Ashwini Upadhyay

OOPS
Exception Handling
Multithreading
Collections Framework
Advance Java - JDBC
Advance Java - Servelet

OOPS
Lec 1

Programming lang. are divided into 2 types :-
1) Procedure lang. → C, cobal, Pascal
2) object oriented lang. → Java, C#, Python

→ In Procedure lang. we will develop function and procedure. if we want to add more functionality then we need to develop more function there is no security.
→ If we want to develop a project using OOP lang. than we have to use classes and object.
↳ Blueprint ↳ Instance of a class

→ OOPS lang. are secure
→ the main adv. is code reusability.

OOPS principles

1) Encapsulation (Data hiding)
↳ is a used to combine our variable and method as a single entity.
↳ we can achieve Encapsulation by using classes
↳ class Demo {
// variable
// method
}

2) Abstraction (eg: ATM)
↳ means hiding unnecessary data and providing only required data.
↳ we can achieve abstraction using interface and abstract class.
ex: we will not bother about how laptop is working internally.

3) Polymorphism
↳ Exhibiting multiple behaviors based on situation is c/as Polymorphism
↳ There are two types of polymorphism
↳ overloading
↳ overriding

4) Inheritance
↳ means creating new classes based on existing ones.
↳ acquires all the properties and behaviors of a parent class.

Instance variable : 1) variable which are declare inside the class but outside the method is called instance variable.

2) when we create an object then only memory will be allocated for instance variable.

3) If we don't initialize instance variable, it will be initialize by default based on the datatype when the object is created.

4) Every object will maintain its own copy of instance variable.

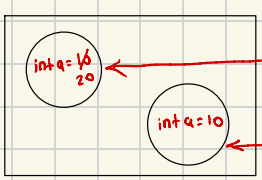
```
package Java;
public class Instancevariable {
    int a = 10; // instance
    int b; // instance
    public static void main(String[] args) {
        Instancevariable instance = new Instancevariable();
        System.out.println(instance.a); // 10
        System.out.println(instance.b); // 0
        instance.a = 100;
        instance.b = 200;
        System.out.println(instance.a); // 100
        System.out.println(instance.b); // 200
    }
}
```

Rule: In static we cannot use instance variable directly, we want to create object first.

static variable

```
package Java;
public class Staticvariable {
    static int a = 10;
    public static void main(String[] args){
        System.out.println(a); // 10
    }
}
public class B {
    public void demo(){
        System.out.println(Staticvariable.a); //10
    }
}
```

class name where static variable declared



int a = 10

InstanceVariable instance1 = new Instancevariable();

instance1.a = 20

InstanceVariable instance2 = new Instancevariable();

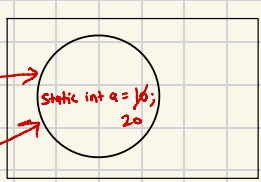
(Note: In Instance variable each obj create a different copy)

Static int a = 10;

StaticVariable obj1 = new StaticVariable

obj1.a = 20

StaticVariable obj2 = new StaticVariable



(Note: In static each obj share a same copy)

Methods

OOPS
Lec 2

→ return type is data type indicates what type of value is return by the function / method

Note: If function is not returning anything then we use 'void' in case of void no need to add return type.

→ method name: to identify and access the method we should give a suitable name for a method which called by method name.

→ method parameters: are the variable that will recieve the value that are passed into the method by the time of calling.

access modifiers ← static method ← return type → method name → method parameters

```
public static void main (String [] args) {
    //method body
}
```

```java
package Dec302023;

2 usages  new *
public class Animal {

    2 usages  new *
    public String ranveer()
    {
        System.out.println("papa papa papa papa");
        return "papa papa papa papa";
    }

    no usages  new *
    public String rashmika()
    {
        System.out.println("----------");
        return "----------";
    }
}
```

```java
package Dec302023;


new *
public class Papa {
    new *
    public static void main(String[] args) {
        Animal animal=new Animal();
        animal.ranveer();
        animal.ranveer();
    }
}
```

Animal.java

```java
package Dec302023;

public class Animal {  2 usages  amanrk2801

    public String ranveer()  1 usage  amanrk2801
    {
        //System.out.println("papa papa papa papa");
        return "papa papa papa papa";
    }

    public String ranveer(String arr)  2 usages  amanrk2801
    {

        return arr;
    }

    public int ranveer(int a)  1 usage  amanrk2801
    {

        return a;
    }

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

    public String rashmika()  1 usage  amanrk2801
    {
       // System.out.println("----------");
        return "----------";
    }
    public String rashmika(String arr) { return arr; }
}
```

Papa.java

```java
package Dec302023;


public class Papa {  amanrk2801 *
    public static void main(String[] args) {  amanrk2801 *
        Animal animal=new Animal();
        String s= animal.ranveer();
        System.out.println(s); // papa papa papa papa
        String s1 =animal.rashmika();
        System.out.println(s1); // ----------
        System.out.println("---------object with para---------");
        String r1=animal.ranveer( arr: "rashmika");
        System.out.println(r1); // rashmika
        String r2=animal.ranveer( arr: "ranveer");
        System.out.println(r2); // ranveer

        int res = animal.ranveer( a: 10); //ctrl + alt +v
        System.out.println(res); // 10
        System.out.println(animal.ranveer( a: 10, b: 40)); // 50

    }
}
```

Overriding

```
class Dog{
    public void bark(){
        System.out.println("woof ");
    }
}
class Hound extends Dog{
    public void sniff(){
        System.out.println("sniff ");
    }

    public void bark(){
        System.out.println("bowl");
    }
}
```

Same Method Name, Same parameter

Overloading

```
class Dog{
    public void bark(){
        System.out.println("woof ");
    }

    //overloading method
    public void bark(int num){
        for(int i=0; i<num; i++)
            System.out.println("woof ");
    }
}
```

Same Method Name, Different Parameter

this : current class

super : parent class

@method overloading : overloading is allowing methods with same name but different parameters with in the same class.

@method overriding : same method name + same method parameter + same return type (covariant)

Superclass: `Animal.java`

```java
package OverrideExample;

public class Animal {
    // Method to be overridden
    public void sound() {
        System.out.println("Animals make sounds");
    }

    public void eat() {
        System.out.println("Animals eat food");
    }
}
```

Subclass: `Dog.java`

```java
package OverrideExample;

// Dog is a subclass of Animal
public class Dog extends Animal {
    // Overriding the sound method
    @Override
    public void sound() {
        System.out.println("Dog barks");
    }

    // Overriding the eat method
    @Override
    public void eat() {
        System.out.println("Dog eats bones");
    }
}
```

Main Class: `OverrideDemo.java`

```java
package OverrideExample;

public class OverrideDemo {
    public static void main(String[] args) {
        // Reference of superclass with object of subclass
        Animal myDog = new Dog();

        // Calls the overridden methods in the subclass
        myDog.sound();  // Output: Dog barks
        myDog.eat();    // Output: Dog eats bones
    }
}
```

Covariant return type

```
package Dec302023;

class Gadi {  3 usages  1 inheritor  amanrk2801

    Gadi manufacture()  no usages  1 override  amanrk2801
    {
        System.out.println("Gadi lelo");
        return new Gadi();
    }
}

class Ford extends Gadi{  2 usages  amanrk2801
    @Override  no usages  amanrk2801
    Ford manufacture() {
        System.out.println("Ford ki gadi");
        return new Ford();
    }
}
```

Type of Inheritance

OOPS
Lec 3

1) single Inheritance
2) multilevel Inheritance
3) hierarchical Inheritance
4) multiple Inheritance → In Java not allowed

Single Inheritance
Class A
Class B
	public class A {
	.....
	}
	public class B **extends** A {
	.....
	}

Multi Level Inheritance Class A Class B Class C	public class A { .....} public class B **extends** A {.....} public class C **extends** B {..... }
Hierarchical Inheritance Class A Class B Class C	public class A { .....} public class B **extends** A {.....} public class C **extends** A {..... }

Multiple Inheritance
Class A
Class B
Class C
	public class A { .....}
	public class B {.....}
	public class C **extends** A,B {
	.....
	} // Java does not support mutiple Inheritance

Note: Java supports multiple inheritance through interfaces, where a class can implement multiple interfaces

(multiple inheritance example)

```java
package Dec302023;

interface A {
    default void methodA() {
        System.out.println("Method A from interface A");
    }
}
// Interface B
interface B {
    default void methodB() {
        System.out.println("Method B from interface B");
    }
}
// Class implementing both interfaces A and B
class MyClass implements A, B {
    public void myMethod() {
        System.out.println("My method in MyClass");
    }
}
public class MultipleInheritanceDemo {
    public static void main(String[] args) {
        // Creating an object of MyClass
        MyClass obj = new MyClass();
        // Calling methods from both interfaces
        obj.methodA();
        obj.methodB();
        // Calling method defined in MyClass
        obj.myMethod();
    }
}
```

```
"C:\Program Files\Java\jdk-23\bin\
Method A from interface A
Method B from interface B
My method in MyClass

Process finished with exit code 0
```

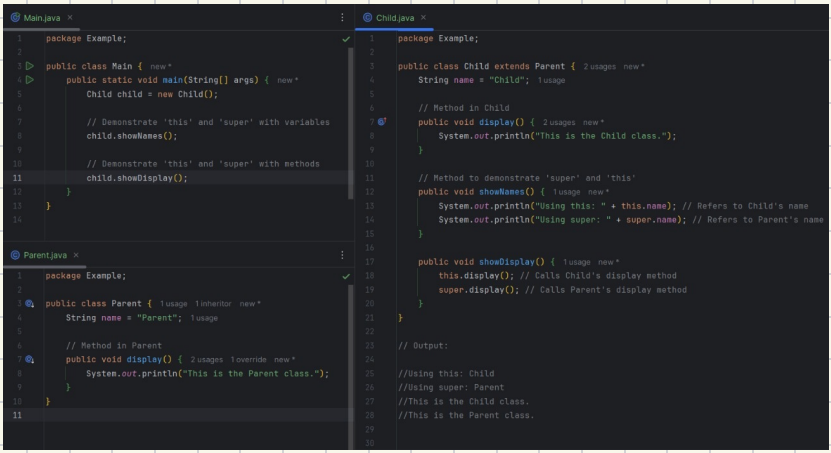
To overcome from this problem

```java
class A{
void msg(){System.out.println("Hello");}
}
class B{
void msg(){System.out.println("Welcome");}
}
class C extends A,B{//suppose if it were

 public static void main(String args[]){
  C obj=new C();
  obj.msg();//Now which msg() method would be invoked?
}
}
```

we use interfaces

Super and this Keyword



Main.java

```java
package Example;

public class Main {
    public static void main(String[] args) {
        Child child = new Child();

        // Demonstrate 'this' and 'super' with variables
        child.showNames();

        // Demonstrate 'this' and 'super' with methods
        child.showDisplay();
    }
}
```

Parent.java

```java
package Example;

public class Parent {
    String name = "Parent";

    // Method in Parent
    public void display() {
        System.out.println("This is the Parent class.");
    }
}
```

Child.java

```java
package Example;

public class Child extends Parent {
    String name = "Child";

    // Method in Child
    public void display() {
        System.out.println("This is the Child class.");
    }

    // Method to demonstrate 'super' and 'this'
    public void showNames() {
        System.out.println("Using this: " + this.name); // Refers to Child's name
        System.out.println("Using super: " + super.name); // Refers to Parent's name
    }

    public void showDisplay() {
        this.display(); // Calls Child's display method
        super.display(); // Calls Parent's display method
    }
}

// Output:

//Using this: Child
//Using super: Parent
//This is the Child class.
//This is the Parent class.
```

Constructor

Person.java

```java
public class Person {
    String name; int age; String address;
    // Default Constructor (implicitly provided if none is defined)
    public Person() {
        System.out.println("Default constructor called.");
    }
    // No-Argument Constructor (explicitly defined)
    public Person(String name) {
        this.name = name;
        System.out.println("No-argument constructor called. Name: " + name);
    }
    // Parameterized Constructor
    public Person(String name, int age, String address) {
        this.name = name;       // Assign parameter to instance variable
        this.age = age;
        this.address = address;
        System.out.println("Parameterized constructor called. Name: " + name + ", Age: " + age + ", Address: " + address);
    }
    // Overloaded Constructor (1)
    public Person(String name, int age) {
        this(name, age, address: "Unknown Address"); // Calling another constructor
        System.out.println("Overloaded constructor (1) called.");
    }
    // Overloaded Constructor (2)
    public Person(int age) {
        this( name: "Unknown Name", age); // Calling another constructor
        System.out.println("Overloaded constructor (2) called.");
    }
    // Method to display details
    public void displayDetails() { System.out.println("Name: " + name + ", Age: " + age + ", Address: " + address); }
}
```

Main.java

```java
public class Main {
    public static void main(String[] args) {
        // Using Default Constructor
        Person person1 = new Person();

        // Using No-Argument Constructor
        Person person2 = new Person( name: "Alice");

        // Using Parameterized Constructor
        Person person3 = new Person( name: "Bob", age: 25, address: "123 Street Name");

        // Using Overloaded Constructor (1)
        Person person4 = new Person( name: "Charlie", age: 30);

        // Using Overloaded Constructor (2)
        Person person5 = new Person( age: 20);

        // Displaying details for the objects
        person2.displayDetails();
        person3.displayDetails();
        person4.displayDetails();
        person5.displayDetails();
    }
}
```

NOTE :

this: Refers to the current class instance (variables or methods).
super: Refers to the parent class instance (variables or methods).
this(): Calls another constructor in the same class.
super(): Calls the constructor of the parent class.

