Computer Programming

- Way of giving computers instructions about what they should do next.

  These ***instructions are known as code***, and computer programmers write code to solve problems or perform a task.

- End goal is to create something:

  Anything from a web page or a piece of software or even just a pretty picture.

- A computer program is a list of instructions that enable a computer to perform a specific task.
- Computer programs can be written in high and low level languages, depending on the task and the hardware being used.

Structured Programming:

Structured programming is a programming paradigm that focuses on organizing code using well-defined control flow structures like sequences, conditionals (if-else), and loops. It aims to enhance code clarity and maintainability by reducing complexity and making it easier to understand. In structured programming, code is typically organized in a linear, sequential manner with the use of functions or procedures for modularity. Key principles include the avoidance of "goto" statements and promoting structured control flow.

Procedural Programming:

Procedural programming is a subset of structured programming that places a strong emphasis on breaking code into procedures or functions. It promotes code modularity, top-down design, and better code reusability. In procedural programming, code is organized around procedures or functions that encapsulate specific tasks. Variables are typically local to these procedures, minimizing unintended side effects. Procedural programming languages like C and Pascal are known for their use of procedures and functions to structure code.

Object-Oriented Programming (OOP):

Object-oriented programming is a programming paradigm that models software as a collection of objects, each representing a real-world entity with data (attributes) and behavior (methods/functions). OOP promotes code organization around these objects, enabling code reusability through inheritance and creating a hierarchy of related objects. Key principles of OOP include encapsulation (hiding internal details), inheritance (sharing attributes and methods among objects), and polymorphism (objects responding differently to the same method call based on their class). Common OOP languages include Java, C++, and Python.

In summary, structured programming focuses on control flow structures, procedural programming adds the concept of procedures/functions for modularity, and object-oriented programming models software as a collection of objects with data and behavior, promoting code organization around real-world entities and fostering code reusability. Each paradigm has its own advantages and is suitable for different types of applications and problem-solving approaches.

Objects and Classes in Java

- An object in Java is the physical as well as a logical entity, whereas, a class in Java is a logical entity only.
- An entity that has state and behaviour is known as an object e.g., chair, bike, marker, pen, table, car, etc. An object has three characteristics:
- **State: represents** the data (value) of an object.
- **Behaviour:** represents the behaviour (functionality) of an object such as deposit, withdraw, etc.
- **Identity:** An object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. However, it is used internally by the JVM to identify each object uniquely.

- **An object is an instance of a class.** A class is a template or blueprint from which objects are created. So, an object is the instance(result) of a class.
- **Object Definitions:**
- An object is *a real-world entity*.
- An object is *a runtime entity*.
- The object is *an entity which has state and behavior*.
- The object is *an instance of a class*.

Class

- A class is a group of objects which have common properties. It is a template or blueprint from which objects are created. It is a logical entity. It can't be physical.
- A class in Java can contain:
- **Fields**
- **Methods**
- **Constructors**
- **Blocks**
- **Nested class and interface**

Inheritance

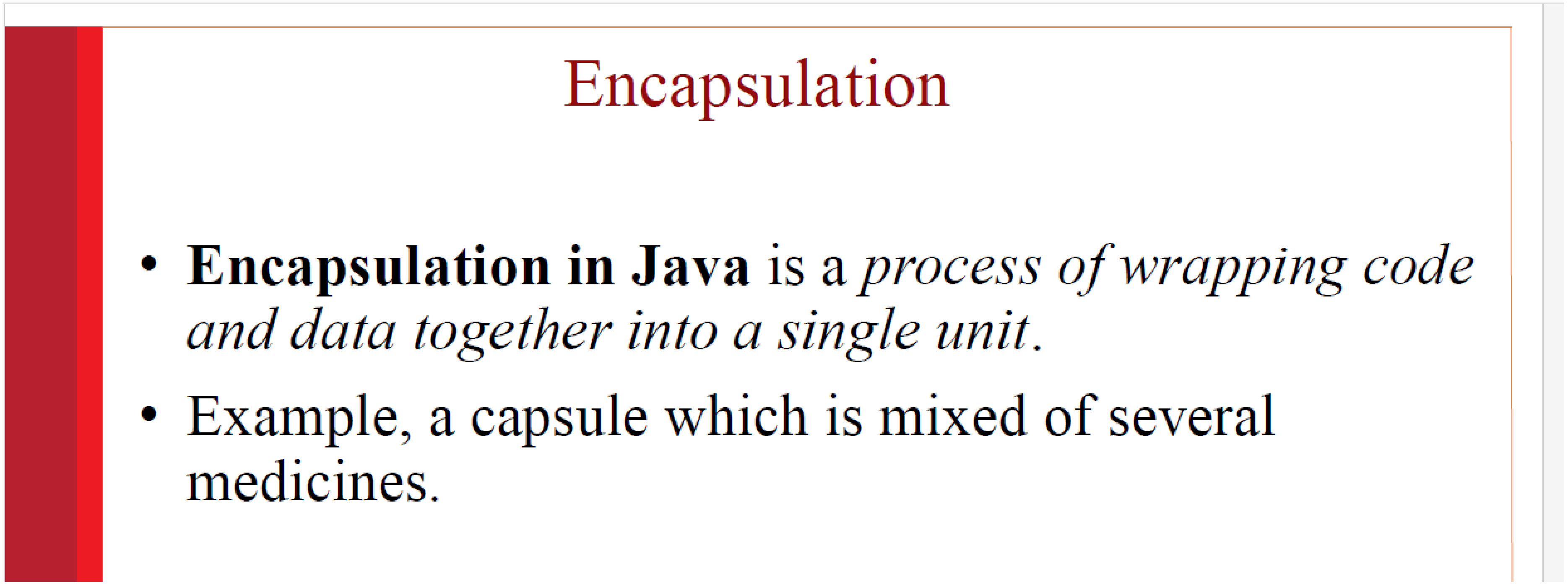
- **Inheritance in Java** is a mechanism in which one object acquires all the properties and behaviours of a parent object. It is an important part of OOPs (Object Oriented programming system).
- The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.
- Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship.

