To become proficient in Spring Boot development, you should focus on mastering the following concepts:

1. \*\*Core Spring Framework\*\*: Understand dependency injection, inversion of control, and Spring beans.

2. \*\*Spring Boot\*\*: Learn how to create and configure Spring Boot applications, including starters, auto-configuration, and externalized configuration.

3. \*\*RESTful Web Services\*\*: Familiarize yourself with building REST APIs using Spring Boot, including handling HTTP requests, responses, and RESTful principles.

4. \*\*Spring Data JPA\*\*: Learn how to use Spring Data JPA to interact with databases, including CRUD operations and querying data.

5. \*\*Spring Security\*\*: Understand how to implement authentication and authorization in Spring Boot applications.

6. \*\*Testing\*\*: Learn how to write unit tests, integration tests, and end-to-end tests for Spring Boot applications using frameworks like JUnit and Mockito.

7. \*\*Dependency Management\*\*: Understand how to manage dependencies using tools like Maven or Gradle.

8. \*\*Microservices Architecture\*\*: Gain knowledge of designing and building microservices using Spring Boot.

9. \*\*Containerization\*\*: Learn how to deploy Spring Boot applications as Docker containers.

10. \*\*Monitoring and Logging\*\*: Understand how to monitor and log Spring Boot applications effectively using tools like Actuator and logging frameworks like Logback or Log4j.

Mastering these concepts will provide a strong foundation for becoming a proficient Spring Boot developer.

Clear Road Map : <https://www.linkedin.com/posts/rajatgajbhiye_spring-springboot-java-activity-7207354340917862401-Uwbe?utm_source=share&utm_medium=member_android>

interview questions: <https://www.linkedin.com/posts/soma-sharma-51643a141_500-interview-questions-for-java-developer-activity-7184460308789309440-bFvn?utm_source=share&utm_medium=member_android>

basic in spring : <https://www.linkedin.com/posts/raul-junco_i-learned-backend-engineering-after-learning-activity-7184523294153306113-5ZTK?utm_source=share&utm_medium=member_android>

Java project: <https://www.linkedin.com/posts/rajatgajbhiye_java-javaresources-corejava-activity-7212311258656608257-Akq3?utm_source=share&utm_medium=member_android>

JAVA + COLLECTION + OOPS + SPRING / SPRING BOOT

### **Core Java Concepts**

1. Object-Oriented Programming (OOP)
   * Classes and Objects
   * Inheritance
   * Polymorphism
   * Encapsulation
   * Abstraction
2. Java Collections Framework
   * Lists, Sets, Maps, Queues
   * Iterators
   * Collection utilities (sorting, searching, etc.)
3. Exception Handling
   * Checked vs Unchecked Exceptions
   * Try-Catch-Finally blocks
   * Custom Exceptions
4. Java I/O
   * Streams (InputStream, OutputStream)
   * Readers and Writers
   * File Handling
5. Concurrency and Multithreading
   * Thread lifecycle
   * Synchronization
   * Executors and Thread Pools
   * Concurrent Collections
6. Java 8 and Above Features
   * Lambda Expressions
   * Stream API
   * Optional
   * Functional Interfaces
   * Default and Static Methods in Interfaces
7. Annotations and Reflection
   * Defining and using annotations
   * Reflection API

### **Spring Framework Concepts**

1. Dependency Injection (DI) and Inversion of Control (IoC)
   * Bean lifecycle
   * ApplicationContext and BeanFactory
   * Scopes of beans (singleton, prototype)
2. Spring Core
   * Spring Container
   * Spring Beans
   * Configuration (XML, Java Config, Annotations)
3. Spring AOP (Aspect-Oriented Programming)
   * Aspects, Join Points, Pointcuts, Advices
   * Use cases of AOP (e.g., logging, transactions)
4. Spring Data Access
   * Spring JDBC
   * ORM with Spring (Hibernate, JPA)
   * Transaction Management
5. Spring MVC
   * Controllers
   * View Resolvers
   * Model and View
   * Form Handling
   * Validation
6. Spring Security
   * Authentication and Authorization
   * Security Filters and Interceptors
   * Method Security

### **Spring Boot Specific Concepts**

1. Spring Boot Basics
   * Spring Boot Starters
   * Auto-Configuration
   * Spring Boot CLI
   * Application Properties and YAML configuration
2. Building RESTful Web Services
   * REST Controllers
   * Request Mapping
   * Path Variables and Request Parameters
   * Response Entity and Status Codes
   * Exception Handling in REST
3. Data Access with Spring Boot
   * Spring Data JPA
   * Repositories (CrudRepository, JpaRepository)
   * Query Methods and Custom Queries
   * Pagination and Sorting
4. Spring Boot Testing
   * Unit Testing (JUnit, Mockito)
   * Integration Testing
   * Testing REST Controllers (MockMvc)
   * Testing Repositories
5. Spring Boot Actuator
   * Health Checks
   * Metrics and Monitoring
   * Custom Actuator Endpoints
6. Spring Boot DevTools
   * Automatic Restart
   * LiveReload
7. Spring Boot Profiles
   * Environment-Specific Configurations
   * @Profile Annotation
   * Activating Profiles