Output:

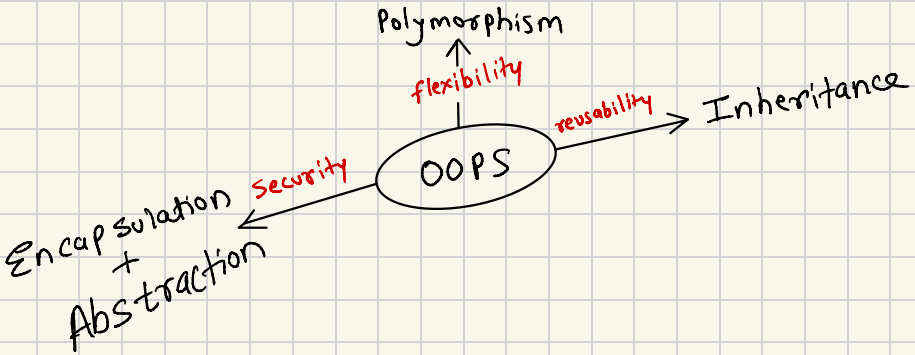
yaml — Copy code

```
Default constructor called.
No-argument constructor called. Name: Alice
Parameterized constructor called. Name: Bob, Age: 25, Address: 123 Street Name
Parameterized constructor called. Name: Charlie, Age: 30, Address: Unknown Address
Overloaded constructor (1) called.
Parameterized constructor called. Name: Unknown Name, Age: 20, Address: Unknown Address
Overloaded constructor (2) called.
Name: Alice, Age: 0, Address: null
Name: Bob, Age: 25, Address: 123 Street Name
Name: Charlie, Age: 30, Address: Unknown Address
Name: Unknown Name, Age: 20, Address: Unknown Address
```


super() and this() must be the first statement in a constructor when used, where super() calls the parent class's constructor, and this() calls another constructor within the same class, enabling proper initialization and constructor chaining.

OOPS Lec 4

Encapsulation: private String name hides the data, with access via getter and setter methods.
Abstraction: makeSound() provides a simplified interface to interact with the behavior.
Inheritance: Dog inherits from Animal.
Polymorphism: Dog overrides makeSound() to provide a specific implementation.



final keyword

- final can be used at variable, method, classes

```
final int a;
```

- variable : if we declare a variable as final then we cannot modify its value [acts like constant].
- method : if we declare method as final then we cannot override.
- class : if we declare class as final then we cannot extends this class.

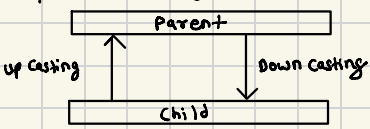
Type casting

- Type casting is the process of converting value from one type to another type.
- Note : In two same datatype type casting is not required.
- Type casting with respect to primitive datatype.
- Type casting
  - ↳ widening : converting lower type into higher datatype (int → long) → no data loss (10 → 10.0)
  - ↳ narrowing : converting higher type into lower datatype (long → int) → data loss (10.55 → 10)

Note : if we don't write any type casting then compiler will write the type casting automatically hence it is c/as implicit type casting.

Type casting with respect to reference type:

- means converting the object from one reference to another reference
- type casting can be done only btw compatible type [parent → child] [type.



Upcasting (Safe and implicit):

Assigning a subclass object to a superclass reference.

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

class Dog extends Animal {
    void makeSound() {
        System.out.println("Dog barks");
    }

    void fetch() {
        System.out.println("Dog fetches a ball");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal animal = new Dog(); // Upcasting
        animal.makeSound(); // Outputs: Dog barks (polymorphism at play)
    }
}
```

Downcasting (Explicit and requires care):

Converting a superclass reference back to a subclass reference. It's risky and needs to be type-checked.

```java
public class Main {
    public static void main(String[] args) {
        Animal animal = new Dog(); // Upcasting
        Dog dog = (Dog) animal;    // Downcasting
        dog.fetch(); // Outputs: Dog fetches a ball
    }
}
```

Key Points:

1. **Upcasting:** Always safe, as a subclass *is-a* type of superclass.
2. **Downcasting:** Needs explicit casting and may throw a `ClassCastException` if the object isn't truly of the subclass type. Always use `instanceof` to verify:

```java
if (animal instanceof Dog) {
    Dog dog = (Dog) animal;
    dog.fetch();
}
```

Abstract class and method

- class which is declare using abstract keyword is called abstract class.
- abstract class may or may not have abstract method.
- we cannot create abstract class object.
- it is used to provide abstraction.
- Abstract class it not provide 100% abstraction bcz we have concrete method inside abstract class.
- an abstract class must declare with an abstract keyword.
- it cannot be instantiated (object can not be created directly)

Abstract method

- method which is declare using abstract keyword clas abstract method.
- method that are declare without body within abstract class.
- the method body will be define by its subclasses.
- abstract method can never be final and static.

Abstract Class and Method Example

```java
// Abstract class
abstract class Animal {
    // Abstract method (no body)
    abstract void sound();

    // Regular method
    void eat() {
        System.out.println("This animal eats food.");
    }
}

// Subclass (inherits from Animal)
class Dog extends Animal {
    // Implement the abstract method
    void sound() {
        System.out.println("The dog barks: Woof!");
    }
}

// Main class
public class Main {
    public static void main(String[] args) {
        Animal myDog = new Dog();
        myDog.sound(); // Calls the implemented method in Dog
        myDog.eat();   // Calls the regular method in Animal
    }
}
```

Output:

```yaml
The dog barks: Woof!
This animal eats food.
```

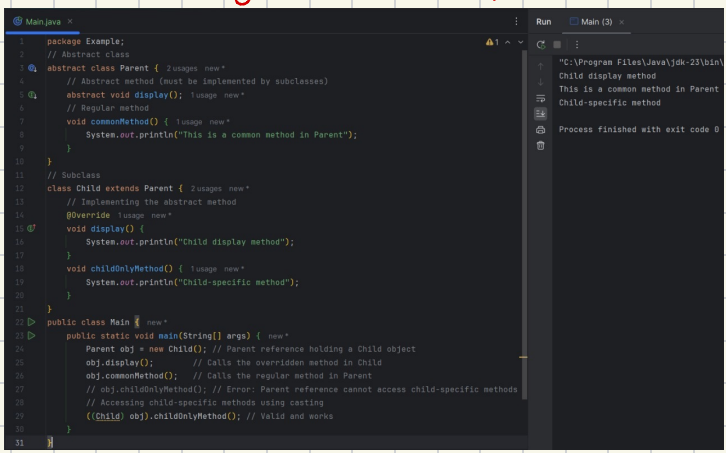
Explanation:

- The `Animal` class is abstract and contains one abstract method (`sound`) and one regular method (`eat`).
- The `Dog` class extends `Animal` and provides an implementation for the `sound` method.
- The abstract method ensures that every subclass of `Animal` must define how the `sound` method works.

OOPS Lec 5

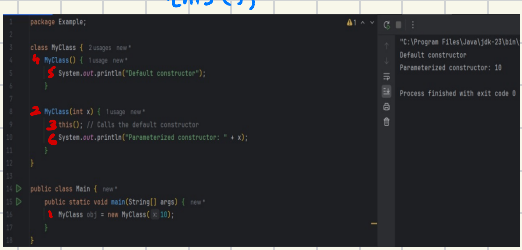
"parent reference is capable of holding child obj"

Parent obj = new Child();

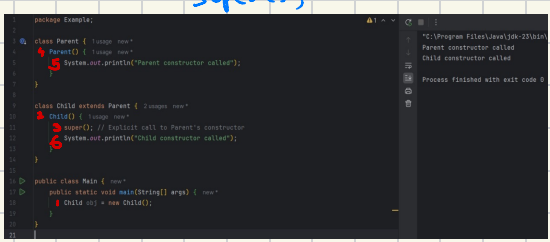


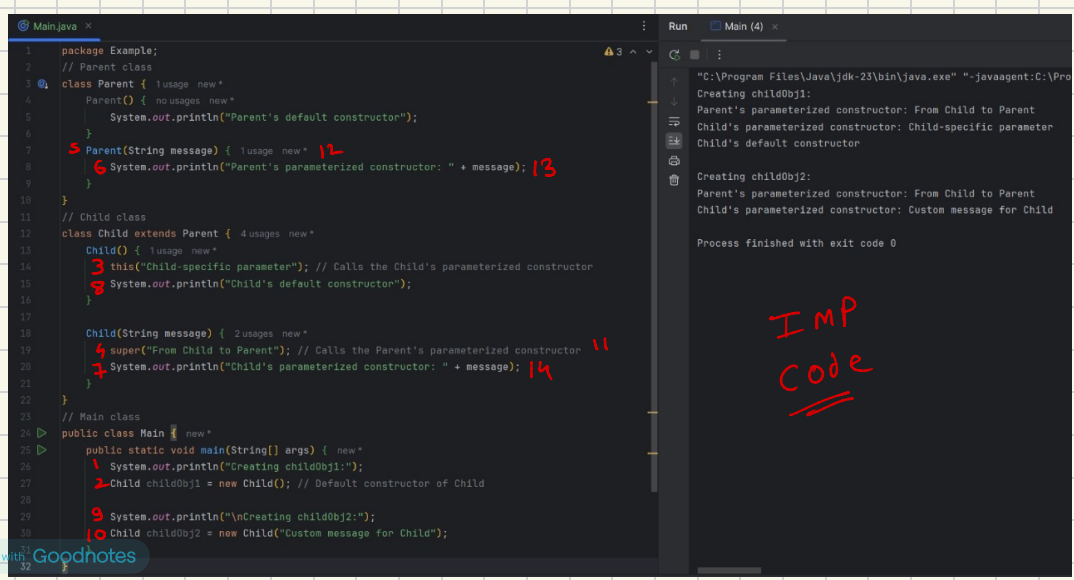
Note: constructor first line always super(); or this();

this();



super();





IMP Code

Interface

- is used to achieve 100% abstraction.
- in interface variables public, static, final by default.
- in interface method are public abstract by default.

Main.java

```java
package Example;
// Main class
public class Main {  new *
    public static void main(String[] args) {  new *
        // Call static method directly from the interface
        Animal.info();
        // Create an object of the implementing class
        Animal myDog = new Dog();
        // Access methods
        myDog.sound(); // Calls the overridden method in Dog
        myDog.eat();   // Calls the overridden default method in Dog
        // Access constant variable
        System.out.println("Animal type: " + Animal.TYPE);
    }
}
```

Dog.java

```java
package Example;
// Implement the interface in a class
public class Dog implements Animal {  1 usage  new *
    // Implement the abstract method
    @Override  1 usage  new *
    public void sound() {
        System.out.println("The dog barks: Woof!");
    }
    // Override the default method (optional)
    @Override  1 usage  new *
    public void eat() {
        System.out.println("The dog eats bones.");
    }
}
```

Animal.java

```java
package Example;

// Define an interface
public interface Animal {  4 usages  1 implementation  new *
    // Variables in interfaces are implicitly public, static, and final
    String TYPE = "Animal"; // Constant variable  2 usages

    // Abstract method (must be implemented by implementing classes)
    void sound();  1 usage  1 implementation  new *

    // Default method (optional for implementation classes to override)
    default void eat() {  1 usage  1 override  new *
        System.out.println("This " + TYPE + " eats food.");
    }

    // Static method (can be called without creating an object)
    static void info() {  1 usage  new *
        System.out.println("This is the Animal interface.");
    }
}

//Output:
//This is the Animal interface.
//The dog barks: Woof!
//The dog eats bones.
//Animal type: Animal
```

variable argument in java:

VarargsExample.java

```java
package Example;

public class VarargsExample {  new *

    // Method with variable arguments (varargs)
    public static void printNumbers(int... numbers) {  3 usages  new *
        // Loop through the numbers array and print each number
        for (int num : numbers) {
            System.out.println(num);
        }
    }

    public static void main(String[] args) {  new *
        // Calling the method with a varying number of arguments
        System.out.println("Calling with no arguments:");
        printNumbers(); // No arguments