Polymorphism

- **Polymorphism in Java** is a concept by which we can perform a *single action in different ways*. Polymorphism is derived from 2 Greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.
- There are two types of polymorphism in Java: compile-time polymorphism and runtime polymorphism.
- We can perform polymorphism in java by method overloading and method overriding.

Abstraction

- **Abstraction** is a process of hiding the implementation details and showing only functionality to the user.
- Another way, it shows only essential things to the user and hides the internal details, for **Example**:sending SMS where you type the text and send the message. You don't know the internal processing about the message delivery.



Encapsulation

- **Encapsulation in Java** is a *process of wrapping code and data together into a single unit.*
- Example, a capsule which is mixed of several medicines.

Platform Independent

•Java is platform independent because it is different from other languages like C, C++, etc. which are compiled into platform specific machines

•while Java is **a write once, run anywhere language**. A platform is the hardware or software environment in which a program runs.

Secured

•Java is best known for its security. With Java, we can develop virus-free systems. Java is secured because:

•**No explicit pointer**

•**Java Programs run inside a virtual machine**

Robust

•Robust simply means strong. Java is robust because:

•It uses strong memory management.

•There is a lack of pointers that avoids security problems.

•There are exception handling and the type checking mechanism in Java. All these points make Java robust.

Architecture-neutral

•Java is architecture neutral because there are no implementation dependent features, for example, the size of primitive types is fixed.

Portable

•Java is portable because it facilitates you to carry the Java bytecode to any platform. It doesn't require any implementation

High-performance

•Java is faster than other traditional interpreted programming languages because Java bytecode is "close" to native code. It is still a little bit slower than a compiled language (e.g., C++).

•Java is an interpreted language that is why it is slower than compiled languages, e.g., C, C++,

Distributed

•Java is distributed because it facilitates users to create distributed applications in Java.

Multi-threaded

•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 multi-media, Web applications, etc.

Dynamic

•Java is a dynamic language. It supports dynamic loading of classes. It means classes are loaded on demand. It also supports functions from its native languages, i.e., C and C++.

•Java supports dynamic compilation and automatic memory management (garbage collection)

Constructors

- A constructor **initializes an object** when it is created. It has the same name as its class and is syntactically similar to a method. However, constructors have no explicit return type.
- We use a constructor to give initial values to the instance variables defined by the class, or to perform any other startup procedures required to create a fully formed object.
- All classes have constructors, whether you define one or not, because Java automatically provides a default constructor that initializes all member variables to zero. However, once you define your own constructor, the default constructor is no longer used.

Types of Constructors

- There are two types of constructors:
  - **Default constructor** (no-arg constructor): Default constructor provides the default values to the object like 0, null etc. It will be invoked at the time of object creation.depending on the type. <class_name>(){}
  - **Parameterized constructor**: A constructor that have parameters. Parameterized constructor is used to provide different values to the distinct objects.
- There is no copy constructor in java. But, we can copy the values of one object to another like copy constructor in C++.
  - By constructor
  - By assigning the values of one object into another
  - By clone() method of Object class

Constructor overloading

- Constructor overloading in Java is a technique of having more than one constructor with different parameter lists.
- They are arranged in a way that each constructor performs a different task.
- They are differentiated by the compiler by the number of parameters in the list and their types.