### **Additional Concepts**

1. Microservices Architecture
   * Building Microservices with Spring Boot
   * Communication between Microservices (RestTemplate, Feign Client)
   * Service Discovery (Eureka)
   * Circuit Breaker (Hystrix, Resilience4j)
   * Configuration Management (Spring Cloud Config)
2. Reactive Programming
   * Project Reactor
   * Mono and Flux
   * WebFlux
3. Containerization and Deployment
   * Docker
   * Kubernetes
   * Continuous Integration/Continuous Deployment (CI/CD)

# Job Requirements:

1. Spring Framework
2. Hibernate
3. Mandatory skills : Java , SpringBoot , Microservice , Angular(10+ versions) or React
4. Years of exp: 3.5 to 7 yrs
5. Location: Chennai only
6. ? In-depth knowledge of core Java, Spring and Hibernate.
7. ? Working knowledge of Angular or React JS.
8. ? Good communication and unit testing knowledge.
9. ? It is good to have knowledge of one of the cloud platforms like AWS/Azure/PCF.
10. ? Work experience in frameworks like JPA, Struts/Spring MVC, Spring Core, Spring AOP, and Spring Data.
11. ? Familiar with Continuous Integration methodologies and tools, including Jenkins.
12. ? Exposure to Microservices, Docker, Kubernetes and cloud deployment.
13. Skill Name: Java AWS
14. Mandatory Skills: Java , SpringBoot , Microservice , RestAPI, AWS (Lambda , EC2 , SQS , S3 , ECS)
15. Years of exp: 3.5 to 7 yrs
16. Location: Chennai only
18. Essential Skills:
19. · 3+ years of in-depth knowledge in core Java, REST, Hibernates, AWS App Development
20. · 2+ years of experience in spring boot
21. · 1+ years of experience in developing Microservices and docker.
22. · 1+ years of experience in serverless architecture and exposure using AWS Lambda/Azure Functions
23. · Minimum 6 months in using AWS PaaS services like storage, Queues, Database, security. It could be the following in AWS
24. · 1+ years of experience in working in any of the cloud platforms like AWS ? App development Services, Integration Services, API Management. etc.
25. · AWS: SQS, SNS, Lambda, Step Function, API Gateway, Dynamo DB, ECS, EC2, Load Balancer, Secrets manager
26. · 2+ years of experience in RESTful Http services design
27. · 2+ years of experience in Java script, JQuery, Bootstrap, Html 5, CSS3
28. · 2+ years of experience with SQL Server, Postgre SQL writing stored procedures, performance tuning and identifying deadlocks, transactions, and data locking/blocking scenarios
29. · Good communication and unit testing knowledge.
30. · Work experience in frameworks like JPA, Struts/Spring MVC, Spring Core, Spring AOP, and Spring Data
31. · Familiar with Continuous Integration methodologies and tools, including Jenkins
32. · Good to have: Docker Containerization Cloud, Kubernetes\_Cloud , UI frameworks

# Questions:

1. interview concepts & road map . it's enough to finish the java & spring - <https://www.linkedin.com/posts/rajatgajbhiye_java-corejava-activity-7224285224208093184-aDRZ?utm_source=share&utm_medium=member_android> \*\*\*\*\*\*\*\*

## <https://www.linkedin.com/posts/rajatgajbhiye_java-sponsoredpost-activity-7220281958931587072-d0y4?utm_source=share&utm_medium=member_android> - concepts of java spring boot

## java concepts <https://www.linkedin.com/posts/rajatgajbhiye_java-javaroadmap-corejava-activity-7216297554458640385-FMHz?utm_source=share&utm_medium=member_android>

1. spring boot concepts <https://www.linkedin.com/posts/rajatgajbhiye_spring-springboot-java-activity-7207354340917862401-Uwbe?utm_source=share&utm_medium=member_android>

## Oops question - <https://www.linkedin.com/posts/jyoti-kumari-668031304_ugcPost-7193068428138549248-tjCE?utm_source=share&utm_medium=member_android>

<https://www.linkedin.com/posts/codepapa360_top-30-oops-interview-questions-activity-7193572788320755712-22GD?utm_source=share&utm_medium=member_android>

<https://www.linkedin.com/posts/rajatgajbhiye_from-my-past-interview-experiences-here-activity-7221476646388682755-Kn3t?utm_source=share&utm_medium=member_android>

1. Questions : <https://www.linkedin.com/posts/prakash-shukla_core-java-interview-qa-activity-7226483674966368256-fKXv?utm_source=share&utm_medium=member_desktop>

<https://www.linkedin.com/posts/sai-kumar-medipelli-5348121a0_java-learning-interview-questions-activity-7226513575572291584-RxsQ?utm_source=share&utm_medium=member_android>

