring(6));
		System.out.println(s.substring(0, 6));
	}
}
```

charAt()

```
package stringInjava;
public class Example5 {
public static void main(String[] args) {
		String name = "bhanupratap";
		char ch = name.charAt(4);
		System.out.println(ch);
```

```
		name.charAt(20);
	}
}
```

contains()

```
package stringInjava;
public class Example7 {

	public static void main(String[] args) {
		String name = "Hello java program";
		System.out.println(name.contains("java program"));
		System.out.println(name.contains("am"));
		System.out.println(name.contains("hello"));
	}
}
package stringInjava;
public class Example8 {

	public static void main(String[] args) {
		String str = "if you want to move in automation learn java";
		if (str.contains("learn java")) {
			System.out.println("I am in");
		} else
			System.out.println("Result not found");
	}
}
```

endsWith()

```
package stringInjava;
public class Example9 {
	public static void main(String[] args) {
		String s1 = "bhanu test";
		System.out.println(s1.endsWith("t"));
		System.out.println(s1.endsWith("test1"));

		String str = "Welcome to youtube channel";
		if (str.endsWith("channel")) {
			System.out.println("Welcome to youtube channel");
		} else
			System.out.println("It does not end with channel");
	}
}
```

equals()

```
package stringInjava;
public class Example10 {
	public static void main(String[] args) {
		String s1 = "java";
		String s2 = "c";
		String s3 = "java";
		String s4 = "python";
		System.out.println(s1.equals(s2));
		System.out.println(s1.equals(s3));
		System.out.println(s1.equals(s4));
	}

}
```

equalsIgnoreCase()

```
package stringInjava;
public class Example11 {
	public static void main(String[] args) {
		String s1 = "bhanu";
		String s2 = "bhanu";
		String s3 = "BHANU";
		String s4 = "TEST";
		System.out.println(s1.equalsIgnoreCase(s2));
		System.out.println(s1.equalsIgnoreCase(s3));
		System.out.println(s1.equalsIgnoreCase(s4));
	}
}
```

format()

The **java string format()** method returns the formatted string by given locale, format and arguments.

Throws

NullPointerException : if format is null.
IllegalFormatException : if format is illegal or incompatible.

```
package stringInjava;
```

```
public class Example12 {
	public static void main(String[] args) {
		String name = "bhanu";
		String sf1 = String.format("name is %s", name);
		String sf2 = String.format("value is %f", 32.33434);
		String sf3 = String.format("value is %.12f", 32.33434);
		System.out.println(sf1);
		System.out.println(sf2);
		System.out.println(sf3);
	}
}
package stringInjava;
public class Example12 {
	public static void main(String[] args) {
		String name = "bhanu";
		String sf1 = String.format("name is %s", name);
		String sf2 = String.format("value is %f", 32.33434);
		String sf3 = String.format("value is %.12f", 32.33434);
		System.out.println(sf1);
		System.out.println(sf2);
		System.out.println(sf3);

		String str1 = String.format("%d", 104); // Integer value
		String str2 = String.format("%s", "Bhanu Pratap"); // String value
		String str3 = String.format("%f", 104.00); // Float value
		String str4 = String.format("%x", 104); // Hexadecimal value
		String str5 = String.format("%c", 'a'); // Char value
		System.out.println(str1);
		System.out.println(str2);
		System.out.println(str3);
		System.out.println(str4);
		System.out.println(str5);
```

example where we are setting width and padding for an integer value.

```
		str1 = String.format("%d", 101);
		str2 = String.format("|%10d|", 101);
		str3 = String.format("|%-10d|", 101);
		str4 = String.format("|% d|", 101);
		str5 = String.format("|%010d|", 101);
		System.out.println(str1);
		System.out.println(str2);
		System.out.println(str3);
```

```
		System.out.println(str4);
		System.out.println(str5);
	}
}
```

getBytes()

Method returns the byte array of the string

```
package stringInjava;
public class Example13 {

	public static void main(String[] args) {
		String s1 = "ABCDEFG";
		byte[] barr = s1.getBytes();
		for (int i = 0; i < barr.length; i++) {
			System.out.println(barr[i]);
		}
	}
}
```

getChars()

Method copies the content of this string into specified char array

```
public void getChars(int srcBeginIndex, int srcEndIndex, char[] destination, int dstBeginInde
x)
```

Throws

It throws StringIndexOutOfBoundsException if beginIndex is greater than endIndex.

```
package stringInjava;
public class Example14 {
	public static void main(String[] args) {
		String str = new String("java is programming language");
		char[] ch = new char[10];
		try {
			str.getChars(5, 15, ch, 0);
			System.out.println(ch);
		} catch (Exception ex) {
			System.out.println(ex);
		}
	}
```

```
}
```

java.lang.ArrayIndexOutOfBoundsException

```
package stringInjava;
public class Example15 {
	public static void main(String[] args) {
		String str = new String("java is programming language");
		char[] ch = new char[10];
		try {
			str.getChars(5, 20, ch, 0);
			System.out.println(ch);
		} catch (Exception ex) {
			System.out.println(ex);
		}
	}
}
```

indexOf()

Method	Description
int indexOf(int ch)	returns index position for the given char value
int indexOf(int ch, int fromIndex)	returns index position for the given char value and from index
int indexOf(String substring)	returns index position for the given substring
int indexOf(String substring, int fromIndex)	returns index position for the given substring and from index

```
package stringInjava;
public class Example16 {
	public static void main(String[] args) {
		String s1 = "hello java program";
		int index1 = s1.indexOf("ja");
		int index2 = s1.indexOf("java");
		System.out.println(index1);
		System.out.println(index2);

		int index3 = s1.indexOf("ro", 4);
		System.out.println(index3);

		int index4 = s1.indexOf('a');
		System.out.println(index4);
	}
```

```
}
```

intern()

```
package stringInjava;
public class Example17 {
	public static void main(String[] args) {
		String s1 = new String("java");
		String s2 = "java";
		String s3 = s1.intern();
		System.out.println(s1 == s2);
		System.out.println(s2 == s3);
	}
}
```

```
package stringInjava;
public class Example18 {
	public static void main(String[] args) {
		String s1 = "Java";
		String s2 = s1.intern();
		String s3 = new String("Java");
		String s4 = s3.intern();
		System.out.println(s1 == s2);
		System.out.println(s1 == s3);
		System.out.println(s1 == s4);
		System.out.println(s2 == s3);
		System.out.println(s2 == s4);
		System.out.println(s3 == s4);
	}
}
```

isEmpty()

```
package stringInjava;
public class Example19 {
	public static void main(String[] args) {
		String s1 = "";
		String s2 = "javatest";

		System.out.println(s1.isEmpty());
		System.out.println(s2.isEmpty());
	}
```

```
}
```

join()

```
package stringInjava;
public class Example20 {
	public static void main(String[] args) {
		String s1 = String.join("-", "hello", "java", "program");
		System.out.println(s1);
	}
}
package stringInjava;
public class Example21 {
	public static void main(String[] args) {
		String d = String.join("/", "20", "04", "2016");
		System.out.print(d);
		String t = String.join(":", "11", "13", "10");
		System.out.println(t);
	}

}
```

lastIndexOf()

Method	Description
int lastIndexOf(int ch)	returns last index position for the given char value
int lastIndexOf(int ch, int fromIndex)	returns last index position for the given char value and from index
int lastIndexOf(String substring)	returns last index position for the given substring
int lastIndexOf(String substring, int fromIndex)	returns last index position for the given substring and from index

```
package stringInjava;
public class Example22 {
	public static void main(String[] args) {
		String s1 = "this is java index";
		int index = s1.lastIndexOf('s');
		System.out.println(index);

		s1.lastIndexOf('s', 5);
		System.out.println(index);
```

```
		s1.lastIndexOf("of");
		System.out.println(index);
	}
}
```

length()

```
package stringInjava;
public class Example23 {
	public static void main(String args[]) {
		String s1 = "bhanujava";
		String s2 = "test";
		System.out.println("string length is: " + s1.length());
		System.out.println("string length is: " + s2.length());
	}

}
```

replace()

```
package stringInjava;
public class Example24 {
	public static void main(String args[]) {
		String s1 = "hello java test";
		String replaceString = s1.replace('a', 'e');
		System.out.println(replaceString);
	}
}
```

split()

```
package stringInjava;
public class Example25 {
	public static void main(String args[]) {
		String s1 = "hello java test";
		String[] words = s1.split("\\s");
		for (String w : words) {
			System.out.println(w);
		}

		s1 = "hello:java:test";
		words = s1.split(":");
		for (String w : words) {
```

```
                        System.out.println(w);
                }
        }
}
```

startsWith()

```
package stringInjava;
public class Example26 {
        public static void main(String[] args) {
                String s1 = "hello java test program";
                System.out.println(s1.startsWith("hel"));
                System.out.println(s1.startsWith("hello java"));

                 System.out.println(s1.startsWith("a",7));

                 System.out.println(s1.startsWith("a",40));
        }
}
```

substring()

Parameters

startIndex : starting index is inclusive
endIndex : ending index is exclusive

```
package stringInjava;
public class Example27 {
        public static void main(String[] args) {
                String s1 = "hellojavatest";
                String substr = s1.substring(0);
                System.out.println(substr);
                String substr2 = s1.substring(5, 10);
                System.out.println(substr2);

                // java.lang.StringIndexOutOfBoundsException:
                String substr3 = s1.substring(5, 15);
        }
}
```

toCharArray()

```java
package stringInjava;
public class Example28 {
	public static void main(String[] args) {
		String s1 = "hello java test";
		char[] ch = s1.toCharArray();
		int len = ch.length;
		System.out.println("Char Array length: " + len);
		System.out.println("Char Array elements: ");
		for (int i = 0; i < len; i++) {
			System.out.println(ch[i]);
		}
	}
}
```

toLowerCase()
toUpperCase()

```java
package stringInjava;
public class Example29 {
	public static void main(String[] args) {
		String s = "hello java";

		System.out.println(s.toUpperCase());

		String s1 = "HELLO";

		System.out.println(s1.toLowerCase());
	}
}
```

trim()

```java
package stringInjava;
public class Example30 {
	public static void main(String[] args) {
		String s1 = " Hello Java ";

		System.out.println("123"+s1.trim()+"12");


		System.out.println("123"+s1+"12");
	}
}
```

valueOf()

The **java string valueOf()** method converts different types of values into string.

- **public static** String valueOf(**boolean** b)
- **public static** String valueOf(**char** c)
- **public static** String valueOf(**char**[] c)
- **public static** String valueOf(**int** i)
- **public static** String valueOf(**long** l)
- **public static** String valueOf(**float** f)
- **public static** String valueOf(**double** d)
- **public static** String valueOf(Object o)