Method

- A Java method is a collection of statements that are grouped together to perform an operation. Syntax is

modifier returnType nameOfMethod (Parameter List) { // method body }

- Modifier is optional. There are two ways in which a method is called i.e. method returns a value or returning nothing

Si no	Constructor	Method
1	Constructor is used to initialize the **state** of an object.	Method is used to expose **behavior** of an object.
2	Constructor must not have return type	Method must have return type.
3	Constructor is invoked implicitly.	Method is invoked explicitly.
4	The java compiler provides a default constructor if you don't have any constructor.	Method is not provided by compiler in any case.
5	Constructor name must be same as the class name	Method name may or may not be same as class name.

This keyword

- **this** is a keyword in Java which is used as a reference to the object of the current class, with in an instance method or a constructor. Using *this* you can refer the members of a class such as constructors, variables and methods.

- usage of java this keyword.
  1. this keyword can be used to refer current class instance variable.
  2. this() can be used to invoke current class constructor.
  3. this keyword can be used to invoke current class method (implicitly)
  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 keyword can also be used to return the current class instance.

Destructor

- A **destructor** is a special member function of a class that is executed whenever an object of it's class goes out of scope or whenever the delete expression is applied to a pointer to the object of that class.
- A destructor will have exact same name as the class prefixed with a tilde (~) and it can neither return a value nor can it take any parameters. Destructor can be very useful for releasing resources before coming out of the program like closing files, releasing memories etc.

Finalize()

- The **finalize()** method is equivalent to a **destructor** of C++. When the job of an object is over, or to say, the object is no more used in the program, the object is known as **garbage**. The process of removing the object from a running program is known as **garbage collection**. finalize() method can be best utilized by the programmer to close the I/O streams, JDBC connections or socket handles etc.

Access Specifiers in Java

- **Public:** A **class, method, constructor, interface** etc declared public can be accessed from any other class. Therefore fields, methods, blocks declared inside a public class can be accessed from any class belonging to the Java Universe. However if the public class we are trying to access is in a different package, then the public class still need to be imported. Because of class inheritance, all public methods and variables of a class are inherited by its subclasses.
- **Protected:** **Variables, methods and constructors** which are declared protected in a superclass can be accessed only by the subclasses in other package or any class within the package of the protected members' class. The protected access modifier **cannot be applied to class and interfaces**. Methods, fields can be declared protected, however methods and fields in a interface cannot be declared protected. Protected access gives the subclass a chance to use the helper method or variable, while preventing a nonrelated class from trying to use it.
- **Private:** **Methods, Variables and Constructors** that are declared private can only be accessed within the declared class itself. Private access modifier is the most restrictive access level. **Class and interfaces cannot be private**. Variables that are declared private can be accessed outside the class if public getter methods are present in the class. Using the private modifier is the main way that an object encapsulates itself and hide data from the outside world.
- **Default:** Default access modifier means we do not explicitly declare an access modifier for a **class, field, method,** etc. A variable or method declared without any access control modifier is available to any other class in the same package. The fields in an interface are implicitly public static final and the methods in an interface are by default public.

List of Java Object class methods.

- The **Object class** is the parent class of all the classes in java bydefault. In other words, it is the topmost class of java.
- The Object class is beneficial if you want to refer any object whose type you don't know.

clone() - Creates and returns a copy of this object.

equals() - Indicates whether some other object is "equal to" this one.

finalize() - Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.

getClass() - Returns the runtime class of an object.

hashCode() - Returns a hash code value for the object.

notify() - Wakes up a single thread that is waiting on this object's monitor.

notifyAll() - Wakes up all threads that are waiting on this object's monitor.

toString() - Returns a string representation of the object. wait() - Causes current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.

Dot operator in java

- It enables you to access instance variables of any objects within a class. It is used to call object methods. "*the **dot***" connects classes and objects to members.
- when you are connecting a **class** name to one of its **static** fields. An example of this is the dot between "System" and "out" in the statements we use to print stuff to the console window. System is the name of a class included in every Java implementation. It has an object reference variable that points to a *PrintStream* object for the console. So, "System.out.println("text") invokes the println() method of the System.out object.

Using Scanner class(contd..)