<https://www.linkedin.com/posts/jainshrayansh_softwareengineer-java-springboot-activity-7226829291336949762-xpZU?utm_source=share&utm_medium=member_android>

1. <https://www.linkedin.com/posts/rajatgajbhiye_java-corejava-javainterview-activity-7214123223532113920-ttDB?utm_source=share&utm_medium=member_android>
2. <https://www.linkedin.com/posts/rajatgajbhiye_java-javaroadmap-corejava-activity-7216297554458640385-FMHz?utm_source=share&utm_medium=member_android> - road map
3. <https://www.linkedin.com/posts/rani-dhage_java-javaresources-corejava-activity-7215312174783578112-LKbe?utm_source=share&utm_medium=member_android>
4. <https://chatgpt.com/share/d0d9c01c-96ed-462c-954c-1e575577433b> - JPA concepts should know from chat gpt
5. <https://www.linkedin.com/posts/rajatgajbhiye_java-technical-interview-questions-ugcPost-7215226028464492545-6epk?utm_source=share&utm_medium=member_android>
6. <https://www.linkedin.com/posts/rani-dhage_java-corejava-javainterview-activity-7214494946618187776-95ng?utm_source=share&utm_medium=member_android> \*\*\*\*\* priority. should know all the concepts. All in DSA - <https://www.linkedin.com/posts/rajatgajbhiye_dsa-datastructures-algorithm-activity-7214485606519697408-9vpi?utm_source=share&utm_medium=member_android>
7. spring swagger hide filed - used to hide the modal unwanted params <https://www.baeldung.com/spring-swagger-hide-field>
8. <https://www.linkedin.com/posts/nikita-sharma-2809_springboot-activity-7216668533579997184-nZMG?utm_source=share&utm_medium=member_android>
9. Interview questions book

<https://www.linkedin.com/feed/update/urn:li:activity:7193273854490808322?utm_source=share&utm_medium=member_android>

1. <https://chatgpt.com/share/29de2418-cabb-46ba-a073-eef08a7a5f9f>
2. <https://www.linkedin.com/posts/rajatgajbhiye_java-javaresources-corejava-activity-7211586521424928770-FKR8?utm_source=share&utm_medium=member_android>
3. <https://www.linkedin.com/posts/rajatgajbhiye_java-technical-interview-questions-ugcPost-7206616694646087681-aDot?utm_source=share&utm_medium=member_android>
4. <https://www.linkedin.com/posts/rajatgajbhiye_java-javadev-corejava-activity-7213398434219757568-KiMy?utm_source=share&utm_medium=member_android>
5. <https://www.linkedin.com/posts/shubham-soni-374a86175_sprint-boot-handy-doc-ugcPost-7220297755456417792-8NBF?utm_source=share&utm_medium=member_android>
6. <https://www.linkedin.com/posts/algotutor_java-technical-interview-questions-activity-7205517739107504128-k3YJ?utm_source=share&utm_medium=member_android>
7. <https://www.baeldung.com/spring-controller-vs-restcontroller>
8. <https://www.linkedin.com/posts/rajatgajbhiye_microservices-ugcPost-7202523505509195776-Xkpy?utm_source=share&utm_medium=member_android>
9. <https://www.linkedin.com/posts/hcl-higher-coading-language_java-questions-activity-7209832991831781376-s0Tj?utm_source=share&utm_medium=member_android>
10. <https://www.linkedin.com/posts/rani-dhage_java-question-ugcPost-7207246724992610305-wWLQ?utm_source=share&utm_medium=member_android>
11. <https://www.linkedin.com/posts/rani-dhage_java-interview-coding-activity-7210597464490315776-mL-o?utm_source=share&utm_medium=member_android>
12. <https://www.linkedin.com/posts/algotutor_java-technical-interviews-questions-activity-7214939823537631232-fDZh?utm_source=share&utm_medium=member_android>
13. <https://www.linkedin.com/posts/shristi-mishra-43484b303_java-notes-ugcPost-7211374602516611073-9hid?utm_source=share&utm_medium=member_android>
14. queryforList, query, update, all the options of jdbc and usage
15. Jxjxjxjx

# Answer:

## Java

## How Java Works?

## - JDK, JRE, JVM

## - Variables and Datatypes

## - Primitive vs Reference

## - Data types

## - Naming Conventions

## - Casting

## - Operators Keywords

## - Unicode System

## - Reading Input

## If Else

## - Ternary

## - For loop

## - While loop

## - Do-while loop

## - For each loop

## - Nested loop

## - Switch

## - Break and Continue

## Method Syntax

The syntax for defining a method in Java includes the following components:

1. Access Modifier
2. Return Type
3. Method Name
4. Parameter List (in parentheses)
5. Method Body (in curly braces)

accessModifier returnType methodName(parameterList) {

// Method body

}

public void sayHello() { System.out.println("Hello, World!"); }

// Method with parameters and with return type

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

## - Parameters

Parameters are variables that are passed to methods to provide input values. They are defined within the parentheses in the method declaration. Parameters allow methods to perform tasks with different data inputs.