```
package stringInjava;
public class Example31 {
	public static void main(String[] args) {
		float f = 10.05f;
		double d = 10.02;
		String s1 = String.valueOf(f);
		String s2 = String.valueOf(d);
		System.out.println(s1);
		System.out.println(s2);

		char ch1 = 'A';
		char ch2 = 'B';
		s1 = String.valueOf(ch1);
		s2 = String.valueOf(ch2);
		System.out.println(s1);
		System.out.println(s2);

		int value = 30;
		s1 = String.valueOf(value);
		System.out.println(s1);
	}

}
```

Java String Buffer class

Java StringBuffer class is used to create mutable (modifiable) string.

Constructor	Description
StringBuffer()	creates an empty string buffer with the initial capacity of 16.
StringBuffer(String str)	creates a string buffer with the specified string.
StringBuffer(int capacity)	creates an empty string buffer with the specified capacity as length.

Type	Method	Description
public synchronized StringBuffer	append(String s)	is used to append the specified string with this string. The append() method is overloaded like append(char), append(boolean), append(int), append(float), append(double) etc.
public synchronized StringBuffer	insert(int offset, String s)	is used to insert the specified string with this string at the specified position. The insert() method is overloaded like insert(int, char), insert(int, boolean), insert(int, int), insert(int, float), insert(int, double) etc.
public synchronized StringBuffer	replace(int startIndex, int endIndex, String str)	is used to replace the string from specified startIndex and endIndex.
public synchronized StringBuffer	delete(int startIndex, int endIndex)	is used to delete the string from specified startIndex and endIndex.
public synchronized StringBuffer	reverse()	is used to reverse the string.
public int	capacity()	is used to return the current capacity.
public void	ensureCapacity(int minimumCapacity)	is used to ensure the capacity at least equal to the given minimum.
public char	charAt(int index)	is used to return the character at the specified position.
public int	length()	is used to return the length of the string i.e. total number of characters.
public String	substring(int beginIndex)	is used to return the substring from the specified beginIndex.
public String	substring(int beginIndex, int endIndex)	is used to return the substring from the specified beginIndex and endIndex.

By Bhanu Pratap Singh
https://www.youtube.com/user/MrBhanupratap29/playlists
https://www.facebook.com/learnbybhanupratap/
https://www.udemy.com/seleniumbybhanu/

```
package stringBuffer;

public class Example1 {

	public static void main(String[] args) {

		StringBuffer sb = new StringBuffer("hello ");
		sb.append("Java");
		System.out.println(sb);

		// insert() method

		StringBuffer sb1 = new StringBuffer("Hello ");
		sb1.insert(1, "Java");
		System.out.println(sb1);

		// replace() method
		StringBuffer sb2 = new StringBuffer("Hello");
		sb2.replace(1, 3, "Java");
		System.out.println(sb2);

		// delete() method
		StringBuffer sb3 = new StringBuffer("Hello");
		sb3.delete(1, 3);
		System.out.println(sb3);

		// capacity() method

		StringBuffer sb4 = new StringBuffer();
		System.out.println(sb4.capacity());// default 16
		sb4.append("Hello");
		System.out.println(sb4.capacity());// now 16
		sb4.append("java is my favourite language");
		System.out.println(sb4.capacity());// now (16*2)+2=34 i.e //
(oldcapacity*2)+2
	}

}
package stringBuffer;

public class Example2 {

	public static void main(String args[]) {
```

```
                StringBuffer sb = new StringBuffer();
                System.out.println(sb.capacity());// default 16
                sb.append("Hello");
                System.out.println(sb.capacity());// now 16
                sb.append("java is programming language");
                System.out.println(sb.capacity());// now (16*2)+2=34 i.e
                                                                        //
(oldcapacity*2)+2
                sb.ensureCapacity(10);// now no change
                System.out.println(sb.capacity());// now 34
                sb.ensureCapacity(50);// now (34*2)+2
                System.out.println(sb.capacity());// now 70
        }
}
```

Java StringBuilder class

Java StringBuilder class is used to create mutable (modifiable) string. The Java StringBuilder class is same as StringBuffer class except that it is non-synchronized.

Constructor	Description
StringBuilder()	creates an empty string Builder with the initial capacity of 16.
StringBuilder(String str)	creates a string Builder with the specified string.
StringBuilder(int length)	creates an empty string Builder with the specified capacity as length.

Performance Test of String and StringBuffer

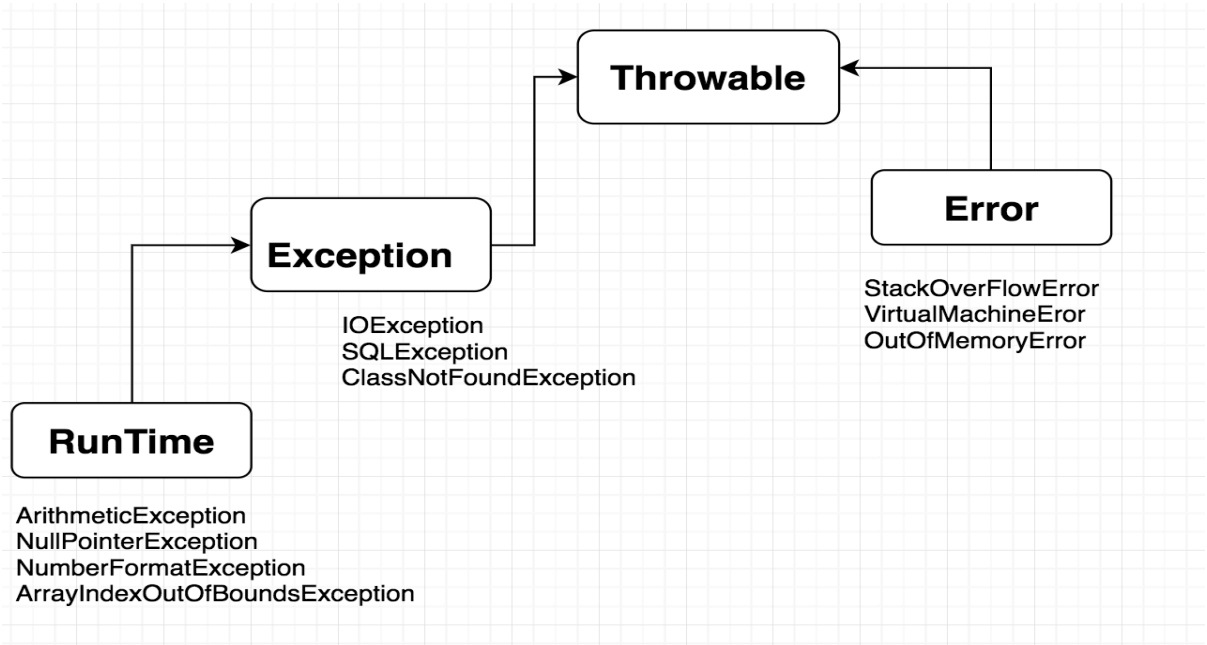
```
package stringBuffer;
public class Example3 {
        public static String concatString() {
                String t = "Hello";
                for (int i = 0; i < 10000; i++) {
                        t = t + "Java";
                }
                return t;
        }

        public static String concatStringBuffer() {
                StringBuffer sb = new StringBuffer("Hello");
                for (int i = 0; i < 10000; i++) {
                        sb.append("Java");
```

By Bhanu Pratap Singh
https://www.youtube.com/user/MrBhanupratap29/playlists
https://www.facebook.com/learnbybhanupratap/
https://www.udemy.com/seleniumbybhanu/

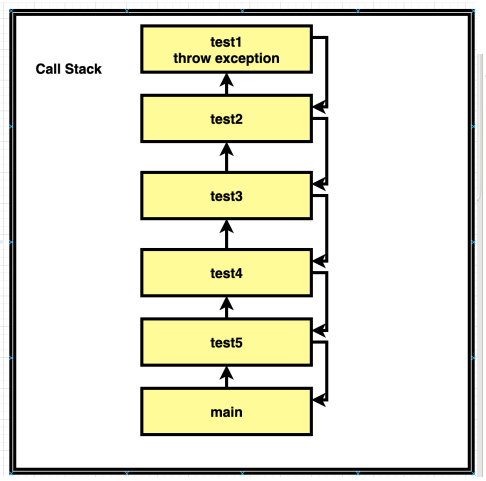
```
		}
		return sb.toString();
	}

	public static void main(String[] args) {
		long startTime = System.currentTimeMillis();
		Example3.concatString();
		long endTime = System.currentTimeMillis();
		System.out.println("Time Take my string=" + (endTime - startTime));

		long startTime1 = System.currentTimeMillis();
		Example3.concatStringBuffer();
		long endTime1 = System.currentTimeMillis();

		System.out.println("Time Take my string=" + (endTime1 - startTime1));
	}

}
```

Exception in Java

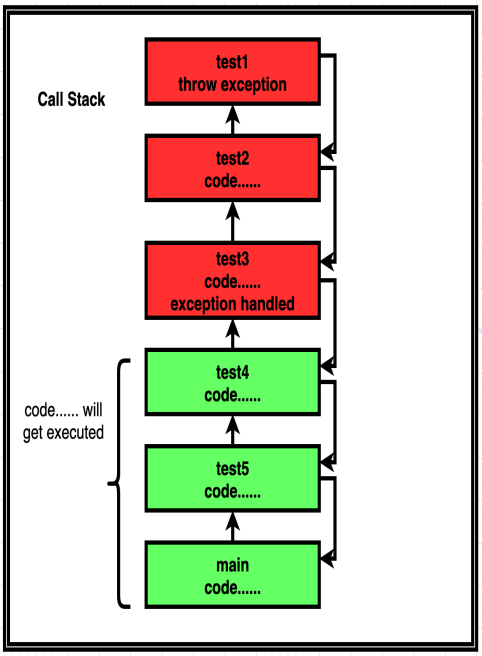
Exceptions are events that occur during the execution of programs that disrupt the normal flow of instructions (e.g. divide by zero, array access out of bound, etc.). In **Java**, an **exception** is an object that wraps an error event that occurred within a method and contains: Information about the error including its type.

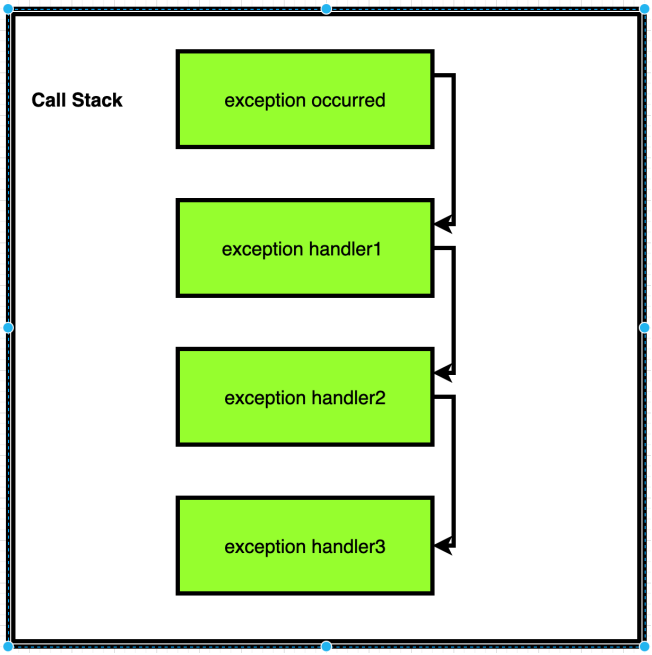


By Bhanu Pratap Singh
https://www.youtube.com/user/MrBhanupratap29/playlists
https://www.facebook.com/learnbybhanupratap/
https://www.udemy.com/seleniumbybhanu/