- The **Java Scanner** class breaks the input into tokens using a delimiter that is whitespace by default. It provides many methods to read and parse various primitive values

Method	Description
public String next()	it returns the next token from the scanner.
public String nextLine()	it moves the scanner position to the next line and returns the value as a string.
public byte nextByte()	it scans the next token as a byte.
public short nextShort()	it scans the next token as a short value.
public int nextInt()	it scans the next token as an int value.
public long nextLong()	it scans the next token as a long value.
public float nextFloat()	it scans the next token as a float value.
public double nextDouble()	it scans the next token as a double value.

Static method

- A static method belongs to the class rather than object of a class.
- A static method can be invoked without the need for creating an instance of a class.
- static method can access static data member and can change the value of it.
- There are two main restrictions for the static method. They are:
    - The static method can not use non static data member or call non-static method directly.
    - this and super cannot be used in static context

Static variable

•When a variable is declared with the keyword “static”, its called a “**class variable**”.

•All instances share the same copy of the variable. A class variable can be accessed directly with the class, without the need to create a instance. It makes your program **memory efficient.**

Static block

Is used to initialize the static data member. It is executed before main method at the time of classloading. So this is one of the way to **execute a program without main() method.**

`switch` Statement Rules

- The switch-expression must yield a value of char, byte, short, or int type and

must always be enclosed in parentheses.

- The value1, ..., and valueN must have the same data type as the value of the switch-expression.

- The resulting statements in the case statement are executed when the value in the case statement matches the value of the switch-expression. (The case statements are executed in sequential order.)

- The keyword break is optional, but it should be used at the end of each case in order to terminate the remainder of the switch statement.

- If the break statement is not present, the next case statement will be executed.

`switch` Statement Rules, cont.

- The default case, which is optional, can be used to perform actions when none of the specified cases is true.
- The order of the cases (including the default case) does not matter. However, it is a good programming style to follow the logical sequence of the cases and place the default case at the end.

Collection Framework

- Provides an architecture to store and manipulate the group of objects.
- Achieve all the operations that you perform on a data such as **searching, sorting, insertion, manipulation, and deletion.**
- Java Collection means a single unit of objects.
- Java Collection framework provides many **interfaces** (Set, List, Queue, Deque) and **classes** (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet).

Collection Interface

- Interface which is implemented by all the classes in the collection framework.
- It declares the methods that every collection will have.

Collection interface builds the foundation on which the collection framework depends.

- Some of the methods of Collection interface are
  - Boolean add (Object obj)
  - Boolean addAll (Collection c)
  - void clear(), etc.

  which are implemented by all the subclasses of Collection interface.

ArrayList v/s LinkedList

ArrayList	LinkedList
1) ArrayList internally uses a **dynamic array** to store the elements.	LinkedList internally uses a **doubly linked list** to store the elements.
2) Manipulation with ArrayList is **slow** because it internally uses an array. If any element is removed from the array, all the bits are shifted in memory.	Manipulation with LinkedList is **faster** than ArrayList because it uses a doubly linked list, so no bit shifting is required in memory.
3) An ArrayList class can **act as a list** only because it implements List only.	LinkedList class can **act as a list and queue** both because it implements List and Deque interfaces.
4) ArrayList is **better for storing and accessing** data.	LinkedList is **better for manipulating** data.

ArrayList v/s Vector

ArrayList	Vector
1) ArrayList is **not synchronized.**	Vector is **synchronized.**
2) ArrayList **increments 50%** of current array size if the number of elements exceeds from its capacity.	Vector **increments 100%** means doubles the array size if the total number of elements exceeds than its capacity.
3) ArrayList is **not a legacy** class. It is introduced in JDK 1.2.	Vector is a **legacy** class.
4) ArrayList is **fast** because it is non-synchronized.	Vector is **slow** because it is synchronized, i.e., in a multithreading environment, it holds the other threads in runnable or non-runnable state until current thread releases the lock of the object.
5) ArrayList uses the **Iterator** interface to traverse the elements.	A Vector can use the **Iterator** interface or **Enumeration** interface to traverse the elements.