**Types of Parameters:**

1. **Formal Parameters**: Defined in the method declaration.
2. **Actual Parameters (Arguments)**: Values passed to the method when it is called.

public class ParameterExample {

// Method with parameters

public void greet(String name) {

System.out.println("Hello, " + name + "!");

}

public static void main(String[] args) {

ParameterExample example = new ParameterExample();

example.greet("Alice"); // "Alice" is the actual parameter

}

}

## - Method Scope

Method scope refers to the visibility and lifetime of variables and methods within a program. In Java, the scope of a variable or method defines where it can be accessed within the code.

#### **Types of Scopes in Java**

1. **Local Scope**: Variables declared within a method.
2. **Class/Instance Scope**: Variables (fields) declared within a class but outside any method.
3. **Block Scope**: Variables declared within a block of code (inside curly braces {}).

## - Shadowing

Shadowing occurs when a local variable in a method or a block has the same name as a field (class/instance variable). The local variable "shadows" or hides the field within its scope.

public class ShadowingExample {

int x = 5; // Instance variable

void display() {

int x = 10; // Local variable that shadows the instance variable

System.out.println("Local x: " + x); // Prints 10

System.out.println("Instance x: " + this.x); // Prints 5

}

public static void main(String[] args) {

ShadowingExample example = new ShadowingExample();

example.display();

}

}

## Why we need array?

Arrays are essential in programming because they allow us to store multiple values of the same type in a single data structure. They provide a systematic way to manage large amounts of data, enabling efficient storage, access, and manipulation of elements.

## - What is array?

An array is a collection of elements, all of the same type, stored in contiguous memory locations. Each element can be accessed by its index, with indexing starting from 0.

## - Internal working of array.

Arrays are allocated in contiguous memory locations, meaning that the elements of the array are stored next to each other in memory. This allows for efficient access using an index, as the address of any element can be calculated using its index and the base address of the array.

int[] numbers = new int[5]; // Declaration of an array of integers with 5 elements

numbers[0] = 10;

numbers[1] = 20;

numbers[2] = 30;

numbers[3] = 40;

numbers[4] = 50;

## - Dynamic memory allocation

In Java, arrays are dynamically allocated. When you declare an array, you need to specify its size. The Java runtime environment allocates memory for the array elements during runtime.

int size = 10;

int[] dynamicArray = new int[size];

## - Array input and output

import java.util.Scanner;

public class Main {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.print("Enter the number of elements: ");

int n = scanner.nextInt();

int[] array = new int[n];

System.out.println("Enter the elements:");

for (int i = 0; i < n; i++) {

array[i] = scanner.nextInt();

}

System.out.println("Array elements are:");

for (int i = 0; i < n; i++) {

System.out.print(array[i] + " ");

}

}

}

## - Array Passing in function

Arrays can be passed to functions just like other data types. When an array is passed to a function, the reference to the array is passed, not the actual array itself.

public class Main {

public static void main(String[] args) {

int[] array = {1, 2, 3, 4, 5};

printArray(array);

}

public static void printArray(int[] array) {

for (int element : array) {

System.out.print(element + " ");

}

}

}

## - Multidimensional array

A multidimensional array is an array of arrays. The most common form is the two-dimensional array, which can be thought of as a table with rows and columns.

public class Main {

public static void main(String[] args) {

int[][] matrix = {

{1, 2, 3},

{4, 5, 6},

{7, 8, 9}

};

// Printing the two-dimensional array

for (int i = 0; i < matrix.length; i++) {

for (int j = 0; j < matrix[i].length; j++) {

System.out.print(matrix[i][j] + " ");

}

System.out.println();

}

}

}

## OOPS

**Object-Oriented Programming** is a methodology or paradigm to design a program using classes and objects. It simplifies software development and maintenance. It is based on several key principles, including encapsulation, inheritance, polymorphism, and abstraction. Java is a popular object-oriented programming language that fully supports these principles.

## Object and class

#### **Objects**

An object is a real-world entity with state and behavior. In Java, an object is an instance of a class. It contains data (fields or properties) and methods to manipulate that data.

public class Main {

public static void main(String[] args) {

// Creating an object of the class Car

Car myCar = new Car();

// Setting the object's state

myCar.color = "Red";

myCar.model = "Tesla Model 3";

// Calling the object's method

myCar.displayInfo();

}

}

#### **Classes**

A class is a blueprint for creating objects. It defines a set of properties (fields) and methods that the created objects will have.

public class Car {

// Fields (properties) of the class

String color;

String model;

// Method of the class

void displayInfo() {

System.out.println("Car Model: " + model);

System.out.println("Car Color: " + color);

}

}

## - Constructor

A constructor is a special method that is called when an object is instantiated. It is used to initialize the object's state. Constructors have the same name as the class and do not have a return type.