```
<terminated> ExceptionTest [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_231.jdk/Contents/Home/bin/java (03-Apr-2020, 11:00:
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at ExceptionTest.tes1(ExceptionTest.java:5)
        at ExceptionTest.tes2(ExceptionTest.java:9)
        at ExceptionTest.tes3(ExceptionTest.java:13)
        at ExceptionTest.tes4(ExceptionTest.java:17)
        at ExceptionTest.tes5(ExceptionTest.java:21)
        at ExceptionTest.main(ExceptionTest.java:26)
```





How JVM handle an Exception?

Default Exception Handling : Whenever inside a method, if an exception has occurred, the method creates an Object known as Exception Object and hands it off to the run-time system(JVM). The exception object contains name and description of the exception, and current state of the program where exception has occurred. Creating the Exception Object and handling it to the run-time system is called throwing an Exception.There might be the list of the methods that had been called to get to the method where exception was occurred. This ordered list of the methods is called **Call Stack**.Now the following procedure will happen.

- The run-time system searches the call stack to find the method that contains block of code that can handle the occurred exception. The block of the code is called **Exception handler**.

By Bhanu Pratap Singh
https://www.youtube.com/user/MrBhanupratap29/playlists
https://www.facebook.com/learnbybhanupratap/
https://www.udemy.com/seleniumbybhanu/

- The run-time system starts searching from the method in which exception occurred, proceeds through call stack in the reverse order in which methods were called.
- If it finds appropriate handler then it passes the occurred exception to it. Appropriate handler means the type of the exception object thrown matches the type of the exception object it can handle.
- If run-time system searches all the methods on call stack and couldn't have found the appropriate handler then run-time system handover the Exception Object to **default exception handler** , which is part of run-time system. This handler prints the exception information in the following format and terminates program **abnormally**.

Code without and without exception handling

```
package exception;
public class Example1 {
	public void test1() {
		int i = 90 / 0;
		System.out.println("execution completed");
	}

	public void test2() {
		try {
			int i = 90 / 0;
		} catch (Exception e) {
			System.out.println("exception occured");
		}
		System.out.println("execution completed");
	}

	public static void main(String[] args) {
		Example1 obj = new Example1();
		//obj.test1();
		obj.test2();
	}

}
```

Types of Exception

Built-in exceptions are the exceptions which are available in Java libraries. These exceptions are suitable to explain certain error situations. Below is the list of important built-in exceptions in Java.

1. **Arithmetic Exception**
   It is thrown when an exceptional condition has occurred in an arithmetic operation.

2. **ArrayIndexOutOfBoundException**
   It is thrown to indicate that an array has been accessed with an illegal index. The index is either negative or greater than or equal to the size of the array.
3. **ClassNotFoundException**
   This Exception is raised when we try to access a class whose definition is not found
4. **FileNotFoundException**
   This Exception is raised when a file is not accessible or does not open.
5. **IOException**
   It is thrown when an input-output operation failed or interrupted
6. **InterruptedException**
   It is thrown when a thread is waiting , sleeping , or doing some processing , and it is interrupted.
7. **NoSuchFieldException**
   It is thrown when a class does not contain the field (or variable) specified
8. **NoSuchMethodException**
   It is thrown when accessing a method which is not found.
9. **NullPointerException**
   This exception is raised when referring to the members of a null object. Null represents nothing
10. **NumberFormatException**
    This exception is raised when a method could not convert a string into a numeric format.
11. **RuntimeException**
    This represents any exception which occurs during runtime.
12. **StringIndexOutOfBoundsException**
    It is thrown by String class methods to indicate that an index is either negative than the size of the string

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
	at exception.Example2.main(Example2.java:6)
```

```
package exception;
public class Example2 {
	public static void main(String[] args) {
		int i = 90 / 0;
	}

}
```

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 8
	at exception.Example3.main(Example3.java:7)
```

```
package exception;
```

```
public class Example3 {
	public static void main(String[] args) {
		int a[] = { 10, 20, 30 };
		System.out.println(a[8]);
	}
}
```

```
Exception in thread "main" java.lang.NullPointerException
	at exception.Example4.main(Example4.java:7)
```

```
package exception;
public class Example4 {
	public static void main(String[] args) {
		String s1 = null;
		System.out.println(s1.toString());
	}
}
```

Checked Exception Example

```
Exception in thread "main" java.lang.ClassNotFoundException: Test12
	at java.net.URLClassLoader.findClass(URLClassLoader.java:381)
	at java.lang.ClassLoader.loadClass(ClassLoader.java:424)
```

```
package exception;
public class Example5 {
	public static void main(String[] args) throws ClassNotFoundException {
		Class.forName("Test12");
	}
}
```

IOException is checked exception

```
Exception in thread "main" java.io.FileNotFoundException:  (No such file or directory)
	at java.io.FileInputStream.open0(Native Method)
```

```
package exception;
import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;

public class Example6 {
	public static void main(String[] args) throws IOException {
		BufferedReader rd = new BufferedReader(new FileReader(new File("")));
		rd.readLine();
```

```
        }
}
```

How Programmer handles an exception?

Customized Exception Handling : Java exception handling is managed via five keywords: **try**, **catch**, **throw, throws, and finally**. Briefly, here is how they work. Program statements that you think can raise exceptions are contained within a try block. If an exception occurs within the try block, it is thrown. Your code can catch this exception (using catch block) and handle it in some rational manner. System-generated exceptions are automatically thrown by the Java run-time system. To manually throw an exception, use the keyword throw. Any exception that is thrown out of a method must be specified as such by a throws clause. Any code that absolutely must be executed after a try block completes is put in a finally block.

Try, Catch and Finally syntax

```
try {
        // block of code to monitor for errors
        // the code you think can raise an exception
        }
        catch (ExceptionType1 ex) {
        // exception handler for ExceptionType1
        }
        catch (ExceptionType2 ex) {
        // exception handler for ExceptionType2
        }
        // optional
        finally {
        // block of code to be executed after try block ends
        }
```

Points to remember :

- In a method, there can be more than one statements that might throw exception, So put all these statements within its own **try** block and provide separate exception handler within own **catch** block for each of them.
- If an exception occurs within the **try** block, that exception is handled by the exception handler associated with it. To associate exception handler, we must put **catch** block after it. There can be more than one exception handlers. Each **catch** block is a exception handler that handles the exception of the type indicated by its argument. The argument, Exception Type declares the type of the exception that it can handle and must be the name of the class that inherits from **Throwable** class.

By Bhanu Pratap Singh
https://www.youtube.com/user/MrBhanupratap29/playlists
https://www.facebook.com/learnbybhanupratap/
https://www.udemy.com/seleniumbybhanu/

- For each try block there can be zero or more catch blocks, but **only one** finally block.
- The finally block is optional. It always gets executed whether an exception occurred in try block or not . If exception occurs, then it will be executed after **try and catch blocks.** And if exception does not occur then it will be executed after the **try** block. The finally block in java is used to put important codes such as clean up code e.g. closing the file or closing the connection.

```
package exception;
public class Example7 {
	public void handelException() {
		try {
			int i = 90 / 0;
		} catch (Exception e) {
			System.out.println("exception handeled");
		}
		System.out.println("code execution completed");
	}

	public void donotHandelException() {
		int i = 90 / 0;
		System.out.println("code execution completed");
	}

	public static void main(String[] args) {
		Example7 obj = new Example7();
		obj.handelException();
		obj.donotHandelException();
	}
}
```

Handel Exception only for NumberFormatException

```
package exception;
public class Example8 {
	public void handelException() {
		try {
			int i = 90 / 1;
			int a[] = { 20 };
			int k = a[2];
		} catch (ArithmeticException e) {
			System.out.println("exception handeled");
		}
		System.out.println("code execution completed");
	}
```

By Bhanu Pratap Singh
https://www.youtube.com/user/MrBhanupratap29/playlists
https://www.facebook.com/learnbybhanupratap/
https://www.udemy.com/seleniumbybhanu/

```
	public static void main(String[] args) {
		Example8 obj = new Example8();
		obj.handelException();
	}
}
```

Handel Exception for NumberFormatException and ArrayIndexOutOfBoundsException

```
package exception;
public class Example9 {
	public void handelException() {
		try {
			int i = 90 / 1;
			int a[] = { 20 };
			int k = a[2];
		} catch (ArithmeticException e) {
			System.out.println("ArithmeticException handeled");
		} catch (ArrayIndexOutOfBoundsException e) {
			System.out.println("ArrayIndexOutOfBoundsException handeled");
		}
		System.out.println("code execution completed");
	}

	public static void main(String[] args) {
		Example9 obj = new Example9();
		obj.handelException();
	}
}
```

Handel Individual and all type of exception

```
package exception;
public class Example10 {
	public void handelException() {
		try {
			int i = 90 / 1;
			int a[] = { 20 };
			int k = a[0];

			String s1 = null;
			System.out.println(s1.length());
		} catch (ArithmeticException e) {
			System.out.println("ArithmeticException handeled");
		} catch (ArrayIndexOutOfBoundsException e) {
			System.out.println("ArrayIndexOutOfBoundsException handeled");
```

```
		} catch (Exception e) {
			System.out.println("Exception=" + e.getClass().getName());
		}
		System.out.println("code execution completed");
	}

	public static void main(String[] args) {
		Example10 obj = new Example10();
		obj.handelException();
	}
}
```

Nested Try Catch

```
package exception;
public class Example11 {
	public void test1() {
		int i = 90 / 1;
	}

	public void test2() {
		int a[] = { 20 };
		int k = a[1];
	}

	public void test3() {
		try {
			test1();
			System.out.println("No exception is test1() method");
			try {
				test2();
				System.out.println("No exception is test2() method");
			} catch (ArrayIndexOutOfBoundsException e) {
				System.out.println("ArrayIndexOutOfBoundsException
handeled");
			}
		} catch (ArithmeticException e) {
			System.out.println("ArithmeticException handeled");
		}
	}

	public static void main(String[] args) {
		Example11 obj = new Example11();
		obj.test3();
	}
```

```
}
```

```
package exception;

public class Example12 {

	public static void main(String args[]) {
		try {
			try {
				try {
					int arr[] = { 1, 2, 3, 4 };
					System.out.println(arr[10]);
				} catch (ArithmeticException e) {
					System.out.print("Arithmetic Exception");
					System.out.println(" handled in Third try-block");
				}
			} catch (ArithmeticException e) {
				System.out.print("Arithmetic Exception");
				System.out.println(" handled in Second try-block");
			}
		} catch (ArithmeticException e3) {
			System.out.print("Arithmetic Exception");
			System.out.println(" handled in main try-block");
		} catch (ArrayIndexOutOfBoundsException e4) {
			System.out.print("ArrayIndexOutOfBoundsException");
			System.out.println(" handled in main try-block");
		} catch (Exception e5) {
			System.out.print("Exception");
			System.out.println(" handled in main try-block");
		}
	}
}
```

```
package exception;
public class Example13 {
	public static void main(String args[]) {
		// Parent try block
		try {
			// Child try block1
			try {
				System.out.println("Inside block1");
				int b = 40 / 0;
				System.out.println(b);
```

```
                } catch (ArithmeticException e) {
                        System.out.println("Exception in Child try block1");
                }
                // Child try block2
                try {
                        System.out.println("Inside block2");
                        int b = 40 / 0;
                        System.out.println(b);
                } catch (ArrayIndexOutOfBoundsException e) {
                        System.out.println("Exception in Child try block2");
                }
                System.out.println("Just other statement");
        } catch (ArithmeticException e) {
                System.out.println("Arithmetic Exception");
                System.out.println("Inside parent try catch block");
        } catch (ArrayIndexOutOfBoundsException e) {
                System.out.println("ArrayIndexOutOfBoundsException");
                System.out.println("Inside parent try catch block");
        } catch (Exception e5) {
                System.out.println("Exception");
                System.out.println("Inside parent try catch block");
        }
        System.out.println("Hello java");
    }
}
```

A **finally block** contains all the crucial statements that must be executed whether exception occurs or not. The statements present in this block will always execute regardless of whether exception occurs in try block or not such as closing a connection, stream etc.