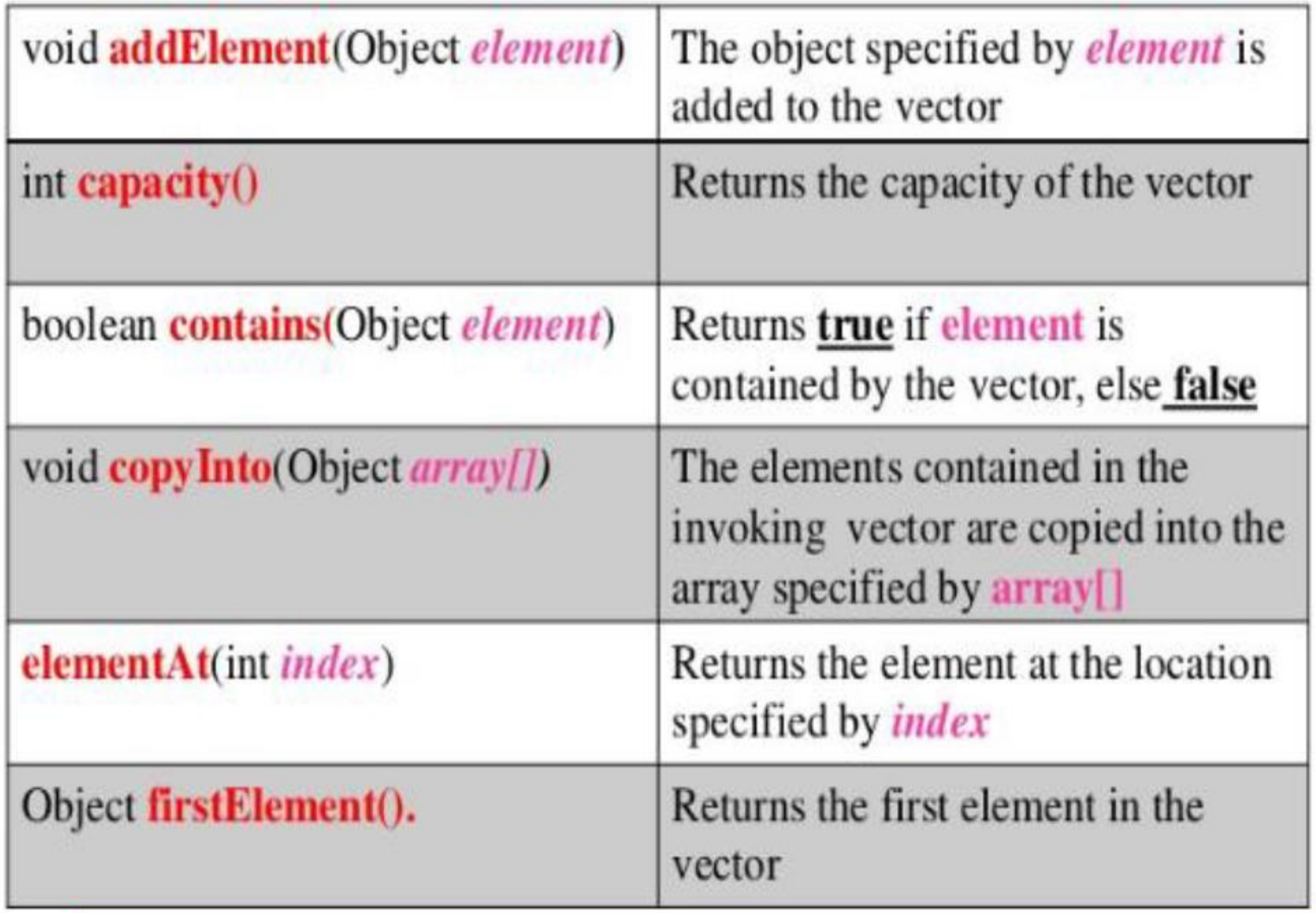
Vectors

- Vector implements a **dynamic array of objects**.
- Vector proves to be very useful **if you don't know the size of the array in advance** or you just need one that **can change sizes o**ver the lifetime of a program.
- Vector can contain heterogeneous objects
- We **cannot store elements of primitive data type;** first it need to be converted to objects. A vector can store any objects.
- Its defined in ***java.util*** package and class member of the Java Collections Framework.