**Types of Constructors:**

1. **Default Constructor**: A no-argument constructor that is provided by Java if no other constructor is defined.

public class Car {

String color;

String model;

// Default constructor

public Car() {

color = "Black";

model = "Default Model";

}

void displayInfo() {

System.out.println("Car Model: " + model);

System.out.println("Car Color: " + color);

}

public static void main(String[] args) {

Car myCar = new Car(); // Calls the default constructor

myCar.displayInfo();

}

}

**Output :**

Car Model: Default Model

Car Color: Black

1. **Parameterized Constructor**: A constructor that accepts parameters to initialize the object's state with specific values.

public class Car {

String color;

String model;

// Parameterized constructor

public Car(String color, String model) {

this.color = color;

this.model = model;

}

void displayInfo() {

System.out.println("Car Model: " + model);

System.out.println("Car Color: " + color);

}

public static void main(String[] args) {

Car myCar = new Car("Red", "Tesla Model 3"); // Calls the parameterized constructor

myCar.displayInfo();

}

}

**Output 👍**

Car Model: Tesla Model 3

Car Color: Red

## - Inheritance

Inheritance allows one class to inherit the fields and methods of another class. This helps in reusing code and establishing a natural hierarchical relationship between classes.

class Animal {

public void eat() {

System.out.println("This animal eats food.");

}

}

class Dog extends Animal {

public void bark() {

System.out.println("The dog barks.");

}

}

public class Main {

public static void main(String[] args) {

Dog dog = new Dog();

dog.eat(); // Inherited method

dog.bark();

}

}

## - Polymorphism

Polymorphism allows objects of different classes to be treated as objects of a common superclass. It can be achieved through method overriding and method overloading.

* 1. **Method Overriding**: A subclass provides a specific implementation of a method that is already defined in its superclass.
  2. **Method Overloading**: Multiple methods with the same name but different parameters within the same class. Method overloading is a compile-time polymorphism feature in Java.

**Method Overriding :**

class Animal {

public void sound() {

System.out.println("This animal makes a sound.");

}

}

class Dog extends Animal {

@Override

public void sound() {

System.out.println("The dog barks.");

}

}

class Cat extends Animal {

@Override

public void sound() {

System.out.println("The cat meows.");

}

}

public class Main {

public static void main(String[] args) {

Animal myDog = new Dog();

Animal myCat = new Cat();

myDog.sound(); // The dog barks.

myCat.sound(); // The cat meows.

}

}

**Method overload:**

public class Calculator {

// Method to add two integers

public int add(int a, int b) {

return a + b;

}

// Method to add three integers

public int add(int a, int b, int c) {

return a + b + c;

}

// Method to add two double values

public double add(double a, double b) {

return a + b;

}

// Method to add two strings

public String add(String a, String b) {

return a + b;

}

public static void main(String[] args) {

Calculator calc = new Calculator();

// Using the overloaded methods

System.out.println("Sum of two integers: " + calc.add(10, 20));

System.out.println("Sum of three integers: " + calc.add(10, 20, 30));

System.out.println("Sum of two doubles: " + calc.add(10.5, 20.5));

System.out.println("Concatenation of two strings: " + calc.add("Hello", " World"));

}

}

## - Encapsulation

Encapsulation is the mechanism of restricting direct access to some of an object's components and providing controlled access via methods. It helps in protecting the internal state of the object and ensures that the object maintains a valid state.

class Person {

private String name;

private int age;

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public int getAge() {

return age;

}

public void setAge(int age) {

if (age > 0) {

this.age = age;

}

}

}

public class Main {

public static void main(String[] args) {

Person person = new Person();

person.setName("Alice");

person.setAge(30);

System.out.println("Name: " + person.getName());

System.out.println("Age: " + person.getAge());

}

}

## - Abstraction

Abstraction is the concept of hiding the complex implementation details and showing only the essential features of an object. This is typically achieved through abstract classes and interfaces.

**Abstract class:**

abstract class Animal {

abstract void sound();

public void sleep() {

System.out.println("This animal sleeps.");

}

}

class Dog extends Animal {

@Override

void sound() {

System.out.println("The dog barks.");

}

}

public class Main {

public static void main(String[] args) {

Dog dog = new Dog();

dog.sound(); // The dog barks.

dog.sleep(); // This animal sleeps.

}

}

**Abstract Interface :**

interface Animal {

void sound();

void sleep();

}

class Dog implements Animal {

@Override

public void sound() {

System.out.println("The dog barks.");

}

@Override

public void sleep() {

System.out.println("The dog sleeps.");

}

}

public class Main {

public static void main(String[] args) {

Dog dog = new Dog();

dog.sound(); // The dog barks.

dog.sleep(); // The dog sleeps.

}

}

## What is String?

It is basically an object that represents sequence of char values.

It is a widely used class for storing and manipulating text. The String class in Java is immutable, meaning once a String object is created, its content cannot be changed. This immutability is beneficial for various reasons, including security, synchronization, and caching.