```
package exception;
public class Example14 {
    public static void main(String[] args) {
        try {
            int num = 9 / 0;
            System.out.println(num);
        } catch (ArithmeticException e) {
            System.out.println("Number should not be divided by zero");
        }
        /*
         * Finally block will always execute even if there is no exception in
         * try block
         */
        finally {
            System.out.println("This is finally block");
```

```
		}
		System.out.println("Out of try-catch-finally");
	}
}
```

Finally will get executed even if there is exception in program and it is not handled.

```
package exception;
public class Example15 {
	public static void main(String[] args) {
		try {
			int num = 9 / 0;
			System.out.println(num);
		} catch (NumberFormatException e) {
			System.out.println("Number should not be divided by zero");
		}
		finally {
			System.out.println("This is finally block");
		}
		System.out.println("Out of try-catch-finally");
	}
}
```

```
package exception;
public class Example16 {
	public static void main(String[] args) {
		try {
			int num = 9 / 0;
			System.out.println(num);
		} catch (ArithmeticException e) {
			System.out.println("Number should not be divided by zero");
		} finally {
			int num = 9 / 0;
			System.out.println(num);
			System.out.println("This is finally block");
		}
	}
}
```

```
package exception;
public class Example17 {
	public static void main(String[] args) {
		try {
			int num = 9 / 1;
			System.out.println(num);
			return;
```

```
		} catch (ArithmeticException e) {
			System.out.println("Number should not be divided by zero");
		} finally {
			System.out.println("This is finally block");
		}
	}
}
```

throw exception in java

We can define our own set of conditions or rules and throw an exception explicitly using throw keyword. For example, we can throw ArithmeticException when we divide number by 5, or any other numbers, what we need to do is just set the condition and throw any exception using throw keyword. Throw keyword can also be used for throwing custom exceptions,

```
package exception;
public class Example18 {

	static void checkEligibiltyProcess(int age, int weight) {
		if (age < 10 && weight < 30) {
			throw new ArithmeticException("Student is not eligible for
registration");
		} else {
			System.out.println("Student is eligible for registration");
		}
	}

	public static void main(String args[]) {
		System.out.println("Welcome to the Admission process");
		checkEligibiltyProcess(10, 39);
		System.out.println("Have a good day");

		checkEligibiltyProcess(9, 25);
	}
}
```

```
Welcome to the Admission process
Student is eligible for registrationException in thread "main"
Have a good day
java.lang.ArithmeticException: Student is not eligible for registration
	at exception.Example18.checkEligibiltyProcess(Example18.java:7)
	at exception.Example18.main(Example18.java:18)
package exception;

public class Example19 {
```

```
	static void checkEligibiltyProcess(int age) {
		if (age < 18) {
			throw new ArithmeticException("Not Eligible for voting");
		} else {
			System.out.println("Eligible for voting");
		}
	}

	public static void main(String[] args) {
		Example19.checkEligibiltyProcess(20);
	}

}
```

throws Keyword

Any method that is capable of causing exceptions must list all the exceptions possible during its execution, so that anyone calling that method gets a prior knowledge about which exceptions are to be handled. A method can do so by using the **throws** keyword.

```
package exception;
public class Example20 {
	static void check() throws ArithmeticException {
		System.out.println("Inside check function");
		throw new ArithmeticException("demo");
	}

	public static void main(String args[]) {
		try {
			check();
		} catch (ArithmeticException e) {
			System.out.println("caught" + e);
		}
	}

}
package exception;
import java.io.IOException;
public class Example21 {

	void test() throws IOException {
		throw new IOException("device error");// checked exception
	}
```

```
	void tset2() throws IOException {
		test();
	}

	void tset1() {
		try {
			tset2();
		} catch (Exception e) {
			System.out.println("exception handled");
		}
	}

	public static void main(String args[]) {
		Example21 obj = new Example21();
		obj.tset1();
		System.out.println("normal flow...");
	}
}
package exception;
public class Example22 {
	void test1() throws ArithmeticException {
		throw new ArithmeticException("Calculation error");
	}

	void test2() throws ArithmeticException {
		test1();
	}

	void test3() {
		try {
			test2();
		} catch (ArithmeticException e) {
			System.out.println("ArithmeticException handled");
		}
	}

	public static void main(String args[]) {
		Example22 obj = new Example22();
		obj.test3();
		System.out.println("End Of Program");
	}
}
package exception;
public class Example23 {
```

```
	void method() throws ArithmeticException {
		throw new ArithmeticException("ArithmeticException Occurred");
	}
}

package exception;
class Example24 {
	public static void main(String args[]) throws ArithmeticException {
		Example23 obj = new Example23();
		obj.method();

		System.out.println("End Of Program");
	}
}
```

Throw exception in catch block

```
package exception;
public class Example25 {
	static void fun() {
		try {
			throw new NullPointerException("demo");
		} catch (NullPointerException e) {
			System.out.println("Caught inside fun().");
			throw e;
		}
	}

	public static void main(String args[]) {
		try {
			fun();
		} catch (NullPointerException e) {
			System.out.println("Caught in main.");
		}
	}
}
```

Important points to remember about throws keyword:

- throws keyword is required only for checked exception and usage of throws keyword for unchecked exception is meaningless.
- throws keyword is required only to convince compiler and usage of throws keyword does not prevent abnormal termination of program.

- By the help of throws keyword we can provide information to the caller of the method about the exception.

Custom Exception

```
package customException;
import java.io.Serializable;
public class DataException extends Exception implements Serializable {

	/**
	 *
	 */
	private static final long serialVersionUID = 1L;

	DataException(String msg, Throwable t) {
		super(msg, t);
	}

	DataException(String msg) {
		super(msg);
	}

}
package customException;
public class TestDataException {
	public static void test1(int a) throws DataException {
		if (a < 10) {
			throw new DataException("data is not valid");
		}
	}

	public static void test2() throws DataException {
		try {
			int i = 9 / 0;
		} catch (ArithmeticException e) {
			throw new DataException("data is not valid", e);
		}
	}

	public static void main(String[] args) throws DataException {
		TestDataException.test2();
	}

}
package customException;
```

```
import java.io.Serializable;
public class DAOException extends Exception implements Serializable {

	private ErrorCode erroCode;

	public ErrorCode getErroCode() {
		return erroCode;
	}

	/**
	 *
	 */
	private static final long serialVersionUID = 1L;

	DAOException(String msg, Throwable t) {
		super(msg, t);
	}

	DAOException(String msg) {
		super(msg);
	}

	DAOException(ErrorCode erroCode,String msg, Throwable t) {
		super(msg, t);
		this.erroCode = erroCode;
	}
}

package customException;
public enum ErrorCode {

	INVALID_DATA,
	INVALID_STATAE,
	INVALID_CITY;

}

package customException;
public class TestException {

	public static void test1(int a) throws DAOException{
		if(a<10){
			throw new DAOException("data is not valid");
		}
```

```
	}

	public static void test2() throws DAOException{
		try {
			int i = 9/0;
		} catch (ArithmeticException e) {
			throw new DAOException("data is not valid", e);
		}
	}

	public static void test3() throws DAOException{
		try {
			int i = 9/0;
		} catch (ArithmeticException e) {
			throw new DAOException(ErrorCode.INVALID_DATA,"data is not
valid", e);
		}
	}

