1. **Dynamic Method Dispatch**
2. **Abstract Classes**
3. **Uses of final keyword**

---

1. Dynamic Method Dispatch (Run-time Polymorphism)

Definition

Dynamic Method Dispatch in Java is the mechanism by which a **call to an overridden method** is resolved at **runtime** rather than **compile-time**.
It is used to achieve **run-time polymorphism**.

Key Points

- Occurs **only** with **overridden methods**, not with data members.
- Requires **inheritance**.
- Reference variable of **parent class** refers to the **object of child class**.
- Method that gets executed is determined by **object type**, not reference type.

Syntax

ParentClass ref;

ref = new ChildClass(); // runtime decision for method execution

Example

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

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

```
class Cat extends Animal {
    void sound() {
        System.out.println("Cat meows");
    }
}

public class DynamicDispatchDemo {
    public static void main(String[] args) {
        Animal a;

        a = new Dog();
        a.sound(); // Calls Dog's version

        a = new Cat();
        a.sound(); // Calls Cat's version
    }
}
```

Output

```
Dog barks
Cat meows
```

2. Abstract Classes

Definition

An **abstract class** in Java is a class that is declared using the abstract keyword and **cannot be instantiated** directly.
It may contain:

- **Abstract methods** (without body)
- **Concrete methods** (with body)

Key Points

- Used when you want to provide a **base class** with **common code** and enforce implementation of some methods in subclasses.
- Subclasses must **override all abstract methods** or be declared abstract themselves.
- Can have **constructors, static methods, and final methods**.

Syntax

```
abstract class ClassName {
   abstract void methodName(); // abstract method
   void normalMethod() { }     // concrete method
}
```

Example

```
abstract class Shape {
   abstract void draw(); // abstract method
   void message() {
      System.out.println("Drawing a shape");
   }
}
class Circle extends Shape {
   void draw() {
      System.out.println("Drawing a Circle");
   }
}
```

```
class Rectangle extends Shape {
    void draw() {
        System.out.println("Drawing a Rectangle");
    }
}

public class AbstractDemo {
    public static void main(String[] args) {
        Shape s;

        s = new Circle();
        s.draw();
        s.message();

        s = new Rectangle();
        s.draw();
        s.message();
    }
}
```

Output

```
Drawing a Circle
Drawing a shape
Drawing a Rectangle
Drawing a shape
```

3. Uses of final Keyword

The final keyword in Java can be applied to:

- **Variables** → makes them constants (value cannot change once assigned)
- **Methods** → prevents method overriding
- **Classes** → prevents inheritance

3.1. final Variable

```
public class FinalVariableDemo {
    public static void main(String[] args) {
        final int SPEED_LIMIT = 80;
        System.out.println("Speed Limit: " + SPEED_LIMIT);

        // SPEED_LIMIT = 100; // Error: cannot assign a value to final variable
    }
}
```

Output

```
Speed Limit: 80
```

3.2. final Method

```
class Vehicle {
    final void run() {
        System.out.println("Vehicle is running");
    }
}

class Car extends Vehicle {
    // void run() { } // Error: Cannot override final method
}
```

```
public class FinalMethodDemo {
    public static void main(String[] args) {
        Car c = new Car();
        c.run();
    }
}
```

Output

```
Vehicle is running
```

3.3. final Class

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

// class SportsBike extends Bike { } // Error: Cannot inherit from final class

public class FinalClassDemo {
    public static void main(String[] args) {
        Bike b = new Bike();
        b.display();
    }
}
```

Output

```
This is a final class
```