        System.out.println("\nCalling with one argument:");
        printNumbers(5); // One argument

        System.out.println("\nCalling with multiple arguments:");
        printNumbers(1, 2, 3, 4, 5); // Multiple arguments
    }
}
```

Run — VarargsExample

```
"C:\Program Files\Java\jdk-23\bin\
Calling with no arguments:

Calling with one argument:
5

Calling with multiple arguments:
1
2
3
4
5

Process finished with exit code 0
```

Exception Handling

Lec 1

- **Exception**: An unwanted unexpected even that disturb normal flow of program is called exception.
- **Exception Handling**: In my program if I am having any code that may raise a exception is called risky code, this should be placed in try catch block. (note: exception show in runtime)

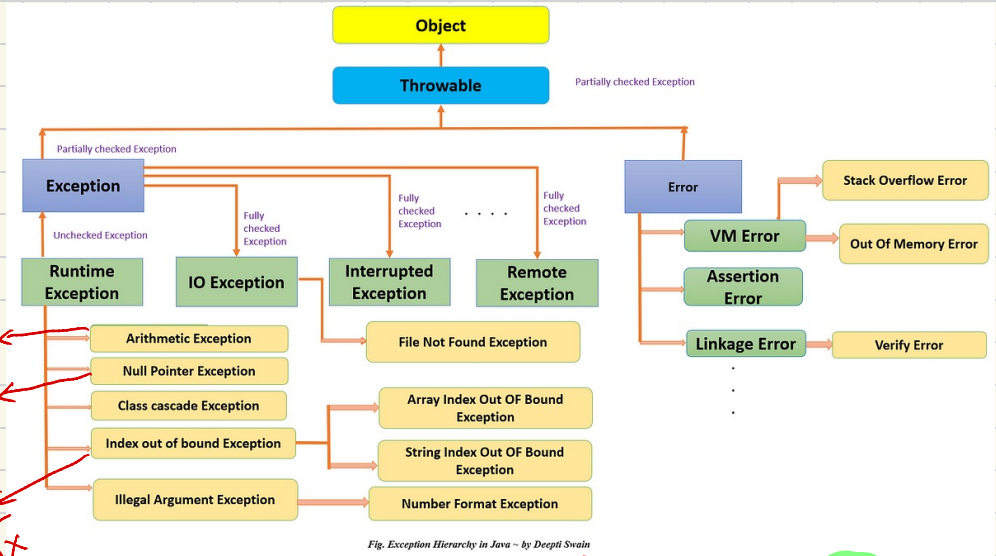


Fig. Exception Hierarchy in Java ~ by Deepti Swain

Note: If developer not Handle Exception then JVM use default Exception Handler

```java
package Example;

public class ExceptionHandlingExample {
    public static void main(String[] args) {
        try {
            // Code that may throw an exception
            int result = divide(10, 0); // Division by zero, which will cause an ArithmeticEx
            System.out.println("Result: " + result);
        } catch (ArithmeticException e) {
            // Handling ArithmeticException
            System.out.println("Error: Cannot divide by zero.");
        } finally {
            // This block is always executed
            System.out.println("This block is always executed.");
        }

        System.out.println("Program continues after handling the exception.");
    }

    // Method that divides two numbers
    public static int divide(int a, int b) {
        return a / b; // This will throw ArithmeticException if b is zero
    }
}
```

```
"C:\Program Files\Java\jdk-23\bin\java.exe" "-java
Error: Cannot divide by zero.
This block is always executed.
Program continues after handling the exception.

Process finished with exit code 0
```

or e.printStackTrace();

e.printStackTrace();
SOP(e.toString()); extends
SOP(e.getMessage());

Exception description Location

Arithmetic Exception

final, finally, finalize

```java
public class FinalFinallyFinalizeExample {
    // final variable (constant)
    final int MAX_VALUE = 100;

    // Method that demonstrates 'final' with method parameter
    public void displayValue(final int value) {
        // value cannot be reassigned because it's declared as 'final'
        System.out.println("Value: " + value);
    }

    // Method that demonstrates 'finally' block
    public void testFinally() {
        try {
            // Code that might throw an exception
            System.out.println("In try block");
            int result = 10 / 0;  // This will cause ArithmeticException
        } catch (ArithmeticException e) {
            System.out.println("Exception caught: " + e.getMessage());
        } finally {
            // This block will always execute
            System.out.println("This is the finally block. It always executes.");
        }
    }
    // Method that demonstrates 'finalize' method (called by garbage collector)
    @Override
    protected void finalize() {
        // Cleanup code before object is garbage collected
        System.out.println("Finalize method called. Object is being garbage collected.");
    }

    public static void main(String[] args) {
        FinalFinallyFinalizeExample example = new FinalFinallyFinalizeExample();

        // Demonstrating 'final' with variable
        System.out.println("MAX_VALUE: " + example.MAX_VALUE);

        // Demonstrating 'final' with method parameter
        example.displayValue(50);

        // Demonstrating 'finally' block
        example.testFinally();

        // Demonstrating 'finalize' method
        example = null;  // Nullifying reference to make the object eligible for garbage collection
        System.gc();     // Requesting garbage collection (may not run immediately)

        // Adding delay to ensure finalize() has time to execute
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

```
MAX_VALUE: 100
Value: 50
In try block
Exception caught: / by zero
This is the finally block. It always executes.
Finalize method called. Object is being garbage collected.
```

throw: Used to explicitly throw an exception within a method.

Example: throw new Exception("An error occurred");

throws: Used in a method signature to declare that the method may throw one or more exceptions.

Example: public void myMethod() throws Exception {}

Throws

ThrowsExample.java

```java
package Example;
public class ThrowsExample {

    // Method that declares it might throw an exception
    public static void readFile(String filename) throws Exception {
        if (filename == null) {
            // Throwing an exception manually
            throw new Exception("Filename cannot be null");
        }
        System.out.println("Reading file: " + filename);
        // Simulating reading a file
    }

    // Method that calls readFile and handles the exception
    public static void processFile(String filename) {
        try {
            readFile(filename); // Calling the method that throws an exception
        } catch (Exception e) {
            System.out.println("Caught exception: " + e.getMessage());
        }
    }

    public static void main(String[] args) {
        // Calling processFile with a valid filename
        System.out.println("With valid filename:");
        processFile( filename: "myfile.txt");

        // Calling processFile with a null filename to trigger an exception
        System.out.println("\nWith null filename:");
        processFile( filename: null);
    }
}
```

Run — ThrowsExample

```
"C:\Program Files\Java\jdk-23\bin\java.exe"
With valid filename:
Reading file: myfile.txt

With null filename:
Caught exception: Filename cannot be null

Process finished with exit code 0
```

Note: Throws delicate task to his upper class to handle exception if upper class not handle it so handled by JVM itself.

How to create custom Exception

checked Exception

CustomExceptionExample.java

```java
package Example;
/*
* A custom exception that requires the calling code to handle it using try-catch or throws.
* */
// Custom checked exception
class InvalidAgeException extends Exception {  3 usages  new *
    // Constructor that accepts a message
    public InvalidAgeException(String message) {  1 usage  new *
        super(message);
    }
}

public class CustomExceptionExample {  new *

    // Method that throws the custom exception if age is invalid
    public static void validateAge(int age) throws InvalidAgeException {  1 usage  new *
        if (age < 18) {
            throw new InvalidAgeException("Age must be 18 or older.");
        }
        System.out.println("Age is valid.");
    }

    public static void main(String[] args) {  new *
        try {
            // Calling the method with an invalid age
            validateAge(15);
        } catch (InvalidAgeException e) {
            // Handling the custom exception
            System.out.println("Caught custom exception: " + e.getMessage());
        }
    }
}
```

Run — CustomExceptionExample (1)

```
"C:\Program Files\Java\jdk-23\bin\java.exe" "-jave
Caught custom exception: Age must be 18 or older.

Process finished with exit code 0
```

unchecked Exception

CustomUncheckedExceptionExample.java

```java
package Example;
/*
* A custom exception that does not require
* explicit handling with try-catch or throws because it extends RuntimeException.
* */
// Custom unchecked exception
class InvalidAgeException extends RuntimeException {  2 usages  new *
    // Constructor that accepts a message
    public InvalidAgeException(String message) {  1 usage  new *
        super(message);
    }
}
public class CustomUncheckedExceptionExample {  new *

    // Method that throws the custom unchecked exception
    public static void validateAge(int age) {  1 usage  new *
        if (age < 18) {
            throw new InvalidAgeException("Age must be 18 or older.");
        }
        System.out.println("Age is valid.");
    }

    public static void main(String[] args) {  new *
        try {
            // Calling the method with an invalid age
            validateAge(15);
        } catch (InvalidAgeException e) {
            // Handling the custom unchecked exception
            System.out.println("Caught custom exception: " + e.getMessage());
        }
    }
}
```

Run — CustomUncheckedExceptionExample (1)

```
"C:\Program Files\Java\jdk-23\bin\java.exe" "-java
Caught custom exception: Age must be 18 or older.

Process finished with exit code 0
```

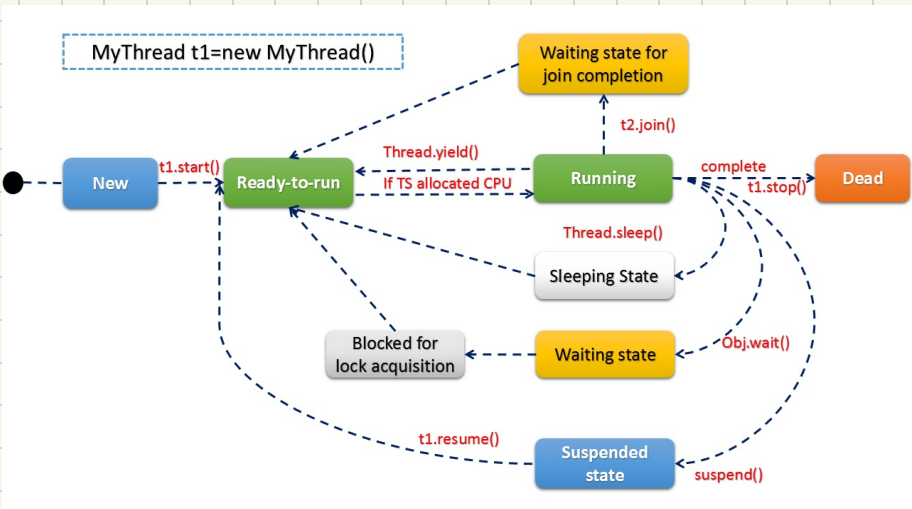
multithreading In Java

Lec 1

multitasking

- Process Based
- Thread Based

- **Process-based**: Think of a **web browser** (like Chrome) as a process. Each open tab in the browser is typically a separate process, running independently with its own memory to avoid one tab's crash affecting others.
- **Thread-based**: Within a web browser tab, multiple threads might be running simultaneously—for example, one thread handling user input, another rendering the page, and another managing background network requests.



Life cycle of Thread

1. New: A thread is created but hasn't started yet (t1.start() not called).
2. Ready-to-Run: Thread is ready for execution but waiting for CPU allocation by the Thread Scheduler. (TS)
3. Running: Thread is actively executing its run() method.
4. Thread.yield(): Suggests to the scheduler to allow other threads to execute. (Current execution partially pause)
5. Sleeping State: Thread is paused temporarily using Thread.sleep() for a specified time.
6. Waiting State: Thread waits indefinitely until another thread signals it (e.g., using obj.wait()).
7. Blocked for Lock Acquisition: Thread is waiting to acquire a lock on a synchronized resource.
8. Suspended State: Thread is paused using suspend() (deprecated due to deadlock risks).
9. Waiting State for Join Completion: Thread waits for another thread to finish execution (t2.join()).
10. Dead: Thread has completed its execution and cannot be restarted.

Thread Class

The `Thread` class is used to create a thread by extending it. You override the `run()` method to define the thread's task.

Example:

```
class MyThread extends Thread {
    @Override
    public void run() {
        System.out.println("Thread is running using Thread class.");
    }
}

public class ThreadExample {
    public static void main(String[] args) {
        MyThread thread = new MyThread(); → Instantiation of Thread
        thread.start(); // Starts the thread
    }
}
```

Runnable Interface (Best way)

The `Runnable` interface is implemented by a class to define a thread task, allowing the class to extend other classes if needed.