	public static void main(String[] args) throws DAOException{
		TestException.test2();
		TestException.test3();
	}
}
```

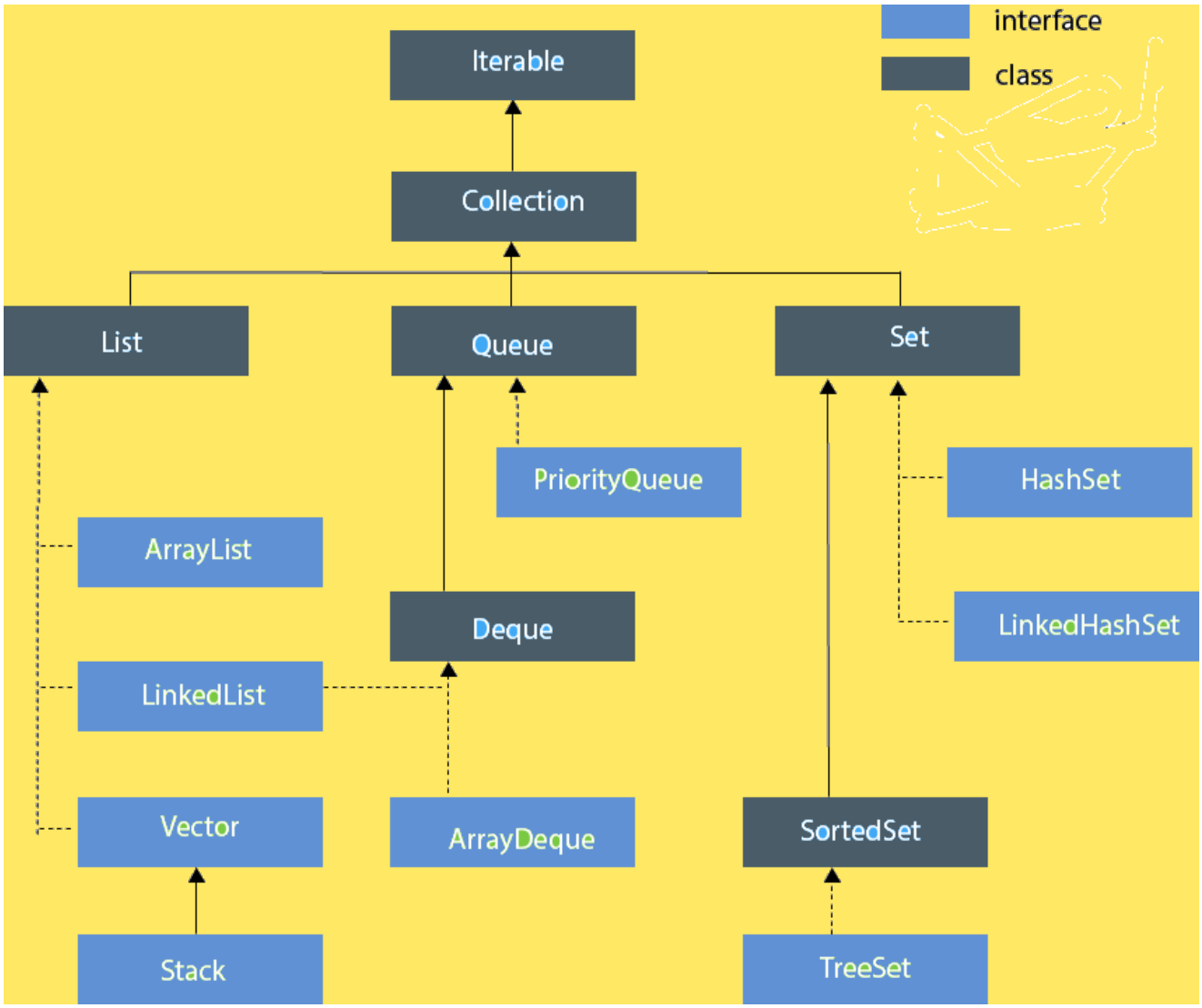
Difference between throw and throws

throw	throws
throw keyword is used to throw an exception explicitly.	throws keyword is used to declare an exception possible during its execution.
throw keyword is followed by an instance of Throwable class or one of its sub-classes.	throws keyword is followed by one or more Exception class names separated by commas.
throw keyword is declared inside a method body.	throws keyword is used with method signature (method declaration).
We cannot throw multiple exceptions using throw keyword.	We can declare multiple exceptions (separated by commas) using throws keyword.

Collections in Java

A Collection is a group of individual objects represented as a single unit. Java provides Collection Framework which defines several classes and interfaces to represent a group of objects as a single unit.
The Collection interface (**java.util.Collection**) and Map interface (**java.util.Map**) are the two main “root” interfaces of Java collection classes.

Hierarchy of Collection Framework

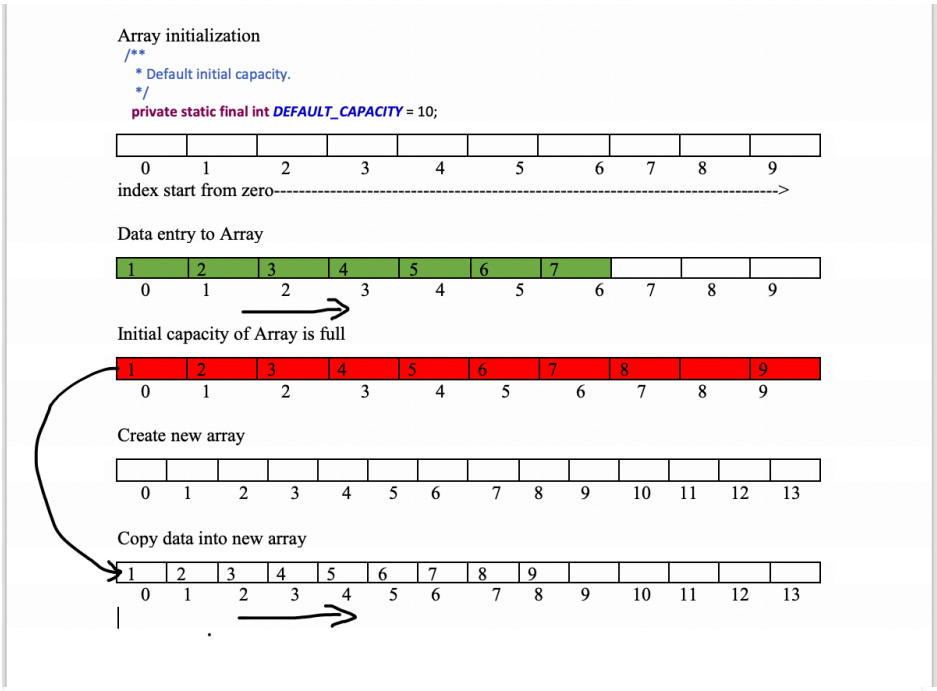


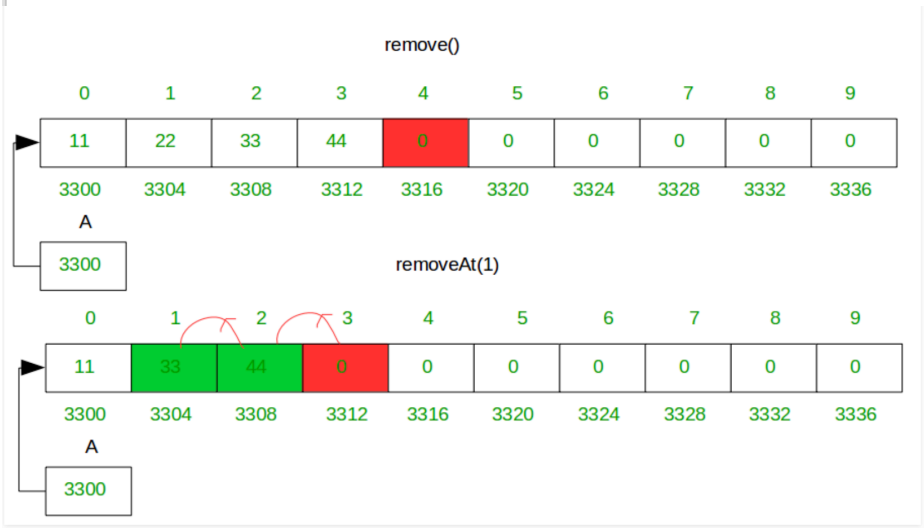
An **ArrayList in Java** represent a resizable list of objects. We can add, remove, find, sort and replace elements in this list. ArrayList is part of Java's collection framework and implements Java's **List** interface.

Hierarchy of ArrayList class

Java ArrayList class extends **AbstractList** class which implements **List** interface. The List interface extends **Collection** and **Iterable** interfaces in hierarchical order.

By Bhanu Pratap Singh
https://www.youtube.com/user/MrBhanupratap29/playlists
https://www.facebook.com/learnbybhanupratap/
https://www.udemy.com/seleniumbybhanu/





Array List Features

Array List has following features –

- Ordered – Elements in arraylist preserve their ordering which is by default the order in which they were added to the list.
- Index based – Elements can be randomly accessed using index positions. Index start with '0'.
- Dynamic resizing – ArrayList grows dynamically when more elements needs to be added than it's current size.
- Non synchronized – ArrayList is not synchronized, by default. Programmer needs to use synchronized keyword appropriately or simply use Vector class.

By Bhanu Pratap Singh
https://www.youtube.com/user/MrBhanupratap29/playlists
https://www.facebook.com/learnbybhanupratap/
https://www.udemy.com/seleniumbybhanu/

- Duplicates allowed – We can add duplicate elements in array list. It is not possible in sets.

```
package arrayList;
import java.util.ArrayList;
public class Example1 {

	public static void main(String[] args) {
		ArrayList<String> list = new ArrayList<>();
		list.add("Test1");
		list.add("Test2");
		list.add("Test3");
		list.add("Test4");
		list.add("Test5");
	}

}
```

Addition and removal of data from array list

```
package arrayList;
import java.util.ArrayList;
public class Example2 {

	public static void main(String[] args) {
		ArrayList<String> list = new ArrayList<>();
		list.add("Test1");
		list.add("Test2");
		list.add("Test3");
		list.add("Test4");
		list.add("Test5");
		System.out.println(list);
		list.remove(1);
		list.remove(1);
		list.remove(1);
		list.remove(1);
		System.out.println("After data removal from list");
		System.out.println(list);
	}

}
```

Iterate Array List

```
package arrayList;
import java.util.ArrayList;
import java.util.Iterator;
```

```
public class Example3 {

	public static void main(String[] args) {
		ArrayList<String> list = new ArrayList<>();
		list.add("Test1");
		list.add("Test2");
		list.add("Test3");
		list.add("Test4");
		list.add("Test5");
		System.out.println(list);
		System.out.println("");
		System.out.println("Iterate ArrayList Through For Loop");
		for (int i = 0; i < list.size(); i++) {
			System.out.println(list.get(i));
		}

		System.out.println("Iterate ArrayList Through For Each Loop");
		for (String li : list) {
			System.out.println(li);
		}

		System.out.println("Iterate ArrayList Through Iterator");
		Iterator<String> itr = list.iterator();
		while(itr.hasNext()){
			System.out.println(itr.next());
		}

	}
}
```

ArrayList addAll() method example

```
package arrayList;
import java.util.ArrayList;

public class Example4 {

	public static void main(String[] args) {
		ArrayList<String> list = new ArrayList<>();
		list.add("Test1");
		list.add("Test2");
		list.add("Test3");
		list.add("Test4");
		list.add("Test5");
		System.out.println(list);
```

```
		ArrayList<String> list1 = new ArrayList<>();
		list1.add("Test11");
		list1.add("Test21");

		list.addAll(list1);

		System.out.println(list);
	}
}
```

ArrayList clear() method example

```
package arrayList;

import java.util.ArrayList;

public class Example5 {

	public static void main(String[] args) {
		ArrayList<String> list = new ArrayList<>();
		list.add("Test1");
		list.add("Test2");
		list.add("Test3");
		list.add("Test4");
		list.add("Test5");

		list.clear();

		System.out.println(list);
	}
}
```

How to add Custom object to array List

```
package arrayList;

public class Student {

	private int age;
	private String name;
	private String school;

	public Student(int age, String name, String school) {
		super();
```

```
		this.age = age;
		this.name = name;
		this.school = school;
	}

	@Override
	public String toString() {
		return "Student [age=" + age + ", name=" + name + ", school=" + school + "]";
	}

}
package arrayList;

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

public class Example8 {

	public static void main(String[] args) {
		List<Student> list = new ArrayList<>();
		list.add(new Student(10, "Amit", "DS"));
		list.add(new Student(11, "Mohan", "AS"));
		list.add(new Student(12, "Ram", "GS"));
		list.add(new Student(13, "Suresh", "KS"));
		list.add(new Student(14, "Bhanu", "DK"));

		System.out.println(list);
	}

}
```

ArrayList clone()

ArrayList clone() method is used to create a **shallow copy** of the list. In the new list, only object references are copied.
If we change the object state inside first arraylist, then changed object state will be reflected in cloned arraylist as well.
Learn About Cloning
What is Java clone?

So cloning is about creating the copy of original object. Its dictionary meaning is : “make an identical copy of“.

By default, java cloning is 'field by field copy' i.e. as the Object class does not have idea about the structure of class on which clone() method will be invoked.

So, JVM when called for cloning, do following things:

If the class has only primitive data type members then a completely new copy of the object will be created and the reference to the new object copy will be returned.
If the class contains members of any class type then only the object references to those members are copied and hence the member references in both the original object as well as the cloned object refer to the same object.
Apart from above default behavior, you can always override this behavior and specify your own. This is done using overriding clone() method.

Java Cloneable interface and clone() method
Every language which supports cloning of objects has its own rules and so does java. In java, if a class needs to support cloning it has to do following things:

You must implement Cloneable interface.
You must override clone() method from Object class.

Java clone() method

```
/*
Creates and returns a copy of this object. The precise meaning of "copy" may depend on the class of the object.
The general intent is that, for any object x, the expression:
1) x.clone() != x will be true
2) x.clone().getClass() == x.getClass() will be true, but these are not absolute requirements.
3) x.clone().equals(x) will be true, this is not an absolute requirement.
*/

protected native Object clone() throws CloneNotSupportedException;
```

1. First statement **guarantees** that cloned object will have separate memory address assignment.
2. Second statement **suggest** that original and cloned objects should have same class type, but it is not mandatory.
3. Third statement **suggest** that original and cloned objects should have be equal using equals() method, but it is not mandatory.

Let's understand **Java clone with example**. Our first class is **Employee** class with 3 attributes – **id, name and department.**

Shallow clone is *"default implementation"* in Java. In overridden clone method, if you are not cloning all the object types (not primitives), then you are making a shallow copy.