Vectors

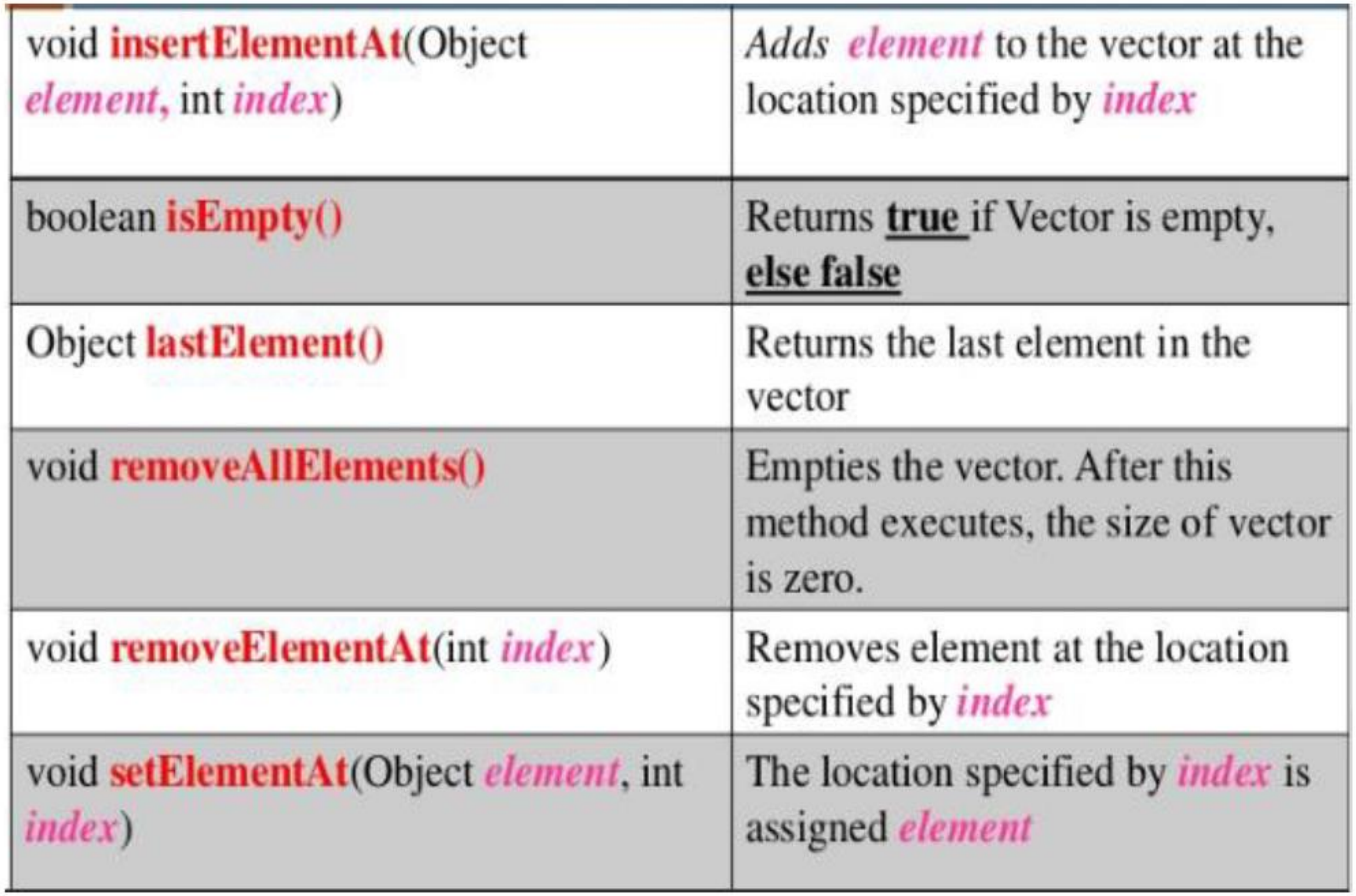
- Vector implements List Interface
- A vector has an initial capacity, if this capacity is reached then size of vector automatically increases.
- This default initial capacity of vectors are 10.
- Each vector tries to optimize storage management by maintaining a *capacity* and a *capacityIncrement* arguments.
- To traverse elements of a vector class we use **Enumeration** interface.