Example:

```
class MyRunnable implements Runnable {
    @Override
    public void run() {
        System.out.println("Thread is running using Runnable interface.");
    }
}

public class RunnableExample {
    public static void main(String[] args) {
        MyRunnable myRunnable = new MyRunnable();
        Thread thread = new Thread(myRunnable);
        thread.start(); // Starts the thread
    }
}
```

Key Difference:

- **Thread class**: Extending the `Thread` class means your class cannot extend any other class.
- **Runnable interface**: Provides more flexibility, as your class can implement `Runnable` and still extend another class.

Real-life analogy:

- **Thread class**: Like a dedicated worker who directly performs tasks.
- **Runnable interface**: Like a manager giving tasks to a worker (the `Thread`).

list of multithreading-related jargons in Java with concise definitions:

1. Thread: A lightweight sub-process that performs tasks concurrently within a program.
2. Runnable: An interface used to define a thread's task by implementing the run() method.
3. Thread Life Cycle: The states of a thread: New, Runnable, Running, Blocked/Waiting, and Terminated.
4. Daemon Thread: A background thread that runs until all user threads finish execution (e.g., garbage collector).
5. Thread Priority: Determines the execution preference of threads (range: MIN_PRIORITY to MAX_PRIORITY).
6. Synchronization: A mechanism to control thread access to shared resources, preventing data inconsistency.
7. Synchronized Block: A smaller critical section in code to reduce the scope of synchronization.
8. Lock: An advanced synchronization mechanism that provides explicit lock control for threads.
9. ReentrantLock: A type of lock allowing the same thread to acquire it multiple times.
10. Thread Pool: A pool of pre-created threads used to execute tasks, improving resource management.
11. Executor Framework: A Java API for managing and controlling thread pools efficiently.
12. Future: Represents the result of an asynchronous computation.
13. Callable: A task that returns a result and can throw exceptions, used with Future.
14. Volatile: A keyword ensuring changes to a variable are visible to all threads immediately.
15. ThreadLocal: Provides each thread with its own isolated copy of a variable.
16. Deadlock: A situation where two or more threads are blocked indefinitely, waiting for each other's resources.
17. Race Condition: A problem when multiple threads access shared resources without proper synchronization.
18. Wait(): A method to pause a thread until another thread notifies it.
19. Notify() / NotifyAll(): Methods to wake up waiting threads.
20. Fork/Join Framework: Used for parallel task execution by dividing tasks into smaller subtasks.
21. Semaphore: A concurrency control mechanism for limiting thread access to resources.
22. CyclicBarrier: A synchronization aid that allows threads to wait until a common barrier point is reached.
23. CountDownLatch: A tool to make threads wait until a set of operations are completed.
24. Atomic Variables: Variables that ensure atomic (thread-safe) updates without explicit synchronization.
25. Interrupt(): A method to signal a thread to stop or alter its behavior.
26. Join(): Makes a thread wait for another thread to finish execution.
27. Yield(): A hint to the thread scheduler to temporarily pause the current thread.
28. Concurrency: The ability of a system to execute multiple threads simultaneously.
29. Parallelism: True simultaneous execution of tasks on multiple processors.
30. Thread-safe: Code or data structure that operates correctly when accessed by multiple threads concurrently.

Thread Scheduler in Java

The Thread Scheduler is part of the Java Virtual Machine (JVM) responsible for managing the execution of multiple threads based on their priority and state.

Key Points about the Thread Scheduler:

1. Purpose: It decides which thread to run next from the pool of threads in the Runnable state.
2. Algorithm: Implementation-dependent (e.g., Time Slicing or Preemptive Scheduling).
   - Time Slicing: Each thread gets a fixed amount of CPU time.
   - Preemptive Scheduling: Higher-priority threads preempt lower-priority ones.
3. Non-deterministic: The order of thread execution is unpredictable and cannot be guaranteed.
4. Priority: Influences scheduling but does not guarantee execution order (Thread.MIN_PRIORITY = 1, Thread.MAX_PRIORITY = 10).

Example Demonstration:

```
class MyThread extends Thread {
    public MyThread(String name) {
        super(name);
    }

    @Override
    public void run() {
        System.out.println(getName() + " is running.");
    }
}

public class ThreadSchedulerExample {
    public static void main(String[] args) {
        MyThread t1 = new MyThread("Thread 1");
        MyThread t2 = new MyThread("Thread 2");
        MyThread t3 = new MyThread("Thread 3");

        t1.setPriority(Thread.MIN_PRIORITY); // Priority = 1
        t2.setPriority(Thread.NORM_PRIORITY); // Priority = 5
        t3.setPriority(Thread.MAX_PRIORITY); // Priority = 10

        t1.start();
        t2.start();
        t3.start();
    } System.out.println(t1.getPriority);
}
```

run method → for assign job

start → new thread created

Output:

The output may vary each time you run the program, depending on how the thread scheduler manages threads. For example:

```
Thread 3 is running.
Thread 2 is running.
Thread 1 is running.
```

Note:

The actual behavior of the thread scheduler is platform-dependent (e.g., JVM implementation and operating system). Priority hints are not always respected on all platforms.

1. Get Thread Name (getName())

- Returns the name of the thread.

Example:

```
Thread t1 = new Thread(() -> System.out.println("Running thread"));
System.out.println("Thread Name: " + t1.getName());
```

2. Set Thread Name (setName(String name))

- Sets a custom name for the thread.

Example:

```
Thread t1 = new Thread(() -> System.out.println("Running thread"));
t1.setName("CustomThread");
System.out.println("Updated Thread Name: " + t1.getName());
```

Example with Both Methods

```
class MyThread extends Thread {
    @Override
    public void run() {
        System.out.println("Thread running with name: " + getName());
    }
}

public class ThreadNameExample {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        t1.setName("Worker-1");
        t1.start(); // Start the thread
    }
}
```

Output:

```
Thread running with name: Worker-1
```

Lec 2

In Java, several methods can be used to **prevent or pause thread execution**, depending on the situation:

1. `Thread.sleep(milliseconds)`:
   - Temporarily pauses the thread execution for the specified time.
   - Example:

```
Thread.sleep(1000); // Pauses for 1 second
```

2. `Thread.yield()`:
   - Hints to the Thread Scheduler to pause the current thread and allow other threads to execute.
   - Example:

```
Thread.yield();
```

3. `wait()` (on an object):
   - Makes the thread wait indefinitely until it is notified by another thread.
   - Example:

```
synchronized (lock) {
    lock.wait();
}
```

4. `join()`:
   - Prevents the current thread from proceeding until the specified thread finishes execution.
   - Example:

```
t.join(); // Waits for thread `t` to complete
```

5. **Blocked State (e.g., synchronized block)**:
   - Prevents execution if the thread is waiting to acquire a lock.
   - Example:

```
synchronized (lock) {
    // Other thread blocks until lock is released
}
```

6. **Deprecated Methods (Avoid Using)**:
   - `suspend()` - Suspends the thread indefinitely until `resume()` is called (can cause deadlock).
   - `stop()` - Abruptly stops the thread, leading to potential resource issues.

Explanation of synchronization, synchronized block, synchronized method, thread-safe, and thread-unsafe concepts in Java:

1. Synchronization

A mechanism to prevent multiple threads from accessing shared resources simultaneously, ensuring data consistency.

Example:

```
synchronized(lock) {
    // Critical section (only one thread can execute at a time)
}
```

2. Synchronized Block

A smaller, more fine-grained critical section within a method to minimize the performance impact of synchronization.

Example:

```
public void update() {
    synchronized(this) {  // Synchronizing only the critical section
        counter++;
    }
}
```

3. Synchronized Method

The entire method is synchronized, allowing only one thread to execute it at a time for the given object.

Example:

```
public synchronized void increment() {
    counter++;
}
```

4. Thread-Safe

Code is thread-safe if it functions correctly when accessed by multiple threads concurrently, usually achieved through synchronization or atomic operations.

Example:

```
public synchronized void deposit(int amount) {
    balance += amount;  // Thread-safe as it's synchronized
}
```

5. Thread-Unsafe

Code is thread-unsafe if multiple threads accessing shared resources can cause data inconsistency or unexpected behavior.

Example:

```
public void deposit(int amount) {
    balance += amount;  // Not thread-safe; multiple threads can corrupt data
}
```

Comparison of Synchronized Block vs Method:

Feature	Synchronized Method	Synchronized Block
Scope of Lock	Entire method	Specific block of code
Performance	Lower (locks entire method)	Higher (locks only critical section)
Flexibility	Less (locks this or class object)	More (custom objects can be locked)

Inter-thread communication is the mechanism by which threads can communicate and coordinate their execution, typically using methods like wait(), notify(), and notifyAll() from the Object class.

Key Methods for Inter-thread Communication:

1. wait():
   - Causes the current thread to wait until another thread calls notify() or notifyAll() on the same object.
   - The thread releases the lock on the object and enters the waiting state.
2. notify():
   - Wakes up one of the threads that are waiting on the same object's monitor.
3. notifyAll():
   - Wakes up all threads waiting on the same object's monitor.

Example: Producer-Consumer Problem

This example demonstrates how threads communicate to manage shared resources.

```
class SharedResource {
    private int data;
    private boolean available = false;

    public synchronized void produce(int value) {
        while (available) {  // Wait until the resource is consumed
            try {
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        data = value;
        available = true;
        System.out.println("Produced: " + value);
        notify(); // Notify consumer thread
    }

    public synchronized void consume() {
        while (!available) {  // Wait until the resource is produced
            try {
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        System.out.println("Consumed: " + data);
        available = false;
        notify(); // Notify producer thread
    }
}

public class InterThreadCommunicationExample {
    public static void main(String[] args) {
        SharedResource resource = new SharedResource();

        // Producer thread
        Thread producer = new Thread(() -> {
            for (int i = 1; i <= 5; i++) {
                resource.produce(i);
            }
        });

        // Consumer thread
        Thread consumer = new Thread(() -> {
            for (int i = 1; i <= 5; i++) {
                resource.consume();
            }
        });

        producer.start();
        consumer.start();
    }
}
```

Output

```
Produced: 1
Consumed: 1
Produced: 2
Consumed: 2
Produced: 3
Consumed: 3
Produced: 4
Consumed: 4
Produced: 5
Consumed: 5
```

modifiers

lec 3

Modifier	Used With	Purpose
`public`	Classes, Methods, Variables	Makes the member accessible everywhere.
`private`	Methods, Variables, nested class	Restricts access to within the same class.
`protected`	Methods, Variables, nested class	Accessible within the same package and by subclasses.
`default`	Methods, Variables	Accessible only within the same package (no keyword needed).
`final`	Classes, Methods, Variables	Prevents inheritance, overriding, or reassignment.
`abstract`	Classes, Methods	Declares something incomplete (must be implemented by subclasses).
`static`	Variables, Methods	Belongs to the class rather than an instance.
`synchronized`	Methods	Restricts execution to one thread at a time.
`transient`	Variables	Excludes a variable from serialization.
`volatile`	Variables	Ensures changes to variables are visible to all threads.

→ Anywhere

→ limited to class

→ class + subclass (same package)

→ limited to package

Public static void main (String[] args)

- Public → Access from anywhere
- static → without creating class object we can call
- void → no return
- main → method name
- (String[] args) → multi argument array (when we want to pass array at runtime)

Modifier	Class	Package	Subclass	World
public	✓	✓	✓	✓
protected	✓	✓	✓	✗
default	✓	✓	✗	✗
private	✓	✗	✗	✗

Explanation of each key component in the Java Collection Framework:

Lec 1

1. Interfaces:

1. Collection: Root interface representing a group of objects.
2. List: Ordered collection allowing duplicate elements (e.g., ArrayList, LinkedList).
3. Set: Unordered collection that does not allow duplicates (e.g., HashSet, TreeSet).
4. Queue: Collection designed for holding elements prior to processing, often FIFO (e.g., PriorityQueue).
5. Deque: Double-ended queue that allows insertion/removal from both ends (e.g., ArrayDeque).
6. Map: Stores key-value pairs, keys must be unique (e.g., HashMap, TreeMap).
7. SortedSet: A Set with elements sorted in natural order (e.g., TreeSet).
8. SortedMap: A Map with keys sorted in natural order (e.g., TreeMap).
9. NavigableSet: Extends SortedSet with methods for navigation (e.g., ceiling, floor).
10. NavigableMap: Extends SortedMap with navigation methods (e.g., descendingMap).