## - String Creation: string literal

There are two ways to create String object:

1. By string literal
2. By new keyword

**Literal**: Using double quotes. These strings are stored in the string pool. Each time you create a string literal, the JVM checks the "string constant pool" first. If the string already exists in the pool, a reference to the pooled instance is returned. If the string doesn't exist in the pool, a new string instance is created and placed in the pool.

String str = "Hello";

**Using new Keyword**: This creates a new String object in the heap. In such case, [JVM](https://www.javatpoint.com/jvm-java-virtual-machine) will create a new string object in normal (non-pool) heap memory, and the literal "Welcome" will be placed in the string constant pool. The variable str will refer to the object in a heap (non-pool).

String str = new String("Hello");

**char** ch[]={'s','t','r','i','n','g','s'};

String s2=**new** String(ch);//converting char array to string

## - String Pooling

String pooling is a mechanism in Java to optimize memory usage and improve performance by reusing String instances. The Java Virtual Machine (JVM) maintains a pool of strings, known as the "string pool" or "string intern pool". When a new string literal is created, the JVM checks if an identical string already exists in the pool. If it does, the existing string's reference is returned; otherwise, a new string is created and added to the pool.

String str1 = "Hello";

String str2 = "Hello";

System.out.println(str1 == str2); // true

## - String Immutability

In Java, String objects are immutable, meaning once a String object is created, it cannot be modified. Any operation that seems to modify the string actually creates a new String object with the modified content.

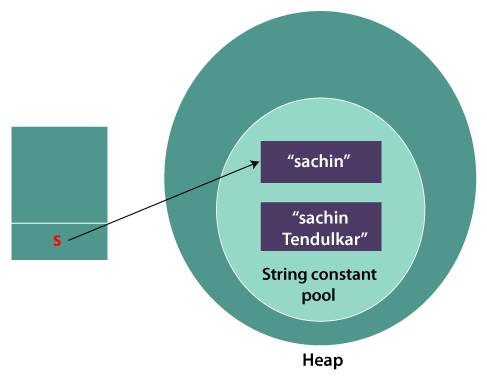
**Benefits of Immutability**:

1. **Thread Safety**: Immutable objects are inherently thread-safe as their state cannot change after creation.
2. **Caching and Reuse**: Because strings are immutable, they can be safely shared and reused, which is why string pooling works efficiently.
3. **Security**: Immutable objects are secure because their state cannot be altered by malicious code.

String s="Sachin";

s.concat(" Tendulkar");//concat() method appends the string at the end

System.out.println(s);//will print Sachin because strings are immutable objects



String s="Sachin";

s=s.concat(" Tendulkar");

System.out.println(s); // Sachin Tendulkar

### Why String objects are immutable in Java?

## As Java uses the concept of String literal. Suppose there are 5 reference variables, all refer to one object "Sachin". If one reference variable changes the value of the object, it will be affected by all the reference variables. That is why String objects are immutable in Java.

## String Comparison: == & equals() & compareTo()

**== Operator**: This compares the references (memory addresses) of the objects to see if they point to the same object.

String str1 = "Hello";

String str2 = "Hello";

String str3 = new String("Hello");

System.out.println(str1 == str2); // true

System.out.println(str1 == str3); // false

**equals() Method**: This compares the content of the strings to see if they are equal.

System.out.println(str1.equals(str2)); // true

System.out.println(str1.equals(str3)); // true

**compareTo :** The compareTo() method in Java is used to compare two String objects lexicographically. This method is defined in the Comparable interface, which the String class implements. The compareTo() method returns an integer that indicates the relationship between the two strings:

1. A negative integer if the calling string is lexicographically less than the argument string.
2. Zero if the calling string is lexicographically equal to the argument string.
3. A positive integer if the calling string is lexicographically greater than the argument string.

public class Main {

public static void main(String[] args) {

String str1 = "apple";

String str2 = "banana";

String str3 = "apple";

String str4 = "Apple";

// Comparing str1 with str2

int result1 = str1.compareTo(str2); // Negative integer because "apple" < "banana"

// Comparing str1 with str3

int result2 = str1.compareTo(str3); // Zero because "apple" == "apple"

// Comparing str1 with str4

int result3 = str1.compareTo(str4); // Positive integer because "apple" > "Apple" (case-sensitive)

// Output results

System.out.println("str1.compareTo(str2): " + result1);

System.out.println("str1.compareTo(str3): " + result2);

System.out.println("str1.compareTo(str4): " + result3);

}

}

**Output :**

str1.compareTo(str2): -1

str1.compareTo(str3): 0

str1.compareTo(str4): 32

str1.compareTo(str4):

The result is 32 because "apple" is lexicographically greater than "Apple". This result is due to the difference in ASCII values between lowercase and uppercase letters ('a' (97) and 'A' (65), respectively).