```
package arrayList;
public class Employee implements Cloneable {
	private int empoyeeId;
	private String employeeName;
	private Department department;

	public Employee(int id, String name, Department dept) {
		this.empoyeeId = id;
		this.employeeName = name;
		this.department = dept;
	}

	@Override
	protected Object clone() throws CloneNotSupportedException {
		return super.clone();
	}

	public int getEmpoyeeId() {
		return empoyeeId;
	}

	public void setEmpoyeeId(int empoyeeId) {
		this.empoyeeId = empoyeeId;
	}

	public String getEmployeeName() {
		return employeeName;
	}

	public void setEmployeeName(String employeeName) {
		this.employeeName = employeeName;
	}

	public Department getDepartment() {
		return department;
	}

	public void setDepartment(Department department) {
		this.department = department;
	}
}
```

By Bhanu Pratap Singh
https://www.youtube.com/user/MrBhanupratap29/playlists
https://www.facebook.com/learnbybhanupratap/
https://www.udemy.com/seleniumbybhanu/

```
package arrayList;
public class Department {

	private int id;
	private String name;

	public Department(int id, String name) {
		this.id = id;
		this.name = name;
	}

	public int getId() {
		return id;
	}

	public void setId(int id) {
		this.id = id;
	}

	public String getName() {
		return name;
	}

	public void setName(String name) {
		this.name = name;
	}

}
```

```
package arrayList;
public class TestCloning {

	public static void main(String[] args) throws CloneNotSupportedException {

		Department dept = new Department(1, "Human Resource");
		Employee original = new Employee(1, "Admin", dept);

		// Lets create a clone of original object
		Employee cloned = (Employee) original.clone();

		// Let verify using employee id, if cloning actually workded
		System.out.println(cloned.getEmpoyeeId());

		// Verify JDK's rules
```

```
		// Must be true and objects must have different memory addresses
		System.out.println(original != cloned);

		// As we are returning same class; so it should be true
		System.out.println(original.getClass() == cloned.getClass());

		// Default equals method checks for references so it should be false. If
		// we want to make it true,
		// then we need to override equals method in Employee class.
		System.out.println(original.equals(cloned));
	}
}
```

```
package arrayList;
public class TestCloning1 {

	public static void main(String[] args) throws CloneNotSupportedException {

		Department hr = new Department(1, "Human Resource");
		Employee original = new Employee(1, "Admin", hr);
		Employee cloned = (Employee) original.clone();

		// Let change the department name in cloned object and we will verify in
		// original object
		cloned.getDepartment().setName("Finance");

		System.out.println(original.getDepartment().getName());
		System.out.println(cloned.getDepartment().getName());
	}
}
```

Oops, cloned object changes are visible in original also. This way cloned objects can make havoc in the system if allowed to do so. Anybody can come and clone your application objects and do whatever he likes. Can we prevent this??
Answer is yes,
Deep clone is the desired behavior in most the cases. In the deep copy, we create a clone which is independent of original object and making changes in the cloned object should not affect original object.

```
package arrayList;

public class Employee1 implements Cloneable {
	private int empoyeeId;
	private String employeeName;
```

```
	private Department1 department;

	public Employee1(int id, String name, Department1 dept) {
		this.empoyeeId = id;
		this.employeeName = name;
		this.department = dept;
	}

	// Modified clone() method in Employee class
	@Override
	protected Object clone() throws CloneNotSupportedException {
		Employee1 cloned = (Employee1) super.clone();
		cloned.setDepartment((Department1) cloned.getDepartment().clone());
		return cloned;
	}

	public int getEmpoyeeId() {
		return empoyeeId;
	}

	public void setEmpoyeeId(int empoyeeId) {
		this.empoyeeId = empoyeeId;
	}

	public String getEmployeeName() {
		return employeeName;
	}

	public void setEmployeeName(String employeeName) {
		this.employeeName = employeeName;
	}

	public Department1 getDepartment() {
		return department;
	}

	public void setDepartment(Department1 department) {
		this.department = department;
	}
}
package arrayList;

public class Department1 implements Cloneable {
```

```
	private int id;
	private String name;

	public Department1(int id, String name) {
		this.id = id;
		this.name = name;
	}

	public int getId() {
		return id;
	}

	public void setId(int id) {
		this.id = id;
	}

	public String getName() {
		return name;
	}

	public void setName(String name) {
		this.name = name;
	}

	@Override
	protected Object clone() throws CloneNotSupportedException {
	  return super.clone();
	}

}
package arrayList;
public class TestCloning2 {

	public static void main(String[] args) throws CloneNotSupportedException {

		Department1 hr = new Department1(1, "Human Resource");

		Employee1 original = new Employee1(1, "Admin", hr);
		Employee1 cloned = (Employee1) original.clone();

		// Let change the department name in cloned object and we will verify in
		// original object
		cloned.getDepartment().setName("Finance");
```

```
            System.out.println(original.getDepartment().getName());
            System.out.println(cloned.getDepartment().getName());
        }
}
```

Java ArrayList contains() – Check if element exists

```
package arrayList;
import java.util.ArrayList;
public class Example9 {

        public static void main(String[] args) {
            ArrayList<String> list = new ArrayList<>();
            list.add("Test1");
            list.add("Test2");
            list.add("Test3");
            list.add("Test4");
            list.add("Test5");

            list.contains("Test1");
        }

}
```

ArrayList indexOf() example to check if element exists

contains() method uses indexOf() method to determine if a specified element is present in the list or not. So we can also directly use the indexOf() method to check the existance of any supplied element value.

```
package arrayList;
import java.util.ArrayList;
public class Example10 {

        public static void main(String[] args) {
            {
                ArrayList<String> list = new ArrayList<>(2);
                list.add("A");
                list.add("B");
                list.add("C");
                list.add("D");
                System.out.println(list.indexOf("A") > 0); // true

                System.out.println(list.indexOf("Z") > 0); // false
```

```
		}
	}
}
```

Arraylist indexOf() – Get index of element in arraylist

This method returns the index of the first occurrence of the specified element in this list. It will return '-1'if the list does not contain the element.

```
package arrayList;
import java.util.ArrayList;
import java.util.Arrays;

public class Example11 {

	public static void main(String[] args) {

		ArrayList<String> list = new ArrayList<>(
				Arrays.asList("test1", "test2", "test3", "test4", "test5", "test6",
"test7", "test8"));

		int firstIndex = list.indexOf("test3");

		System.out.println(firstIndex);

		firstIndex = list.indexOf("hello");

		System.out.println(firstIndex);
	}
}
```

Arraylist lastIndexOf() – Get last index of element in arraylist in Java

This method returns the index of the last occurrence of the specified element in this list. It will return '-1' if the list does not contain the element.

```
package arrayList;

import java.util.ArrayList;
import java.util.Arrays;

public class Example12 {
```

```
	public static void main(String[] args) {
		ArrayList<String> list = new ArrayList<>(
				Arrays.asList("test7", "test2", "test3", "test4", "test5", "tset6",
"trset7", "test8"));

		int lastIndex = list.lastIndexOf("test7");

		System.out.println(lastIndex);

		lastIndex = list.lastIndexOf("hello");

		System.out.println(lastIndex);
	}

}
```

ArrayList removeAll() method example

```
package arrayList;

import java.util.ArrayList;

public class Example13 {

	public static void main(String[] args) {
		ArrayList<String> list = new ArrayList<>();
		list.add("Test1");
		list.add("Test2");
		list.add("Test3");
		list.add("Test4");
		list.add("Test5");

		System.out.println(list);

		list.removeAll(list);

		System.out.println(list);
	}
}
```

```
package arrayList;

import java.util.ArrayList;
```

```
public class Example14 {

	public static void main(String[] args) {
		ArrayList<String> list = new ArrayList<>();
		list.add("Test1");
		list.add("Test2");
		list.add("Test3");
		list.add("Test4");
		list.add("Test5");

		System.out.println(list);

		ArrayList<String> list1 = new ArrayList<>();
		list1.add("Test1");
		list1.add("Test2");
		list1.add("Test3");

		list.removeAll(list1);

		System.out.println(list);
	}

}
```

ArrayList retainAll() method example

ArrayList retainAll() retains only the elements in this list that are contained in the specified method argument collection. Rest all elements are removed from the list. This method is exact opposite to removeAll() method.

```
package arrayList;

import java.util.ArrayList;

public class Example15 {

	public static void main(String[] args) {
		ArrayList<String> list = new ArrayList<>();
		list.add("Test1");
		list.add("Test2");
		list.add("Test3");
		list.add("Test4");
		list.add("Test5");
```

```
		System.out.println(list);

		ArrayList<String> list1 = new ArrayList<>();
		list1.add("Test1");
		list1.add("Test2");
		list1.add("Test3");

		list.retainAll(list1);

		System.out.println(list);
	}

}
```

ArrayList replaceAll() method example

ArrayList replaceAll() retains only the elements in this list that are contained in the specified method argument collection. Rest all elements are removed from the list. This method is exact opposite to removeAll() method.

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

public class Example16 {

	public static void main(String[] args) {
		ArrayList<String> list = new ArrayList<>();
		list.add("Test1");
		list.add("Test2");
		list.add("Test3");
		list.add("Test4");
		list.add("Test1");

		System.out.println(list);

		Collections.replaceAll(list, "Test1","Test30");

		System.out.println(list);
	}
}
```

```java
package arrayList;
public class Example19 {

	public static void main(String args[]) {
		String Str = new String("Welcome to learnjava.com");

		System.out.print("Return Value :");
		System.out.println(Str.replaceAll("(.*)learnjava(.*)", "youtube"));
	}
}
```

```java
package arrayList;
public class Example20 {

	public static void main(String args[]) {
		String str = "String replaceAll() method example!!";
		String strRegExTest = "Java 12 23 String 4 Replace Example";
		String strObj = null;

		// Replace all occurrences of "t" to "T"
		strObj = str.replaceAll("t", "T");
		System.out.println(strObj);

		// Remove all occurrences of "!"
		strObj = str.replaceAll("!", "");
		System.out.println(strObj);

		// Replace "example" to "Example"
		strObj = str.replaceAll("example", "Example");
		System.out.println(strObj);

		// Remove all the numbers
		strObj = strRegExTest.replaceAll("[0-9]+", "");
		System.out.println(strObj);

		// Replace all the words to "Word"
		strObj = strRegExTest.replaceAll("[a-zA-Z]+", "Word");
		System.out.println(strObj);

	}
}
```

ArrayList sort()

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

public class Example17 {

	public static void main(String[] args) {
		ArrayList<Integer> list = new ArrayList<>();
		list.add(10);
		list.add(9);
		list.add(20);
		list.add(11);
		list.add(15);

		System.out.println("before sorting="+list);

		Collections.sort(list);

		System.out.println("After sorting="+list);
	}

}
```

```java
package arrayList;

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

public class Example18 {

	public static void main(String[] args) {
		ArrayList<String> list = new ArrayList<>();
		list.add("Shyam");
		list.add("Bdanu");
		list.add("Ram");
		list.add("Amit");
		list.add("Baoy");

		System.out.println("before sorting=" + list);

		Collections.sort(list);

		System.out.println("After sorting=" + list);
	}
}
```

By Bhanu Pratap Singh
https://www.youtube.com/user/MrBhanupratap29/playlists
https://www.facebook.com/learnbybhanupratap/
https://www.udemy.com/seleniumbybhanu/

ArrayList sort() – Sort list of objects by field

Comparators are most useful when we want to sort a given list of objects – but **not in natural order**.
Java Comparator interface is used to order the objects of a user-defined class.This interface is found in java.util package and contains 2 methods compare(Object obj1,Object obj2) and equals(Object element).
It provides multiple sorting sequences.