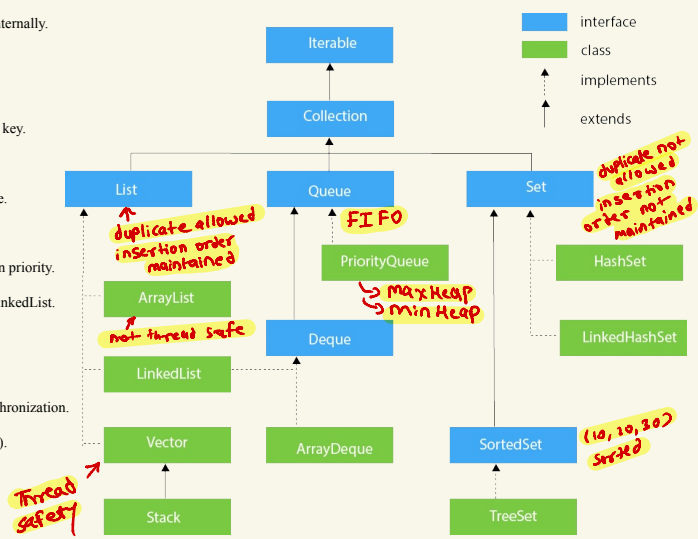
Iterable Interface

The Iterable interface is the root interface for all the collection classes. The Collection interface extends the Iterable interface and therefore all the subclasses of Collection interface also implement the Iterable interface.

The **Collection in Java** is a framework that provides an architecture to store and manipulate the group of objects.

2. Classes:

1. ArrayList: Resizable array implementation of List, fast for random access.
2. LinkedList: Doubly linked list implementation of List and Deque.
3. Vector: Legacy synchronized resizable array.
4. Stack: Last-in, first-out (LIFO) stack extending Vector.
5. HashSet: Unordered collection with no duplicates, uses a HashMap internally.
6. LinkedHashSet: Maintains insertion order in a Set.
7. TreeSet: Sorted Set implementation using a red-black tree.
8. HashMap: Unordered key-value pair implementation, allows one null key.
9. LinkedHashMap: Maintains insertion order in a HashMap.
10. TreeMap: Sorted key-value pair implementation using a red-black tree.
11. Hashtable: Legacy thread-safe key-value pair implementation.
12. PriorityQueue: Implements a min-heap for ordering elements based on priority.
13. ArrayDeque: Resizable array implementation of Deque, faster than LinkedList.

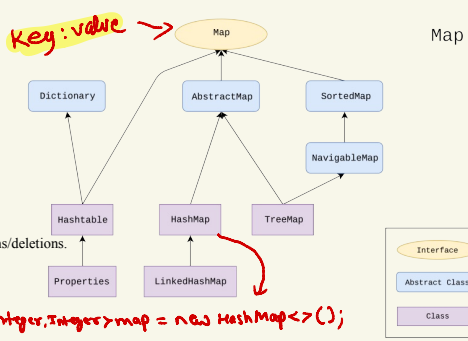


3. Utility Classes:

1. Collections: Utility class for algorithms like sorting, searching, and synchronization.
2. Arrays: Utility class for working with arrays (e.g., sorting, binary search).

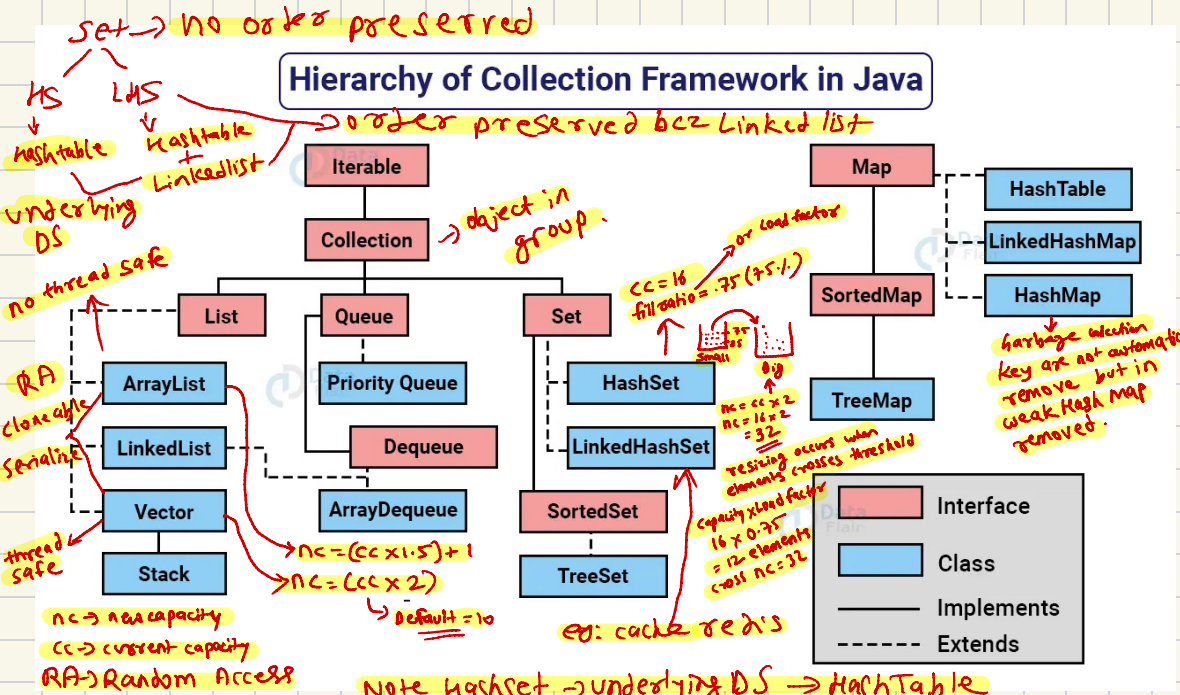
4. Key Concepts:

1. Iterator: Provides a way to traverse elements in a collection sequentially.
2. ListIterator: Bi-directional iterator for List.
3. Enumeration: Legacy iterator for Vector and Hashtable.
4. Comparable: Interface for natural ordering using compareTo().
5. Comparator: Defines custom sorting logic using compare().



5. Differences:

1. ArrayList vs LinkedList: ArrayList is faster for random access; LinkedList is better for frequent insertions/deletions.
2. HashSet vs TreeSet: HashSet is faster (no sorting), TreeSet maintains sorted order.
3. HashMap vs TreeMap: HashMap is faster (unordered), TreeMap maintains sorted keys.
4. Queue vs Deque: Queue follows FIFO, Deque allows operations at both ends.



`Iterable` is the root interface of the Java Collection Framework that provides a way to **traverse elements sequentially** using an **iterator**.

1. `Collection`: A root **interface** in the Java Collection Framework that represents a group of objects (e.g., `List`, `Set`, `Queue`).
2. `Collections`: A **utility class** in `java.util` that provides static methods for operating on collections (e.g., `sort()`, `reverse()`, `synchronizedList()`).

1. `List`: Use when you need an ordered collection with duplicates and index-based access.
2. `Set`: Use when you need a collection with unique elements only.
3. `Queue`: Use when you need elements processed in FIFO (First-In-First-Out) order.
4. `Deque`: Use when you need insertion/removal from both ends of a queue.
5. `Map`: Use when you need to store key-value pairs with unique keys.
6. `SortedSet`: Use when you need a `Set` with elements in sorted order.
7. `SortedMap`: Use when you need a `Map` with keys in sorted order.

Arraylist internals

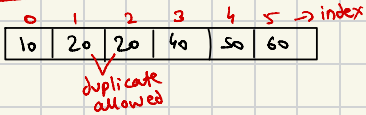
Lec 2

1) RandomAccess

JVM —(gives power)→ RandomAccess (interface)

Marker Interface
↳ no method
↳ no variables

ArrayList implement RandomAccess



- index
- for
- for each

} O(1) Search element

list.get(index)

2) Cloneable → class A implements Cloneable

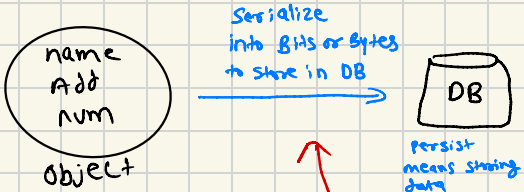
(Marker Interface)

↳ gives same object clone

3) Serializable → class student {

(Marker Interface)

string name
string Add
int num
}



Iterator (One-liner Definition):

Iterator is an interface in Java used to traverse elements sequentially in a collection, with the ability to remove elements during iteration.

Key Methods:

1. hasNext(): Returns true if there are more elements to iterate.
2. next(): Returns the next element in the iteration.
3. remove(): Removes the last element returned by next() (optional).

When to Use:

Use Iterator when you need sequential traversal of a collection without exposing its internal structure.

Example:

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

public class IteratorExample {
    public static void main(String[] args) {
        ArrayList<String> names = new ArrayList<>();
        names.add("Alice");
        names.add("Bob");
        names.add("Charlie");

        // Using Iterator to traverse the collection
        Iterator<String> iterator = names.iterator();
        while (iterator.hasNext()) {
            String name = iterator.next();
            System.out.println(name); // Outputs: Alice, Bob, Charlie

            // Removing "Bob" during iteration
            if (name.equals("Bob")) {
                iterator.remove();
            }
        }

        System.out.println("After removal: " + names); // Outputs: [Alice, Charlie]
    }
}
```

Note: When we need to send or transport data over a network, which only understands bits and bytes, we serialize the data into a binary format (bits and bytes) before sending it, such as to a database (DB)

(only read) (vector, dictionary)

- **Enumerator**: An interface used to iterate over a collection, but it is considered outdated and replaced by `Iterator` in modern Java.
- **Iterator**: An interface that provides a way to iterate over a collection and allows removal of elements during traversal. (all collection)
- **ListIterator**: A specialized iterator for `List` collections that allows bidirectional traversal and modification of the list during iteration. (only for list) (read, write & update)

(read, write)

Hashmap Internals

Lec 3

map.put("one", 1)
map.put("two", 2)
map.put("three", 3)

Hashmap default size 16

→ pass key here "one"

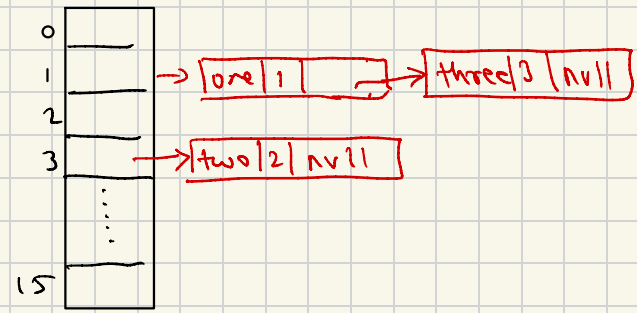
→ hashcode(-){

we get 347937 % 16 ⇒ 0–15

suppose we get '1'

}

suppose for "three" we get same hashcode.



→ equals(-){ (content compare)

equals takes help of hashcode and get hashcode of "one" 347937 if equals get the value

}

one.equals(one) = true ✓

map.get("one")
map.get("two")
map.get("three")
map.get(null, "six")

Note: in this case we get same hashcode of "one" 347937 this is known as collision

one.equals("three") = false ; three.equals("three") = true

Note: in case of null, null value store at index "0".

Hash before any collection uses underlying Data Structure "Hash Table"

HashSet → | 30 | 20 | 10 | 50 | 40 |

internally uses

Hashmap
- ↳ no order preserved
- ↳ duplicate not allowed
- ↳ no indexing

value	PR
30	PR
20	PR
10	PR
50	PR
40	PR

→ present

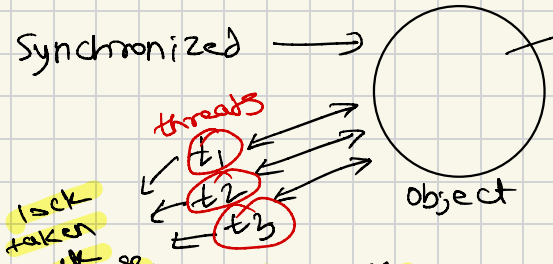
```
hashset.add(30)
if(PR == null)
{
map.put(10, PR);
return true;
}
else
{
return false;
}
```

How make arraylist and HashMAP threadsafe?

Collection.SynchronizedMap(map);

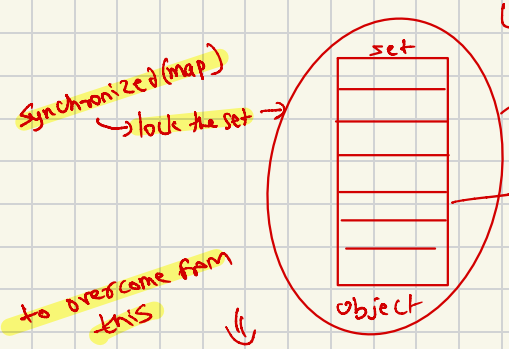
Concurrent Collection: Concurrent Hashmap
CopyOnHashSet
CopyOnArrayList

(bucket level lock / Segment Lock)



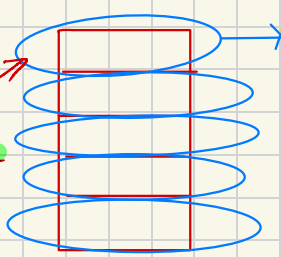
Collection.synchronizedMap(map);

↳ to overcome this problem Java develop concurrent collection.
- ↳ concurrentHashMAP
- ↳ copyonset
- ↳ copyonArrayList



to overcome from this

Bucket lock or segment lock



lock on every segment

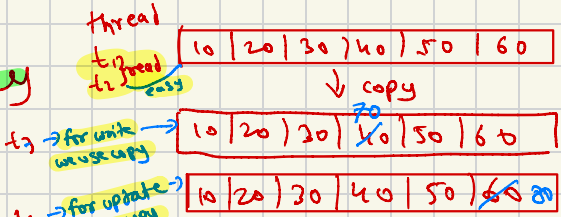
for read operation "no lock required" कोतना भी करो।

if one thread is using this then will be no write/update जो करे thread release नहीं करेगा तो

write/update

↳ bcz to provide data consistency

To make Arraylist thread safe
Copy on Arraylist



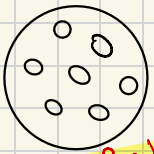
to make highly available

eventual consistency

Note: If we write new value or update new value after some time JVM sync that copy with original arraylist.

Java 8

Collection



group of object

Stream

Youtube Stream

live Stream

comes in bit and bytes over the network.

Java 8 features

1) functional Interface : Interface with "Single Abstract method"

in interface by default all methods are Abstract