**Case-Insensitive Comparison :**

If you want to compare strings in a case-insensitive manner, you can use the compareToIgnoreCase() method:

int result = str1.compareToIgnoreCase(str2); // Zero because "apple" == "Apple" (case-insensitive)

## - PrintStream Class

The PrintStream class in Java is used to output data to a stream, typically for printing text to the console. It is part of the java.io package and provides methods for printing formatted representations of objects to a text output stream. System.out is a commonly used instance of PrintStream.

**Key Characteristics of PrintStream:**

1. **Automatic Flushing**: It can be configured to flush the output buffer automatically.
2. **No IOException**: Unlike other stream classes, PrintStream does not throw IOException; instead, it sets an internal flag that can be checked with the checkError() method.
3. **Convenience Methods**: Provides methods for printing various data types directly.

**Common Methods:**

1. print(): Prints a value.
2. println(): Prints a value followed by a new line.
3. printf(): Prints a formatted string.
4. format(): Alias for printf().

public class Main {

public static void main(String[] args) {

// Using System.out which is an instance of PrintStream

System.out.print("Hello, ");

System.out.println("World!");

int number = 42;

System.out.printf("Number: %d%n", number);

}

}

## - toString() Method

The toString() method in Java is used to provide a string representation of an object. When toString() is called on an object, it returns a string that "textually represents" this object. The default implementation in the Object class returns a string consisting of the class name followed by the object's hash code, but it is often overridden in custom classes to provide more meaningful representations.

class Person {

private String name;

private int age;

public Person(String name, int age) {

this.name = name;

this.age = age;

}

@Override

public String toString() {

return "Person{name='" + name + "', age=" + age + "}";

}

}

public class Main {

public static void main(String[] args) {

Person person = new Person("Alice", 30);

System.out.println(person.toString()); // Person{name='Alice', age=30}

System.out.println(person); // Implicit call to toString(): Person{name='Alice', age=30}

}

}

## - Format Specifiers

Format specifiers are used in formatted strings with methods like printf(), format(), and String.format(). They define how data should be formatted and displayed.

**Common Format Specifiers:**

1. %d: Decimal integer
2. %f: Floating-point number
3. %s: String
4. %c: Character
5. %b: Boolean
6. %n: Platform-specific newline

public class Main {

public static void main(String[] args) {

int intValue = 42;

double doubleValue = 3.14159;

String stringValue = "Hello";

System.out.printf("Integer: %d%n", intValue);

System.out.printf("Double: %.2f%n", doubleValue); // Two decimal places

System.out.printf("String: %s%n", stringValue);

}

}

## - String Concatenation Operator

In Java, the + operator is used to concatenate strings. When a string is concatenated with another data type, the other data type is converted to its string representation.

public class Main {

public static void main(String[] args) {

String greeting = "Hello";

String name = "Alice";

int age = 30;

String message = greeting + ", " + name + "! You are " + age + " years old.";

System.out.println(message); // Hello, Alice! You are 30 years old.

}

}

String s="Sachin"+" Tendulkar";

The code converted into:

String s=(**new** StringBuilder()).append("Sachin").append(" Tendulkar).toString();

## String buffer

StringBuffer is a class in Java used to create mutable sequences of characters. Unlike String, which is immutable, StringBuffer objects can be modified after they are created. This makes StringBuffer suitable for situations where you need to make a lot of modifications to a string, such as appending, inserting, or deleting characters.

**Key Characteristics of StringBuffer:**

1. **Mutable**: The contents of a StringBuffer can be changed after creation.
2. **Thread-Safe**: Methods of StringBuffer are synchronized, making it thread-safe. This means multiple threads can access a StringBuffer object without causing data inconsistency.
3. **Performance**: Because StringBuffer is synchronized, it might be slower than StringBuilder for single-threaded applications. If thread safety is not a concern, StringBuilder is preferred for better performance.

**Common Methods of StringBuffer:**

**Append**: Adds the specified string to the end of the buffer.

StringBuffer sb = new StringBuffer("Hello");

sb.append(" World");

System.out.println(sb); // Hello World

**Insert**: Inserts the specified string at the specified position.

sb.insert(5, ",");

System.out.println(sb); // Hello, World

**Delete**: Removes the characters from the specified start to end positions.

sb.delete(5, 6);

System.out.println(sb); // Hello World

**Replace**: Replaces the characters in the specified range with the specified string.

sb.replace(6, 11, "Java");

System.out.println(sb); // Hello Java

**Reverse**: Reverses the sequence of characters in the buffer.

sb.reverse();

System.out.println(sb); // avaJ olleH

**Capacity and Length**: capacity() returns the current capacity of the buffer, while length() returns the number of characters in the buffer.

int capacity = sb.capacity();

int length = sb.length();

## StringBuilder Class

StringBuilder is a class in Java that provides a mutable sequence of characters. It is similar to StringBuffer but is not synchronized, meaning it is not thread-safe. This makes StringBuilder faster and more efficient than StringBuffer in a single-threaded environment.