```
package arrayList;

public class State {

	private int population;
	private String district;
	private String language;

	public int getPopulation() {
		return population;
	}

	public void setPopulation(int population) {
		this.population = population;
	}

	public String getDistrict() {
		return district;
	}

	public void setDistrict(String district) {
		this.district = district;
	}

	public String getLanguage() {
		return language;
	}

	public void setLanguage(String language) {
		this.language = language;
	}

	public State(int population, String district, String language) {
		super();
		this.population = population;
```

```
		this.district = district;
		this.language = language;
	}

	@Override
	public String toString() {
		return "State [population=" + population + ", district=" + district + ",
language=" + language + "]";
	}
}
```

```
package arrayList;

import java.util.Comparator;

public class LanguageSorter implements Comparator<State> {

	@Override
	public int compare(State o1, State o2) {
		return o1.getLanguage().compareToIgnoreCase(o2.getLanguage());
	}
}
```

```
package arrayList;
import java.util.Comparator;
public class PopulationSorter implements Comparator<State> {

	@Override
	public int compare(State o1, State o2) {
		if (o1.getPopulation() == o2.getPopulation())
			return 0;
		else if (o1.getPopulation() > o2.getPopulation())
			return 1;
		else
			return -1;
	}
}
```

```
package arrayList;

import java.util.Comparator;
public class DistrictSorter implements Comparator<State>{

	@Override
```

```
	public int compare(State o1, State o2) {
		return o1.getDistrict().compareTo(o2.getDistrict());
	}

}
```

```
package arrayList;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class TestObjectSorting {

	public static void main(String[] args) {
		List<State> list = new ArrayList<State>();
		list.add(new State(100, "AB", "BB"));
		list.add(new State(500, "DC", "CC"));
		list.add(new State(300, "BC", "AA"));
		list.add(new State(800, "FC", "FF"));
		list.add(new State(600, "ED", "EE"));

		list.sort(new PopulationSorter());
		System.out.println(list);

		//Collections.sort(list,new LanguageSorter());
		//System.out.println(list);

		list.sort(new LanguageSorter());
		System.out.println(list);

		Collections.sort(list,new DistrictSorter());
		System.out.println(list);

	}

}
```

ArrayList.subList() method

subList() Method Parameters
fromIndex – start index in existing arraylist. It is inclusive.
toIndex – last index in existing arraylist. It is exclusive.

```
package arrayList;

import java.util.ArrayList;
import java.util.Arrays;

public class Example21 {

	public static void main(String[] args) {
		ArrayList<Integer> list = new ArrayList<>(Arrays.asList(0, 10, 20, 30, 40, 50,
60, 70, 80, 90));

		ArrayList<Integer> sublist = new ArrayList<Integer>(list.subList(2, 6));

		System.out.println(sublist);
	}

}
```

Java ArrayList toArray()

```
package arrayList;
import java.util.ArrayList;
import java.util.Arrays;

public class Example22 {

	public static void main(String[] args) {

		ArrayList<String> list = new ArrayList<>(2);

		list.add("A");
		list.add("B");
		list.add("C");
		list.add("D");

		// Convert to object array
		Object[] array = list.toArray();

		System.out.println(Arrays.toString(array));

		// Iterate and convert to desired type
		for (Object o : array) {
			String s = (String) o;
```

```
                System.out.println(s);
            }
        }
}
```

ArrayList addAll() method

ArrayList addAll() method is used to append all of the elements of argument collection to the list at the end.

```
package arrayList;
import java.util.ArrayList;

public class Example23 {

    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>(2);

        list.add("A");
        list.add("B");
        list.add("C");
        list.add("D");

        ArrayList<String> list1 = new ArrayList<>(2);

        list1.add("E");
        list1.add("F");
        list1.add("G");
        list1.add("H");

        list.addAll(list1);

        System.out.println(list);
    }
}
```

ArrayList clear()

ArrayList clear() method is used to removes all of the elements from the list. The list will be empty after this call returns.

```java
package arrayList;
import java.util.ArrayList;
public class Example23 {

	public static void main(String[] args) {
		ArrayList<String> list = new ArrayList<>(2);

		list.add("A");
		list.add("B");
		list.add("C");
		list.add("D");

		ArrayList<String> list1 = new ArrayList<>(2);

		list1.add("E");
		list1.add("F");
		list1.add("G");
		list1.add("H");

		list.addAll(list1);

		System.out.println(list);

		list.clear();
		System.out.println(list);
	}
}
```

Java LinkedList class

Java LinkedList class is doubly-linked list implementation of the List and Deque interfaces. It implements all optional list operations, and permits all elements (including null).

LinkedList Features

- Doubly linked list implementation which implements List and Deque interfaces. Therefore, It can also be used as a Queue, Deque or Stack.
- Permits all elements including duplicates and NULL.
- LinkedList maintains the insertion order of the elements.
- It is not synchronized. If multiple threads access a linked list concurrently, and at least one of the threads modifies the list structurally, it must be synchronized externally.
- Use Collections.synchronizedList(new LinkedList()) to get synchronized linkedlist.

By Bhanu Pratap Singh
https://www.youtube.com/user/MrBhanupratap29/playlists
https://www.facebook.com/learnbybhanupratap/
https://www.udemy.com/seleniumbybhanu/

- The iterators returned by this class are fail-fast and may throw ConcurrentModificationException.
- It does not implement RandomAccess interface. So we can access elements in sequential order only. It does not support accessing elements randomly.
- We can use ListIterator to iterate LinkedList elements.

LinkedList Methods

- boolean add(Object o) : appends the specified element to the end of a list.
- void add(int index, Object element) : inserts the specified element at the specified position index in a list.
- void addFirst(Object o) : inserts the given element at the beginning of a list.
- void addLast(Object o) : appends the given element to the end of a list.
- int size() : returns the number of elements in a list
- boolean contains(Object o) : return true if the list contains a specified element, else false.
- boolean remove(Object o) : removes the first occurence of the specified element in a list.
- Object getFirst() : returns the first element in a list.
- Object getLast() : returns the last element in a list.
- int indexOf(Object o) : returns the index in a list of the first occurrence of the specified element, or -1 if the list does not contain specified element.
- lastIndexOf(Object o) : returns the index in a list of the last occurrence of the specified element, or -1 if the list does not contain specified element.
- Iterator iterator() : returns an iterator over the elements in this list in proper sequence.
- Object[] toArray() : returns an array containing all of the elements in this list in proper sequence.
- List subList(int fromIndex, int toIndex) : returns a view of the portion of this list between the specified fromIndex (inclusive) and toIndex (exclusive).

Linked List Add Methods

```
package linkedList;
import java.util.LinkedList;
import java.util.ListIterator;

public class Example1 {

	public static void main(String[] args) {
		{
```

```
                LinkedList<String> linkedList = new LinkedList<>();

                linkedList.add("A");
                linkedList.add("B");
                linkedList.add("C");
                linkedList.add("D");

                System.out.println(linkedList);

                linkedList.add(4, "4A");
                linkedList.add(5, "5A");

                linkedList.addFirst("TT");
                linkedList.addLast("LL");

                System.out.println(linkedList);

                ListIterator<String> itrator = linkedList.listIterator();
                while (itrator.hasNext()) {
                        System.out.println(itrator.next());
                }
        }
    }
}

package linkedList;
import java.util.LinkedList;
public class Example3 {

    public static void main(String[] args) {

        LinkedList<String> linkedList = new LinkedList<>();

        linkedList.add("A");
        linkedList.add("B");
        linkedList.add("C");
        linkedList.add("D");

        LinkedList<String> linkedList1 = new LinkedList<>();

        linkedList1.add("AA");
        linkedList1.add("BB");
        linkedList1.add("CC");
        linkedList1.add("DD");
```

```
                linkedList.addAll(linkedList1);

                System.out.println(linkedList);

                linkedList.addAll(1, linkedList1);

                System.out.println(linkedList);

        }
}
```

Linked List Get Methods

```
package linkedList;
import java.util.LinkedList;
public class Example2 {

        public static void main(String[] args) {
                {
                        LinkedList<String> linkedList = new LinkedList<>();

                        linkedList.add("A");
                        linkedList.add("B");
                        linkedList.add("C");
                        linkedList.add("D");

                        System.out.println(linkedList.get(0));
                        System.out.println(linkedList.getFirst());
                        System.out.println(linkedList.getLast());
                }
        }
}
```

ArrayList vs LinkedList

- ArrayList is implemented with the concept of dynamic resizable array. While LinkedList is a doubly linked list implementation.
- ArrayList allows random access to it's elements while LinkedList does not.
- LinkedList, also implements Queue interface which adds more methods than ArrayList, such as offer(), peek(), poll(), etc.
- While comparing to LinkedList, ArrayList is slower in add and remove, but faster in get, because there is no need of resizing array and copying content to new array if array gets full in LinkedList.

By Bhanu Pratap Singh
https://www.youtube.com/user/MrBhanupratap29/playlists
https://www.facebook.com/learnbybhanupratap/
https://www.udemy.com/seleniumbybhanu/

- LinkedList has more memory overhead than ArrayList because in ArrayList each index only holds actual object but in case of LinkedList each node holds both data and address of next and previous node.

Java HashSet

The important points about Java HashSet class are:

- HashSet stores the elements by using a mechanism called **hashing.**
- HashSet contains unique elements only.
- HashSet allows null value.
- HashSet class is non synchronized.
- HashSet doesn't maintain the insertion order. Here, elements are inserted on the basis of their hashcode.
- HashSet is the best approach for search operations.
- The initial default capacity of HashSet is 16, and the load factor is 0.75.

Constructor	Description
HashSet()	It is used to construct a default HashSet.
HashSet(int capacity)	It is used to initialize the capacity of the hash set to the given integer value capacity. The capacity grows automatically as elements are added to the HashSet.
HashSet(int capacity, float loadFactor)	It is used to initialize the capacity of the hash set to the given integer value capacity and the specified load factor.
HashSet(Collection<? extends E> c)	It is used to initialize the hash set by using the elements of the collection c.

Return Type	Method	Description
boolean	add(E e)	It is used to add the specified element to this set if it is not already present.
void	clear()	It is used to remove all of the elements from the set.
object	clone()	It is used to return a shallow copy of this HashSet instance: the elements themselves are not cloned.
boolean	contains(Object o)	It is used to return true if this set contains the specified element.
boolean	isEmpty()	It is used to return true if this set contains no elements.
Iterator<E>	iterator()	It is used to return an iterator over the elements in this set.
boolean	remove(Object o)	It is used to remove the specified element from this set if it is present.
int	size()	It is used to return the number of elements in the set.

Spliterator<E>	spliterator()	It is used to create a late-binding and fail-fast Spliterator over the elements in the set.

```
package setClass;
import java.util.HashSet;
import java.util.Iterator;

public class Example1 {

	public static void main(String[] args) {
		HashSet<String> set = new HashSet<String>();

		set.add("Test1");
		set.add("Test2");
		set.add("Test3");
		set.add("Test4");
		set.add("Test5");
		Iterator<String> i = set.iterator();
		while (i.hasNext()) {
			System.out.println(i.next());
		}
	}
}
```

Java HashSet example ignoring duplicate elements

```
package setClass;
import java.util.HashSet;
import java.util.Iterator;
public class Example2 {

	public static void main(String args[]) {
		HashSet<String> set = new HashSet<String>();
		set.add("Test1");
		set.add("Test2");
		set.add("Test3");
		set.add("Test2");
		Iterator<String> itr = set.iterator();
		while (itr.hasNext()) {
			System.out.println(itr.next());
		}
	}
}
```

HashSet example to remove elements

```
package setClass;
import java.util.HashSet;
public class Example3 {

	public static void main(String[] args) {
		HashSet<String> set = new HashSet<String>();
		set.add("Test1");
		set.add("Test2");
		set.add("Test3");
		set.add("Test4");
		System.out.println("An initial list of elements: " + set);
		set.remove("Test1");
		System.out.println("After invoking remove(object) method: " + set);
		HashSet<String> set1 = new HashSet<String>();
		set1.add("Test2");
		set1.add("Test3");
		set.addAll(set1);
		System.out.println("Updated List: " + set);
		set.removeAll(set1);
		System.out.println("After invoking removeAll() method: " + set);
		set.clear();
		System.out.println("After invoking clear() method: " + set);
	}
}
```

Storing custom Object in Hash map

```
package setClass;
public class Book {

	int id;
	String name, author, publisher;
	int quantity;

	public Book(int id, String name, String author, String publisher, int quantity) {
		this.id = id;
		this.name = name;
		this.author = author;
		this.publisher = publisher;
		this.quantity = quantity;
	}
}
```

By Bhanu Pratap Singh
https://www.youtube.com/user/MrBhanupratap29/playlists
https://www.facebook.com/learnbybhanupratap/
https://www.udemy.com/seleniumbybhanu/

```
package setClass;
import java.util.HashSet;
public class Example4 {

	public static void main(String[] args) {

		HashSet<Book> set = new HashSet<Book>();
		Book b1 = new Book(11, "Test1", "author1", "publisher1", 8);
		Book b2 = new Book(12, "Test2", "author2", "publisher2", 4);
		Book b3 = new Book(13, "Test3", "author3", "publisher3", 6);
		set.add(b1);
		set.add(b2);
		set.add(b3);
		for (Book book : set) {
			System.out.println(
					book.id + " " + book.name + " " + book.author + " " +
book.publisher + " " + book.quantity);
		}
	}
}
```

Java LinkedHashSet class

The important points about Java LinkedHashSet class are:

- Java LinkedHashSet class contains unique elements only like HashSet.
- Java LinkedHashSet class provides all optional set operation and permits null elements.
- Java LinkedHashSet class is non synchronized.
- Java LinkedHashSet class maintains insertion order.