```
@FunctionalInterface
interface Ankit
{
    public void m1();
    default void print();
    {
      sop("print");
    }
    static void display()
    {
      Sop("static");
    }
}
```

only one Abstract method allowed. but we can put default and static methods with that.

default method "override"

static method "no override"

no need of obj creation direct call. only access by Ankit.display no other can access this.

implementation होना है तो दो वरना तो जरूरत ही।

```
Public class FunctionalInterfaceDemo implements Ankit{
  public static void main(String[] args){
      Ankit.display(); // bcz this is static
  }
  @Override
  public void m1(){

  }
  @Override
  public void print(){
    Ankit.super.print()
  }
}
```

Lamda

```
Ankit A = () -> {
    SOP("main hu Ankit");
}
A.m1();
```

Functional Interface compliments Lamda Expression

FI → one abstract method

LE → we can implement by Lamda Exp.

Lamda → Anonymous function
↳ no name function

Predicate

```
psvm(String[] args) {
    Predicate<String> p = (String s) -> s.length() > 0;
    boolean a = p.test("Genei");
    SOP(a); // true
}
```

reference variable

implementation

- predicate.test
- predicate.and
- predicate.or
- Predicate.negate

Function is same a Predicate but can retrn any value.

```
psvm(String[] args) {
    Function<String, Integer> f = (String s) -> s.length();
    SOP(f.apply("Genie"));
}
```

Stream

Stream
↳ config → filter, map (methods) → true/false; we can change obj value
↳ processing

```
psvm(String[] args){
    List<Integer> list = new ArrayList<>();
    list.add(10);
    list.add(400);
    list.add(5);
List<Integer> collect = list.stream().filter(i->i>100).collect(collectors.toList());
    SOP(collect); // [400]
    list.stream().filter(i->i>100).forEach(System.out::println); // 400
list.stream().map(i->i+10)
    .forEach(System.out::println); // 20
}                                  // 410
                                   // 15
```

↳ colon method reference

```
Optional<Integer> max = list.stream().max((o1,o2)->o1.compareTo(o2));
SOP(max); // optional [410]
```

When we want to compare string

```
List<Student> collect1 = listOfStudents.stream()
        • sorted(Comparator.comparingInt(Student::getId))
        • collect(Collectors.toList());
    System.out.println(collect1)
```

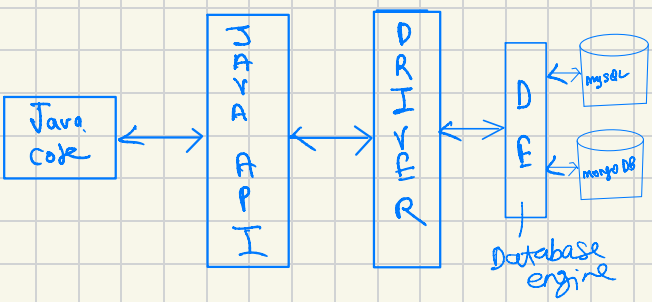
output
[Student{name='Aman', id=1, marks=100}, Student{name='z', id=2, marks=100}]

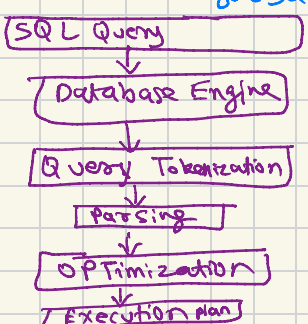
Advance JAVA - JDBC

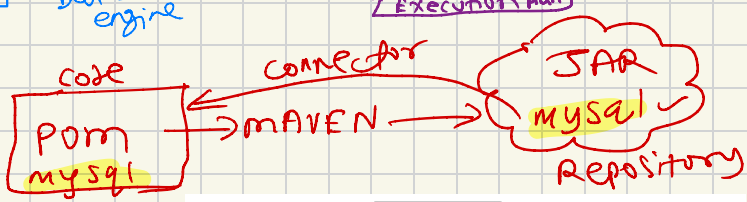
Lec 1

Storage Area
- Temporary Storage Area (JAVA object, Buffer) RAM (volatile)
- Permanent Storage Area (Hard disk, SSD)
  - File System
  - DBMS
    - RDBMS (Table) — MySQL
    - OODBMS (object) — mongo DB
    - ORDBMS (Table + object) — PostgreSQL
  - Data warehouse (Data Analyst)

Store image we use BLOB CLOB in DB





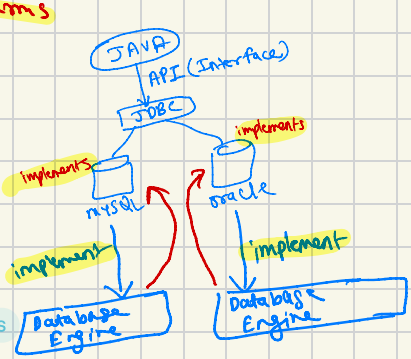


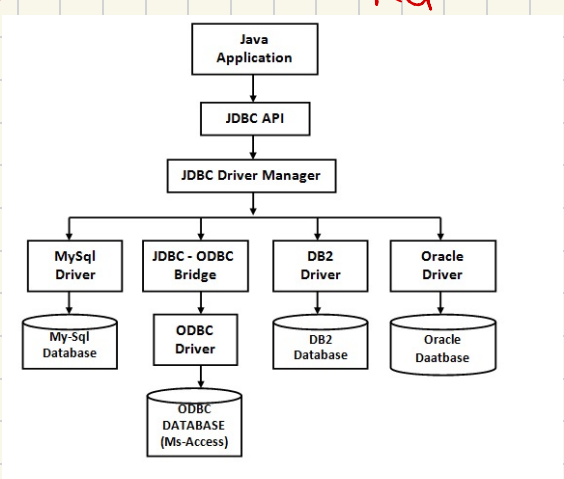
Jar → Interface, class, method

maven terms

dependency

Company give us API we have to implement as we want and use it.





We are learning JDBC only for practice now a days we use SPRING-DATA-JPA

Lec 2

Introduction to JDBC

JDBC is an API (Application Programming Interface) designed to connect with a database and perform basic DB operations.

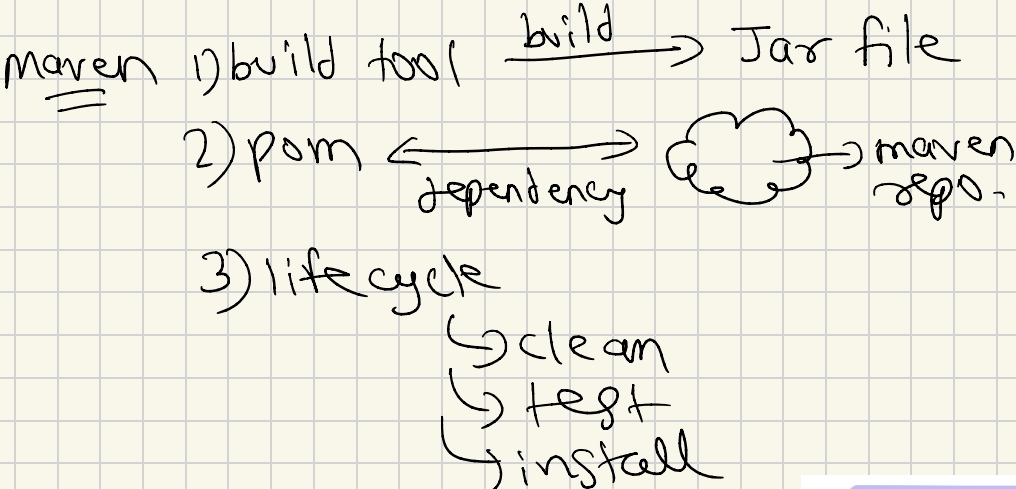
In a Java application:

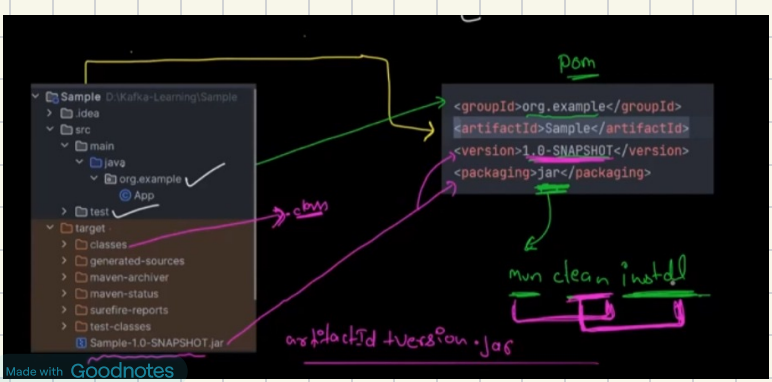
- **JDBC** provides predefined methods to connect to the database.
- It allows the developer to send a representation of the query logic to the database engine.

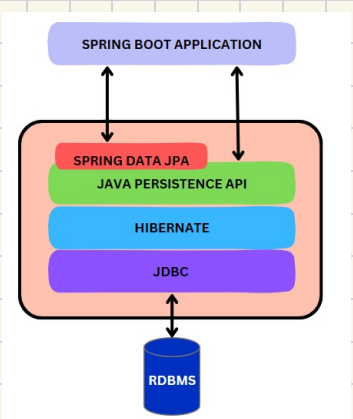
Client: Sends a database query request.
Database Engine: Processes the query and sends the result back.

Data Representation: Data is represented as objects in Java, but JDBC enables developers to write database logic in query language representation.

Drivers:
Drivers are communication mechanisms between the Java application and the database to perform database operations.







Prerequisites:

Home work
CRUD operation
using JDBC

1. Install MySQL and create a database (e.g., `testdb`).
2. Create a table:

```
CREATE TABLE users (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(50),
    email VARCHAR(50)
);
```

3. Add the MySQL JDBC Driver to your project (`mysql-connector-java`).

Simple JDBC Application in Java:

output

```
Connected to the database!
1 row(s) inserted!
Users in the database:
ID: 1, Name: John Doe, Email: john.doe@example.com
```

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

public class SimpleJDBCApp {
    public static void main(String[] args) {
        // Database credentials
        String jdbcURL = "jdbc:mysql://localhost:3306/testdb"; // Replace with your DB name
        String username = "root"; // Replace with your DB username
        String password = "password"; // Replace with your DB password

        // JDBC objects
        Connection connection = null;
        PreparedStatement preparedStatement = null;
        ResultSet resultSet = null;

        try {
            // Step 1: Establish a connection
            connection = DriverManager.getConnection(jdbcURL, username, password);
            System.out.println("Connected to the database!");

            // Step 2: Insert data
            String insertQuery = "INSERT INTO users (name, email) VALUES (?, ?)";
            preparedStatement = connection.prepareStatement(insertQuery);
            preparedStatement.setString(1, "John Doe");
            preparedStatement.setString(2, "john.doe@example.com");
            int rowsInserted = preparedStatement.executeUpdate();
            System.out.println(rowsInserted + " row(s) inserted!");

            // Step 3: Retrieve data
            String selectQuery = "SELECT * FROM users";
            preparedStatement = connection.prepareStatement(selectQuery);
            resultSet = preparedStatement.executeQuery();

            System.out.println("Users in the database:");
            while (resultSet.next()) {
                int id = resultSet.getInt("id");
                String name = resultSet.getString("name");
                String email = resultSet.getString("email");
                System.out.println("ID: " + id + ", Name: " + name + ", Email: " + email);
            }
        } catch (SQLException e) {
            e.printStackTrace();
        } finally {
            // Step 4: Close resources
            try {
                if (resultSet != null) resultSet.close();
                if (preparedStatement != null) preparedStatement.close();
                if (connection != null) connection.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }
}
```

We use PreparedStatement because it precompiles SQL queries, reducing parsing overhead and improving performance in the database engine, while also preventing SQL injection by safely handling input parameters.