Vector Methods



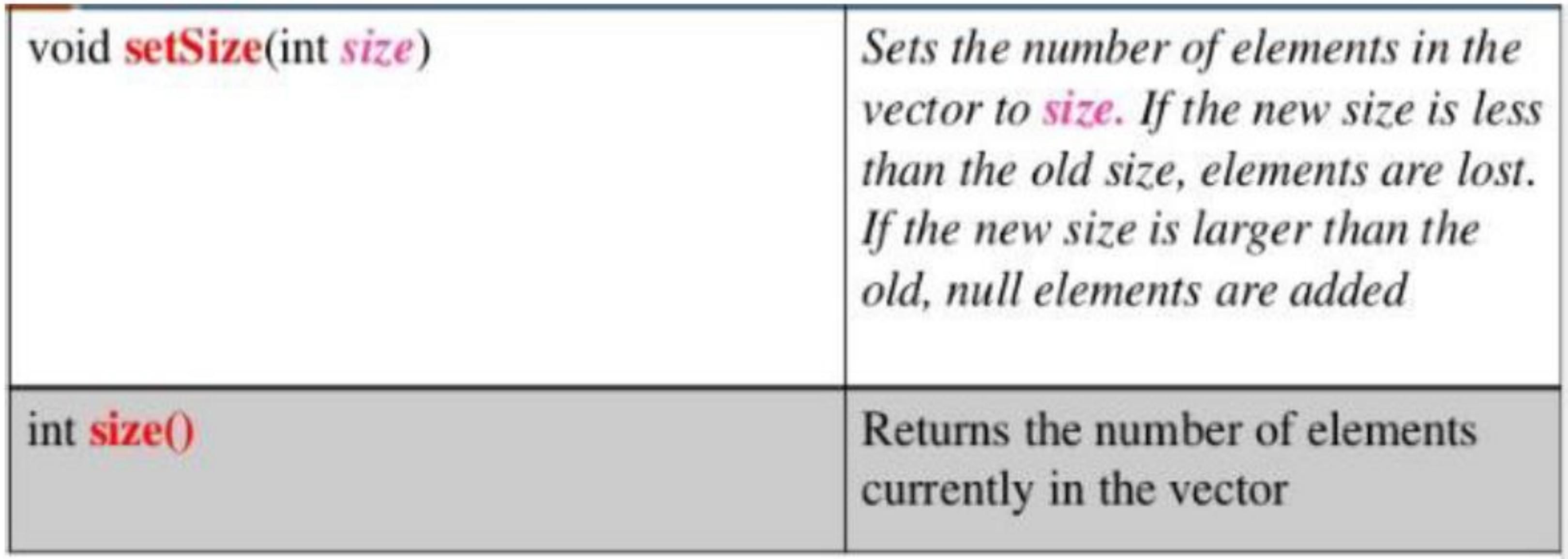
void **addElement**(Object ***element***)	The object specified by ***element*** is added to the vector
int **capacity()**	Returns the capacity of the vector
boolean **contains(**Object ***element***)	Returns **true** if **element** is contained by the vector, else **false**
void **copyInto**(Object ***array[]***)	The elements contained in the invoking vector are copied into the array specified by **array[]**
elementAt(int ***index***)	Returns the element at the location specified by ***index***
Object **firstElement().**	Returns the first element in the vector

Vector Methods



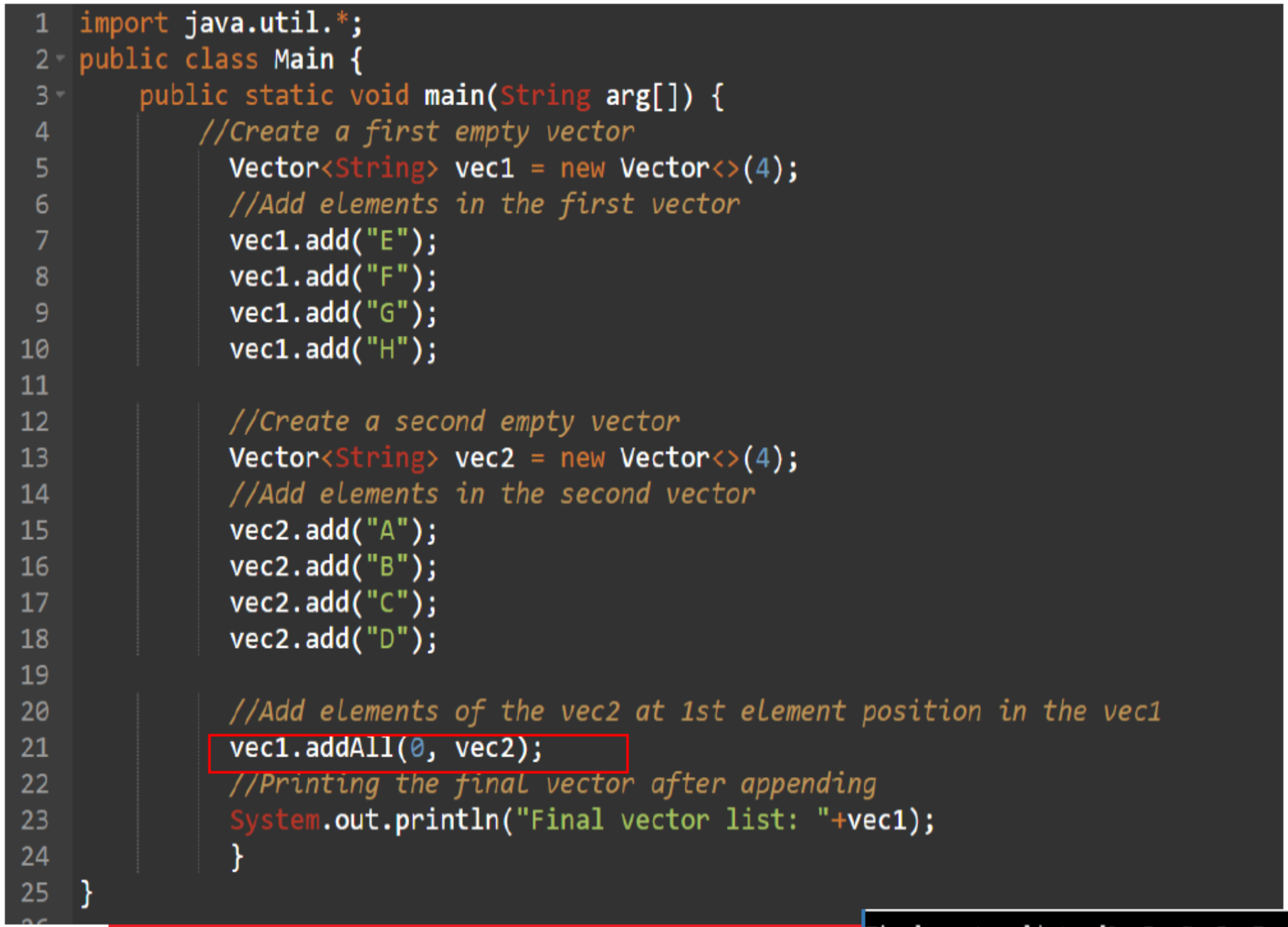
void **insertElementAt**(Object ***element***, int ***index***)	*Adds* ***element*** to the vector at the location specified by ***index***
boolean **isEmpty()**	Returns **true** if Vector is empty, **else false**
Object **lastElement()**	Returns the last element in the vector
void **removeAllElements()**	Empties the vector. After this method executes, the size of vector is zero.
void **removeElementAt**(int ***index***)	Removes element at the location specified by ***index***
void **setElementAt**(Object ***element***, int ***index***)	The location specified by ***index*** is assigned ***element***