```
package setClass;
import java.util.HashSet;
import java.util.Iterator;
import java.util.LinkedHashSet;

public class Example5 {
	public static void main(String args[]) {

		HashSet<String> set = new HashSet<String>();
		set.add("Test1");
		set.add("Test40");
		set.add("Test3");
		set.add("Test2");
```

```
		set.add("Test20");
		set.add("Test2");
		Iterator<String> itr = set.iterator();
		while (itr.hasNext()) {
			System.out.println(itr.next());
		}
		System.out.println("=========");
		LinkedHashSet<String> linkedset = new LinkedHashSet<String>();
		linkedset.add("Test1");
		linkedset.add("Test40");
		linkedset.add("Test3");
		linkedset.add("Test2");
		linkedset.add("Test20");
		linkedset.add("Test2");
		Iterator<String> itr1 = linkedset.iterator();
		while (itr1.hasNext()) {
			System.out.println(itr1.next());
		}
	}
}
```

```
Test1
Test40
Test20
Test3
Test2
=========
Test1
Test40
Test3
Test2
Test20
```

TreeSet

The important points about Java TreeSet class are:

- Java TreeSet class contains unique elements only like HashSet.
- Java TreeSet class access and retrieval times are quiet fast.
- Java TreeSet class doesn't allow null element.
- Java TreeSet class is non synchronized.
- Java TreeSet class maintains ascending order.

```
package setClass;
```

```java
import java.util.Iterator;
import java.util.LinkedHashSet;
import java.util.TreeSet;

public class Example6 {

	public static void main(String[] args) {
		TreeSet<Integer> set = new TreeSet<Integer>();
		set.add(10);
		set.add(40);
		set.add(900);
		set.add(50);
		set.add(6);
		set.add(70);
		Iterator<Integer> itr = set.iterator();
		while (itr.hasNext()) {
			System.out.println(itr.next());
		}
		System.out.println("============");
		TreeSet<String> set1 = new TreeSet<String>();
		set1.add("ABC");
		set1.add("BCD");
		set1.add("EFG");
		set1.add("XXX");
		set1.add("CAB");
		Iterator<String> itr1 = set1.iterator();
		while (itr1.hasNext()) {
			System.out.println(itr1.next());
		}
		System.out.println("============");
		LinkedHashSet<String> set2 = new LinkedHashSet<String>();
		set2.add("ABC");
		set2.add("BCD");
		set2.add("EFG");
		set2.add("XXX");
		set2.add("CAB");
		Iterator<String> itr2 = set2.iterator();
		while (itr2.hasNext()) {
			System.out.println(itr2.next());
		}
	}
}
```

Hash Map Class

By Bhanu Pratap Singh
https://www.youtube.com/user/MrBhanupratap29/playlists
https://www.facebook.com/learnbybhanupratap/
https://www.udemy.com/seleniumbybhanu/

Points to remember

- Java HashMap class contains values based on the key.
- Java HashMap class contains only unique keys.
- Java HashMap class may have one null key and multiple null values.
- Java HashMap class is non synchronized.
- Java HashMap class maintains no order.

```
package mapClass;
import java.util.HashMap;
public class Example1 {

	public static void main(String[] args) {
		HashMap<String, String> map = new HashMap<>();
		map.put("Test1", "A");
		map.put("Test2", "B");
		map.put("Test3", "C");
		map.put("Test4", "D");
		System.out.println(map);
	}

}
```

Constructor In HashMap

Constructor	Description
HashMap()	It is used to construct a default HashMap.
HashMap(Map<? extends K,? extends V> m)	It is used to initialize the hash map by using the elements of the given Map object m.
HashMap(int capacity)	It is used to initializes the capacity of the hash map to the given integer value, capacity.
HashMap(int capacity, float loadFactor)	It is used to initialize both the capacity and load factor of the hash map by using its arguments.

Iterator Over HashMap

```
package mapClass;

import java.util.HashMap;
import java.util.Map;

public class Example1 {

	public static void main(String[] args) {
		HashMap<String, String> map = new HashMap<>();
		map.put("Test1", "A");
```

```
		map.put("Test2", "B");
		map.put("Test3", "C");
		map.put("Test4", "D");
		System.out.println(map);

		for (Map.Entry<String, String> m : map.entrySet()) {
			System.out.println(m.getKey() + " " + m.getValue());
		}
	}

}
```

Get Data from HashMap

```
package mapClass;
import java.util.HashMap;
public class Example3 {

	public static void main(String[] args) {

		HashMap<String, String> map = new HashMap<>();
		map.put("Test1", "A");
		map.put("Test2", "B");
		map.put("Test3", "C");
		map.put("Test4", "D");
		System.out.println(map);

		System.out.println(map.get("Test1"));
	}
}
```

Remove and putAll method of HashMap

```
package mapClass;
import java.util.HashMap;
public class Example4 {

	public static void main(String[] args) {

		HashMap<String, String> map = new HashMap<>();
		map.put("Test1", "A");
		map.put("Test2", "B");
		map.put("Test3", "C");
		map.put("Test4", "D");
		System.out.println(map);
		System.out.println("=========");
```

```
		System.out.println(map.remove("Test1"));

		System.out.println(map);
		System.out.println("=========");
		HashMap<String, String> map1 = new HashMap<>();
		map1.put("Test5", "A1");
		map1.put("Test6", "B1");
		map1.put("Test7", "C1");
		map1.put("Test8", "D1");

		map.putAll(map1);
		System.out.println(map);
	}
}
```

Hash Map Methods

```
package mapClass;
import java.util.HashMap;
public class Example5 {

	public static void main(String[] args) {
		HashMap<String, String> map = new HashMap<>();
		map.put("Test1", "A");
		map.put("Test2", "B");
		map.put("Test3", "C");
		map.put("Test4", "D");

		/**
		 * Removes all of the mappings from this map. The map will be empty
		 * after this call returns.
		 */
		map.clear();

		/**
		 * Returns <tt>true</tt> if this map contains a mapping for the
		 * specified key.
		 */
		System.out.println(map.containsKey("Test1"));

	  /**
	   * value whose presence in this map is to be tested
	   * @return <tt>true</tt> if this map maps one or more keys to the
	   *         specified value
	   */
```

```
		System.out.println(map.containsValue("A"));

		map.put("Test1", "A");
		map.put("Test2", "B");
		map.put("Test3", "C");
		map.put("Test4", "D");

		System.out.println(map.containsKey("Test1"));
		System.out.println(map.containsValue("A"));
	}
}
```

Hash Map Methods

```
package mapClass;
import java.util.HashMap;
import java.util.Map.Entry;
import java.util.Set;

public class Example6 {

	public static void main(String[] args) {
		HashMap<String, String> map = new HashMap<>();
		map.put("Test1", "A");
		map.put("Test2", "B");
		map.put("Test3", "C");
		map.put("Test4", "D");

		/**
		 * @return a set view of the mappings contained in this map.
		 */
		Set<Entry<String, String>> entry = map.entrySet();

		for (Entry<String, String> entry2 : entry) {
			System.out.println(entry2.getKey() + "==" + entry2.getValue());
		}

		/**
		 * Returns <tt>true</tt> if this map contains no key-value mappings.
		 *
		 * @return <tt>true</tt> if this map contains no key-value mappings
		 */
		System.out.println(map.isEmpty());

		/**
```

```
		 * Removes the mapping for the specified key from this map if present.
		 */
		map.remove("Test1");

		map.replace("Test2", "Test1000");

		System.out.println(map);
	}
}
```

Map which will take all type of data

```
package mapClass;
import java.util.HashMap;
import java.util.Map;

public class Example7 {

	public static void main(String[] args) {

		Map<Object, Object> map = new HashMap<>();
		map.put(10, 1000);
		map.put("Test1", "AAA");
		map.put(10.90, 10.999);
		map.put('A', 'n');

		System.out.println(map);

	}
}
```

Map to store object type data

```
package mapClass;
import java.util.HashMap;
import java.util.Map;
public class Example8 {

	public static void main(String[] args) {

		Map<Integer, School> map = new HashMap<>();

		School s1 = new School("Test1", "20", "A");
		School s2 = new School("Test2", "30", "B");
		School s3 = new School("Test3", "40", "C");
		School s4 = new School("Test4", "50", "D");
```

```
		map.put(1, s1);
		map.put(2, s2);
		map.put(3, s3);

		// What happens when you store duplicate key
		map.put(3, s4);

		System.out.println(map);

	}
}
```

Is it possible to Store Multiple null Key and values

```
package mapClass;
import java.util.HashMap;
import java.util.Map;
public class Example9 {

	public static void main(String[] args) {

		Map<Integer, String> map = new HashMap<>();

		map.put(null, "Test1");
		map.put(null, "Test1");
		map.put(null, "Test1");
		map.put(null, "Test1");
		map.put(1, null);
		map.put(2, null);
		map.put(3, null);
		map.put(4, null);

		System.out.println(map);
	}
}
```

Linked Hash Map and Tree Map

LinkedHashMap is a subclass of HashMap. That means it inherits the features of HashMap. In addition, the linked list preserves the insertion-order.

Map Overview

There are 4 commonly used implementations of Map in Java SE - HashMap, TreeMap, Hashtable and LinkedHashMap. If we use one sentence to describe each implementation, it would be the following:

- HashMap is implemented as a hash table, and there is no ordering on keys or values.
- TreeMap is implemented based on red-black tree structure, and it is ordered by the key.
- LinkedHashMap preserves the insertion order
- Hashtable is synchronized, in contrast to HashMap.

```
package mapClass;

import java.util.LinkedHashMap;
import java.util.TreeMap;

public class Example10 {

	public static void main(String[] args) {

		LinkedHashMap<Integer, String> linkedHashMap = new LinkedHashMap<>();
		linkedHashMap.put(10, "Test1");
		linkedHashMap.put(11, "Test2");
		linkedHashMap.put(1000, "Test1");
		linkedHashMap.put(40, "Test4");
		linkedHashMap.put(20, "Test5");

		System.out.println(linkedHashMap);

		TreeMap<Integer, String> treeMap = new TreeMap<>();

		treeMap.put(10, "Test1");
		treeMap.put(11, "Test2");
		treeMap.put(1000, "Test1");
		treeMap.put(40, "Test4");
		treeMap.put(20, "Test5");

		System.out.println(treeMap);
	}
}
```