To save an image in a database, use a BLOB column and store the image as a byte array using a PreparedStatement with the setBinaryStream() or setBytes() method.

List of common JDBC jargons with simple one-line explanations:

1. JDBC (Java Database Connectivity): An API for connecting and executing queries in relational databases using Java.
2. Driver: A software component that establishes a communication bridge between Java applications and the database.
3. Connection: A session between the Java application and the database.
4. Statement: An interface for executing static SQL queries in the database.
5. PreparedStatement: A precompiled SQL query that supports parameterized input for better performance and security.
6. CallableStatement: Used to execute stored procedures in the database.
7. ResultSet: A table-like structure that stores the results of a query executed in the database.
8. DriverManager: A class that manages the set of database drivers and establishes a connection to the database.
9. SQL Exception: An exception thrown when a database access error occurs.
10. Transaction: A group of database operations that are treated as a single unit of work, committed or rolled back as a whole.
11. Auto-commit: A mode where each SQL statement is automatically committed after execution.
12. Batch Processing: A technique to execute multiple SQL queries in one go to improve performance.
13. Connection Pooling: A technique to reuse database connections instead of creating new ones repeatedly.
14. RowSet: A ResultSet-like interface that can be serialized and passed between Java components.
15. Metadata: Data about the structure of database objects (e.g., tables, columns) retrievable via JDBC.
16. ODBC (Open Database Connectivity): A standard API for accessing databases, which JDBC is designed to replace in Java.
17. SQL Injection: A security vulnerability where unvalidated input allows malicious SQL code execution.
18. JNDI (Java Naming and Directory Interface): A mechanism for looking up JDBC DataSource objects in enterprise applications.
19. DataSource: An alternative to DriverManager for obtaining database connections, often used with connection pools.
20. Scrollable ResultSet: A ResultSet that allows moving forward and backward through query results.
21. Updatable ResultSet: A ResultSet that allows modifications to the database through the ResultSet directly.
22. ACID Properties: Ensures transactions are Atomic, Consistent, Isolated, and Durable.
23. Savepoint: A point within a transaction to which you can roll back without affecting earlier changes.
24. Isolation Level: Defines the degree of visibility of transactions' changes to others.
25. SQL Dialect: Variations in SQL syntax and features across different database vendors.

Here are concise one-liners for each term:

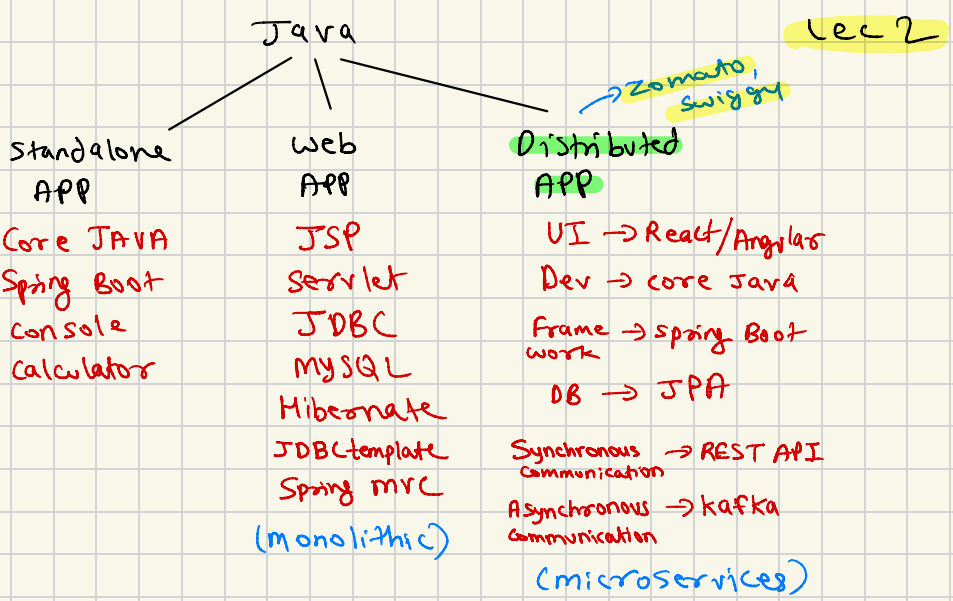
Advance Java - Servlet

Lec 1

1. Trigger: A database function that automatically executes in response to specific events (e.g., INSERT, UPDATE) on a table.
2. Join: Combines rows from two or more tables based on a related column.
3. Partition: Divides a large table into smaller, more manageable pieces for improved query performance and maintenance.
4. Primary Key: A unique identifier for rows in a table that ensures data integrity.
5. Self Join: A join where a table is joined with itself using aliases.
6. Stored Procedure: A precompiled set of SQL statements stored in the database and executed as a single unit.
7. Data Source: An abstraction for database connectivity, often used in enterprise applications for connection pooling.
8. Connection Pooling: Reuses a set of database connections to minimize the overhead of establishing connections repeatedly.

additional important database and JDBC terms every Java developer should know, with one-liners:

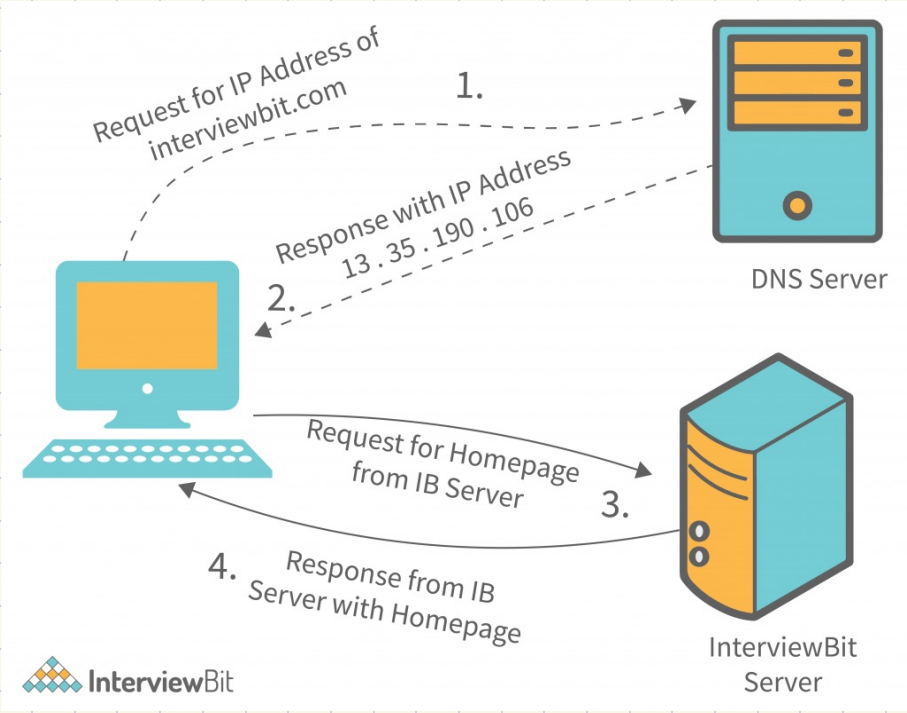
1. Foreign Key: A column that establishes a relationship between two tables by referencing the primary key of another table.
2. Index: A database structure that improves the speed of data retrieval but can slow down write operations (INSERT/UPDATE).
3. Normalization: The process of organizing data to reduce redundancy and improve data integrity.
4. Denormalization: Adding redundancy to optimize query performance, often used in data warehousing.
5. ACID Properties: Guarantees database transactions are Atomic, Consistent, Isolated, and Durable.
6. Query Optimization: Techniques the database engine uses to improve the performance of SQL queries.
7. Subquery: A query nested inside another query to fetch intermediate results.
8. View: A virtual table based on the result of a SELECT query, used for abstraction and simplified querying.
9. Database Schema: A logical structure of database objects like tables, views, and indexes.
10. Cursor: A database object used to iterate over the results of a query one row at a time.
11. Batch Processing: Executes multiple SQL statements in one go to improve performance in bulk operations.
12. LOB (Large Object): Stores large data such as text (CLOB) or binary data (BLOB) in the database.
13. 2-Phase Commit: Ensures atomicity across distributed transactions by preparing all changes before committing them.
14. Lazy Loading: Defers the retrieval of associated data until it's specifically accessed, optimizing resource usage.
15. Eager Loading: Loads all associated data upfront to reduce multiple database queries.
16. Transaction Isolation Levels: Controls visibility of intermediate transaction states, e.g., READ COMMITTED, SERIALIZABLE.
17. Deadlock: A situation where two or more transactions wait for each other to release resources, causing a standstill.
18. Paging: Fetching records in chunks (e.g., using LIMIT/OFFSET) to handle large result sets efficiently.
19. Sharding: Splits data across multiple databases or servers to scale horizontally and improve performance.
20. Database Migration: The process of evolving database schemas in a controlled way (tools like Flyway or Liquibase).
21. Query Execution Plan: The strategy a database uses to execute a query, which can be analyzed for optimization.
22. NoSQL Database: A non-relational database designed for flexible, schema-less storage (e.g., MongoDB, Cassandra).
23. Object-Relational Mapping (ORM): Maps Java objects to database tables (e.g., Hibernate, JPA) to simplify database operations.
24. Data Integrity: Ensures accuracy and consistency of data through constraints like primary keys and foreign keys.
25. Database Locking: Mechanism to control simultaneous access to a database resource to maintain consistency.

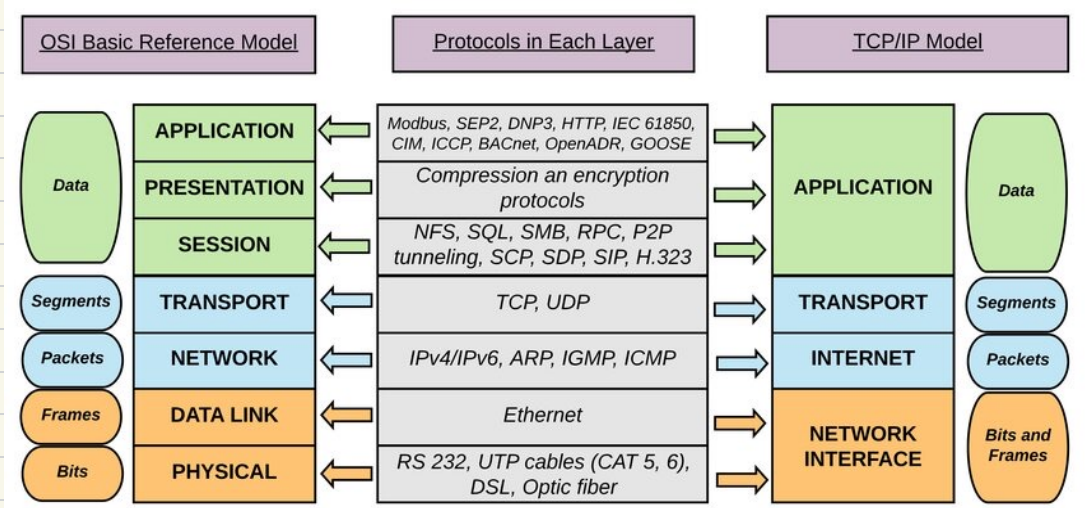


explanations for standalone, web, and distributed apps, along with examples and tools used for development:

1. **Standalone Application**:
   - **Definition**: A desktop application that runs on a single system without relying on a network.
   - **Example**: Media Player or Calculator.
   - **Tools**: Developed using Java Swing, JavaFX, or other desktop frameworks.
2. **Web Application**:
   - **Definition**: An application accessible through a web browser that runs on a server and serves clients over the internet.
   - **Example**: Gmail or Amazon.
   - **Tools**: Developed using Spring Boot, Java Servlets, JSP, or frameworks like Angular (frontend) with REST APIs.
3. **Distributed Application**:
   - **Definition**: An application where components run on multiple systems and communicate over a network.
   - **Example**: Online banking systems or Microservices-based apps.
   - **Tools**: Developed using technologies like Spring Boot, gRPC, REST APIs, Apache Kafka, or cloud platforms like AWS.