**Key Characteristics of StringBuilder:**

1. **Mutable**: The contents of a StringBuilder can be changed after creation.
2. **Not Thread-Safe**: Methods of StringBuilder are not synchronized, so it should not be used in a multi-threaded environment where multiple threads might access the same instance.
3. **Performance**: StringBuilder is faster than StringBuffer because it doesn't have the overhead of synchronization.

It has the same set of methods as stringBuffer.

StringBuilder sb = new StringBuilder("Hello");

sb.append(" World");

System.out.println(sb); // Hello World

## - String methods

The java.lang.String class provides many useful methods to perform operations on sequence of char values.

| **No.** | **Method** | **Description** |
| --- | --- | --- |
| 1 | [char charAt(int index)](https://www.javatpoint.com/java-string-charat) | It returns char value for the particular index |
| 2 | [int length()](https://www.javatpoint.com/java-string-length) | It returns string length |
| 3 | [static String format(String format, Object... args)](https://www.javatpoint.com/java-string-format) | It returns a formatted string. |
| 4 | [static String format(Locale l, String format, Object... args)](https://www.javatpoint.com/java-string-format) | It returns formatted string with given locale. |
| 5 | [String substring(int beginIndex)](https://www.javatpoint.com/java-string-substring) | It returns substring for given begin index. |
| 6 | [String substring(int beginIndex, int endIndex)](https://www.javatpoint.com/java-string-substring) | It returns substring for given begin index and end index. |
| 7 | [boolean contains(CharSequence s)](https://www.javatpoint.com/java-string-contains) | It returns true or false after matching the sequence of char value. |
| 8 | [static String join(CharSequence delimiter, CharSequence... elements)](https://www.javatpoint.com/java-string-join) | It returns a joined string. |
| 9 | [static String join(CharSequence delimiter, Iterable<? extends CharSequence> elements)](https://www.javatpoint.com/java-string-join) | It returns a joined string. |
| 10 | [boolean equals(Object another)](https://www.javatpoint.com/java-string-equals) | It checks the equality of string with the given object. |
| 11 | [boolean isEmpty()](https://www.javatpoint.com/java-string-isempty) | It checks if string is empty. |
| 12 | [String concat(String str)](https://www.javatpoint.com/java-string-concat) | It concatenates the specified string. |
| 13 | [String replace(char old, char new)](https://www.javatpoint.com/java-string-replace) | It replaces all occurrences of the specified char value. |
| 14 | [String replace(CharSequence old, CharSequence new)](https://www.javatpoint.com/java-string-replace) | It replaces all occurrences of the specified CharSequence. |
| 15 | [static String equalsIgnoreCase(String another)](https://www.javatpoint.com/java-string-equalsignorecase) | It compares another string. It doesn't check case. |
| 16 | [String[] split(String regex)](https://www.javatpoint.com/java-string-split) | It returns a split string matching regex. |
| 17 | [String[] split(String regex, int limit)](https://www.javatpoint.com/java-string-split) | It returns a split string matching regex and limit. |
| 18 | [String intern()](https://www.javatpoint.com/java-string-intern) | It returns an interned string. |
| 19 | [int indexOf(int ch)](https://www.javatpoint.com/java-string-indexof) | It returns the specified char value index. |
| 20 | [int indexOf(int ch, int fromIndex)](https://www.javatpoint.com/java-string-indexof) | It returns the specified char value index starting with given index. |
| 21 | [int indexOf(String substring)](https://www.javatpoint.com/java-string-indexof) | It returns the specified substring index. |
| 22 | [int indexOf(String substring, int fromIndex)](https://www.javatpoint.com/java-string-indexof) | It returns the specified substring index starting with given index. |
| 23 | [String toLowerCase()](https://www.javatpoint.com/java-string-tolowercase) | It returns a string in lowercase. |
| 24 | [String toLowerCase(Locale l)](https://www.javatpoint.com/java-string-tolowercase) | It returns a string in lowercase using specified locale. |
| 25 | [String toUpperCase()](https://www.javatpoint.com/java-string-touppercase) | It returns a string in uppercase. |
| 26 | [String toUpperCase(Locale l)](https://www.javatpoint.com/java-string-touppercase) | It returns a string in uppercase using specified locale. |
| 27 | [String trim()](https://www.javatpoint.com/java-string-trim) | It removes beginning and ending spaces of this string. |
| 28 | [static String valueOf(int value)](https://www.javatpoint.com/java-string-valueof) | It converts given type into string. It is an overloaded method. |

## What are Multitasking?

## - Types of Multitasking?

## - What is a Thread in Java?

## - Lifecycle of a Thread

## - What is Multithreading in Java

## - Methods of Multithreading

## What is a Java Collection?

## - What is the Java Collection Framework?

## - Java Collection Framework Hierarchy

## - Collections Interface Methods

## - Java Collections Interface

## - Java Collection Classes

## - Collection API Algorithms

## - Benefits of Java Collections

## Three Classes of Regex

## - Methods

## - Regex in Java Character Class

## - Regex in Java Quantifiers

## - Regex in Java

## - Metacharacters