Vector Methods



void **setSize**(int *size*)	*Sets the number of elements in the vector to* ***size.*** *If the new size is less than the old size, elements are lost. If the new size is larger than the old, null elements are added*
int **size()**	Returns the number of elements currently in the vector

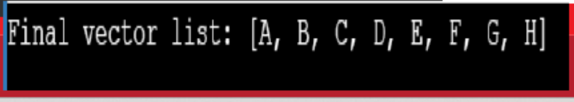
Example: boolean addAll(int index, Collection<? extends E> c)



```java
import java.util.*;
public class Main {
    public static void main(String arg[]) {
        //Create a first empty vector
         Vector<String> vec1 = new Vector<>(4);
         //Add elements in the first vector
         vec1.add("E");
         vec1.add("F");
         vec1.add("G");
         vec1.add("H");

         //Create a second empty vector
         Vector<String> vec2 = new Vector<>(4);
         //Add elements in the second vector
         vec2.add("A");
         vec2.add("B");
         vec2.add("C");
         vec2.add("D");

         //Add elements of the vec2 at 1st element position in the vec1
         vec1.addAll(0, vec2);
         //Printing the final vector after appending
         System.out.println("Final vector list: "+vec1);
         }
}
```



```
Final vector list: [A, B, C, D, E, F, G, H]
```

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.
- It is available since JDK 1.5.
- The **StringBuilder** in Java represents a mutable sequence of characters. Since the String Class in Java creates an immutable sequence of characters, the StringBuilder class provides an alternative to String Class, as it creates a mutable sequence of characters.
- The function of StringBuilder is very much similar to the StringBuffer class, as both of them provide an alternative to String Class by making a mutable sequence of characters. However the StringBuilder class differs from the StringBuffer class on the basis of synchronization.

- The StringBuilder class provides no guarantee of synchronization whereas the StringBuffer class does. Therefore this class is designed for use as a drop-in replacement for StringBuffer in places where the StringBuffer was being used by a single thread (as is generally the case).
- Where possible, it is recommended that this class be used in preference to StringBuffer as it will be faster under most implementations. Instances of StringBuilder are not safe for use by multiple threads. If such synchronization is required then it is recommended that StringBuffer be used.

No.	StringBuffer	StringBuilder
1)	StringBuffer is synchronized i.e. thread safe. It means two threads can't call the methods of StringBuffer simultaneously.	StringBuilder is non-synchronized i.e. not thread safe. It means two threads can call the methods of StringBuilder simultaneously.
2)	StringBuffer is less efficient than StringBuilder.	StringBuilder is more efficient than stringBuffer
3)	StringBuffer was introduced in Java 1.0	StringBuilder was introduced in Java 1.5