Client Server Architecture

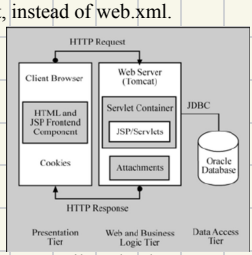




Here's a list of all essential Servlet jargons with one-line definitions:

Lec 3

1. Servlet: A Java class that handles HTTP requests and generates dynamic web content.
2. HTTP (HyperText Transfer Protocol): A protocol for communication between clients (browsers) and servers over the web.
3. Request: The data sent by a client to the server, such as parameters, headers, and body.
4. Response: The data sent back by the server to the client, such as HTML, JSON, or an HTTP status code.
5. Servlet Container: A server (e.g., Tomcat) that manages the lifecycle of servlets and handles HTTP requests.
6. Servlet Lifecycle: The stages of a servlet's execution: initialization (init), request processing (service), and destruction (destroy).
7. HttpServlet: A base class in Java that simplifies handling HTTP-specific requests like GET, POST, PUT, DELETE.
8. doGet(): A method in HttpServlet used to handle HTTP GET requests (typically for retrieving data).
9. doPost(): A method in HttpServlet used to handle HTTP POST requests (typically for sending data to the server).
10. Request Dispatcher: An interface used to forward a request to another servlet or include its response.
11. Session: A mechanism to maintain user state across multiple HTTP requests (e.g., shopping cart).
12. Cookies: Small pieces of data stored on the client side to track user state between HTTP requests.
13. Context Parameters: Application-wide configuration parameters defined in web.xml for servlets.
14. Init Parameters: Configuration parameters specific to a servlet, defined in web.xml or annotations.
15. Filter: A reusable component that intercepts requests and responses for preprocessing or postprocessing (e.g., authentication).
16. Listener: A component that listens to servlet container events (e.g., session creation, attribute addition).
17. web.xml (Deployment Descriptor): An XML file used to configure servlets, mappings, and other web application settings.
18. Annotation-based Configuration: A modern way to configure servlets using annotations like @WebServlet, instead of web.xml.
19. Forward: A method to send a request from one servlet to another without involving the client browser.
20. Redirect: A response sent to the client to make the browser request a different URL.
21. ServletContext: An object representing the entire web application, used to share data among servlets.
22. ServletConfig: An object containing servlet-specific configuration information, like init parameters.
23. Thread Safety: Ensuring a servlet handles multiple concurrent requests without conflicting shared data.
24. Multipart Request: An HTTP request used for file uploads, often processed using libraries like Apache Commons FileUpload.
25. Response Buffering: Temporarily storing the servlet's response before sending it to the client to optimize delivery.
26. Status Codes: HTTP codes sent in the response (e.g., 200 for success, 404 for not found, 500 for server error).
27. Session Tracking: Methods like cookies, URL rewriting, or HttpSession to maintain user session state.
28. Asynchronous Servlets: Servlets that support non-blocking operations using the AsyncContext API.
29. MIME Type: Specifies the type of content being sent in the response (e.g., text/html, application/json).
30. CORS (Cross-Origin Resource Sharing): A policy to allow requests from other domains, configured using response headers.
31. Thread Pooling: A container feature that reuses threads to handle multiple servlet requests efficiently.
32. Gzip Compression: Compresses servlet responses to reduce size and improve performance.
33. Error Handling: Configuring custom error pages or handling exceptions in servlets to provide meaningful responses.
34. Servlet Mapping: The URL pattern associated with a servlet, defined in web.xml or annotations.
35. Session Timeout: The duration a session remains valid without user activity, configurable in web.xml.



HTTP Request Methods

optional
compulsory

GET — Retrieves data or resources from a specified URL. It is used to retrieve information without modifying it.

POST — Submits data or creates a new resource on the server. It is used to send data to be processed by the server, often resulting in the creation of a new resource.

PUT — Updates an existing resource with new data. It replaces the entire resource with the new representation provided.

id name email

DELETE — Deletes a specified resource on the server.

PATCH — Partially updates an existing resource. It is used to apply modifications to a resource, specifying only the changes that need to be made.

id name email

HEAD — Retrieves only the headers of a response. It is used to check the status or headers of a resource without fetching the entire content.

OPTIONS — Retrieves the allowed methods and other information about a resource.

TRACE — Echoes back the received request to the client. It is mainly used for diagnostic purposes.

CONNECT — Establishes a tunnel connection to a remote server, typically through a proxy server.

www.automateNow.io

automateNow

HTTP Status Codes

javaconceptoftheday.com

1xx : Informational Purpose	
100	Continue
101	Switching Protocols
102	Processing
103	Early Hints
2xx : Success	
200	Ok
201	Created
202	Accepted
203	Non-Authoritative Information
204	No Content
205	Reset Content
206	Partial Content
207	Multi Status
208	Already Reported
226	IM Used
3xx : Redirection	
300	Multiple Choices
301	Moved Permanently
302	Found
303	See Other
304	Not Modified
305	Use Proxy
306	No Longer Used
307	Temporary Redirect
308	Moved Permanently

4xx : Client Errors	
400	Bad Request
401	Unauthorized
402	Payment Required
403	Forbidden
404	Not Found
405	Method Not Allowed
406	Not Acceptable
407	Proxy Authentication Is Required
408	Request Time Out
409	Conflict
410	Gone
411	Length Required
412	Precondition Failed
413	Payload Too Large
414	URI Too Long
415	Unsupported Media Type
416	Range Not Satisfiable
417	Expectation Failed
421	Misdirect Request
422	Unprocessable Entity
423	Locked
424	Failed Dependency
425	Too Early
426	Upgrade Required
428	Precondition Required
429	Too Many Requests
431	Request Header Fields Too Large
451	Unavailable For Legal Reasons

5xx : Server Errors	
500	Internal Server Error
501	Not Implemented
502	Bad Gateway
503	Service Unavailable
504	Gateway Timeout
505	HTTP Version Not Supported
507	Insufficient Storage
508	Loop Detected
510	Not Extended
511	Network Authentication Required

Steps to Develop a Servlet and How Servlet, GenericServlet, and HttpServlet Evolve:

lec 4

1. **Using `Servlet`:**
   - Implement the `Servlet` interface.
   - You must override **all methods** (`init`, `service`, `destroy`, `getServletConfig`, `getServletInfo`), even unused ones.
   - **Problem**: Too much boilerplate code.
2. **Using `GenericServlet`:**
   - Extend `GenericServlet`, which provides default implementations for most `Servlet` methods.
   - You only need to implement the `service` method to handle requests.
   - **Limitation**: No HTTP-specific request handling (like GET or POST).
3. **Using `HttpServlet`:**
   - Extend `HttpServlet`, which is specialized for HTTP.
   - Override only the HTTP methods you need, such as `doGet` or `doPost`.
   - Simplifies handling HTTP requests with minimal code.

Example Progression:

- **Servlet**: Boilerplate-heavy.
- **GenericServlet**: Simplifies by focusing on the `service` method.
- **HttpServlet**: Streamlined for HTTP-specific use cases.

Final Recommendation:

For web applications, always prefer `HttpServlet` for handling HTTP requests efficiently.

Lec 5

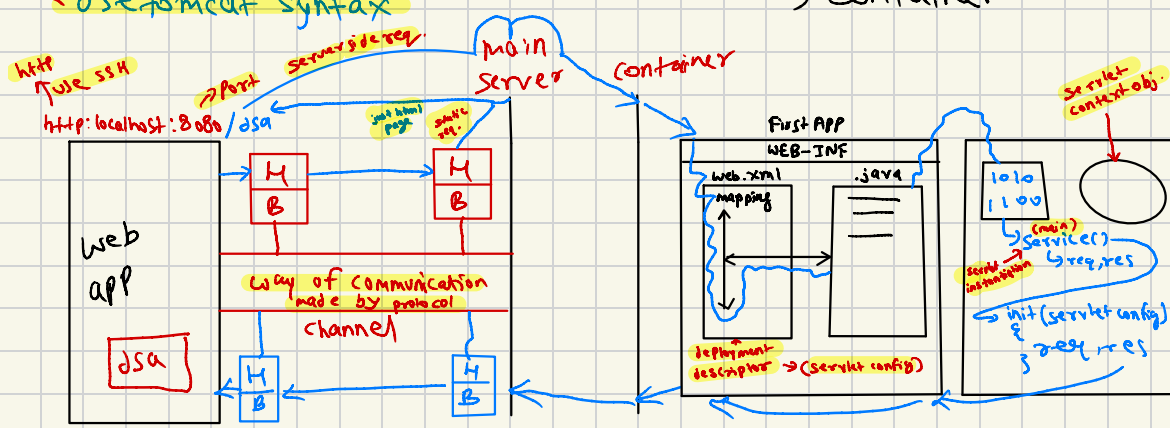
A **server** is a computer system or software that provides resources, services, or functionality to other devices (clients) over a network. Examples include web servers, database servers, and file servers.

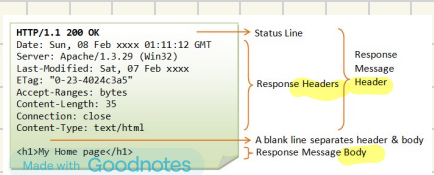
- **Web Server**: Handles HTTP requests and serves static content like HTML, CSS, and JavaScript. Example: Nginx, Apache.
- **Application Server**: Executes business logic and generates dynamic content, often using frameworks or APIs. Example: Tomcat, JBoss, weblogic.
- **Java Simple Run**: Compile with `javac` to bytecode, then run with `java ClassName` in JVM.
- **Tomcat Run**: Start Tomcat, deploy `.war` files, and handle requests on `http://localhost:8080`.

having main method
demo.java → run (javac demo.java, java demo) → output

no main method
demo.java → run with tomcat → output
servlet-api.jar (given by tomcat) → use this with help of classpath
use tomcat syntax

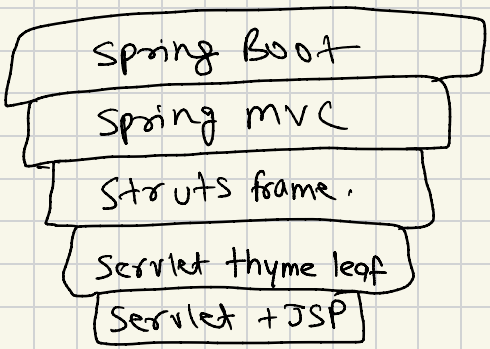
Protocol = TCP < Header(H), Body(B)
Server uses TCP
(archetype-webapp)
Tomcat
↳ main server
↳ container





Servlet → java class → Tomcat → run

How to make servlet
1) Implement Servlet Interface
2) Extend Generic Servlet (abstract class)
3) Extend HTTPServlet (abstract class)



Lec 6

One-liner definitions:

1. Spring Boot: A framework that simplifies Spring application development by providing auto-configuration and embedded servers.
2. Spring MVC: A module of Spring Framework for building web applications using the Model-View-Controller architecture.
3. Struts Framework: A Java-based framework for developing web applications using an MVC design pattern, now largely replaced by modern frameworks.
4. Servlet + Thymeleaf: Combines Servlets for backend logic and Thymeleaf as a template engine for rendering dynamic HTML content.
5. Servlet + JSP: Uses Servlets for backend logic and JavaServer Pages (JSP) for rendering dynamic web pages.

Here are concise explanations for each:

1. Servlet Redirection:
   - Use response.sendRedirect("url") for client-side redirection to another page or URL.
   - Example:

```
response.sendRedirect("home.jsp");
```

2. Setup Default Page:
   - Define a default servlet or welcome file in web.xml:

```
<welcome-file-list>
    <welcome-file>index.html</welcome-file>
    <welcome-file>index.jsp</welcome-file>
</welcome-file-list>
```

3. Session Management: (Session stored in server)
   - Use HttpSession to manage user sessions:

```
HttpSession session = request.getSession();
session.setAttribute("user", "John");
```

4. Cookies: (Cookies stored in browser)
   - Add a cookie to the response:

• key:value, small data, unsafe, stored in client side

```
Cookie cookie = new Cookie("username", "John");
response.addCookie(cookie);
```

   - Retrieve a cookie from the request:

```
Cookie[] cookies = request.getCookies();
```

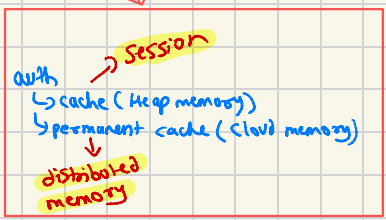
5. Filters in Servlet:
   - Use a Filter to intercept requests and responses for preprocessing or postprocessing.
   - Example in web.xml:

```
<filter>
    <filter-name>MyFilter</filter-name>
    <filter-class>com.example.MyFilter</filter-class>
</filter>
<filter-mapping>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

   - Filter class implementation:

```
public class MyFilter implements Filter {
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
            throws IOException, ServletException {
        // Preprocessing
        chain.doFilter(request, response);
        // Postprocessing
    }
```


• important



Note: RFST → Stateless (session less)