# Moment Platform Digital Experience Platform (DXP)

Java Interview Questions- Sreeni

June 2025

# Java Chapters

## FUNDAMENTALS

## OOPS

## OTHERS

# Java Chapters

## JAVA CODING PROBLEMS

# Java Chapters

## SPRING

1. Basics, IoC & DI
2. Components & Beans
3. Configuration & Annotations
4. Scopes of a bean
5. Others

## SPRING BOOT

6. Basics
7. Project structure, Configuration & Actuator

## SPRING MVC

8. Basics
9. Important Annotations

## REST WEBSERVICES/ REST API

1. Basics
2. HTTP Methods & Status Codes
3. CORS, Serialization, Deserialization, Others
4. Authentication & Authorization

# Java Chapters

## MOCK INTERVIEWS JAVA

1. Java-Basics, Variables & Data types, Operators, Control statements
2. OOPS - Classes, Objects, Access Specifiers, Getter-Setter & this keyword
3. OOPS - Inheritance, Polymorphism, Encapsulation & Abstraction
4. Abstract class & Interface, Constructors
5. Exception Handling
6. Collections
7. Multithreading
8. Generics
9. Types of Classes

## MOCK INTERVIEWS SPRING

1. Spring Basics, IoC, DI, Components & Beans
2. Spring Configuration, Annotations, AOP, Scope of a Bean
3. Spring Boot
4. Spring MVC

# 1. Java - Basics

Q. What is Java?

Q. What are JDK, JRE & JVM? How a Java program compiled or executed? **V. IMP.**

Q. What is compile-time and run-time in Java?

Q. What are the main features and advantages of Java? **V. IMP.**

Q. How Java is platform independent? Why convert java code to bytecode?

Q. How to setup VS Code for Java? (not an interview question)

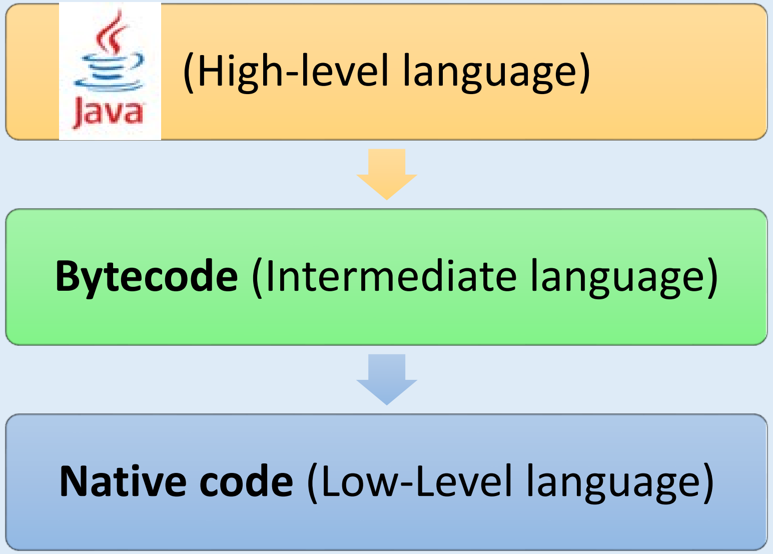
Q. What is main method in Java? What is the role of public, static and void in it? **V. IMP.**

Q. What is Java Bytecode? What is high-level, low-level code?

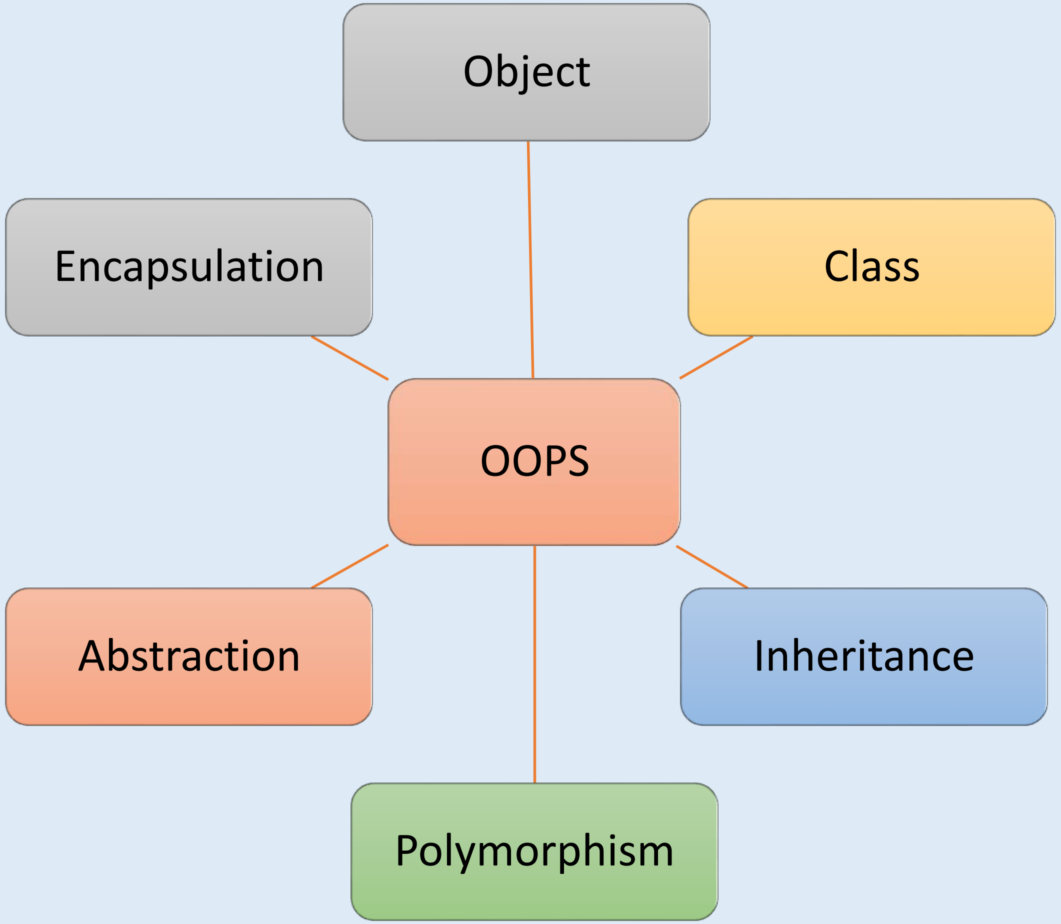
Back to main index

# Q. What is Java?

- Java is a high-level, object-oriented programming language.



Java
(High-level language)
**Bytecode** (Intermediate language)
**Native code** (Low-Level language)



back to chapter index

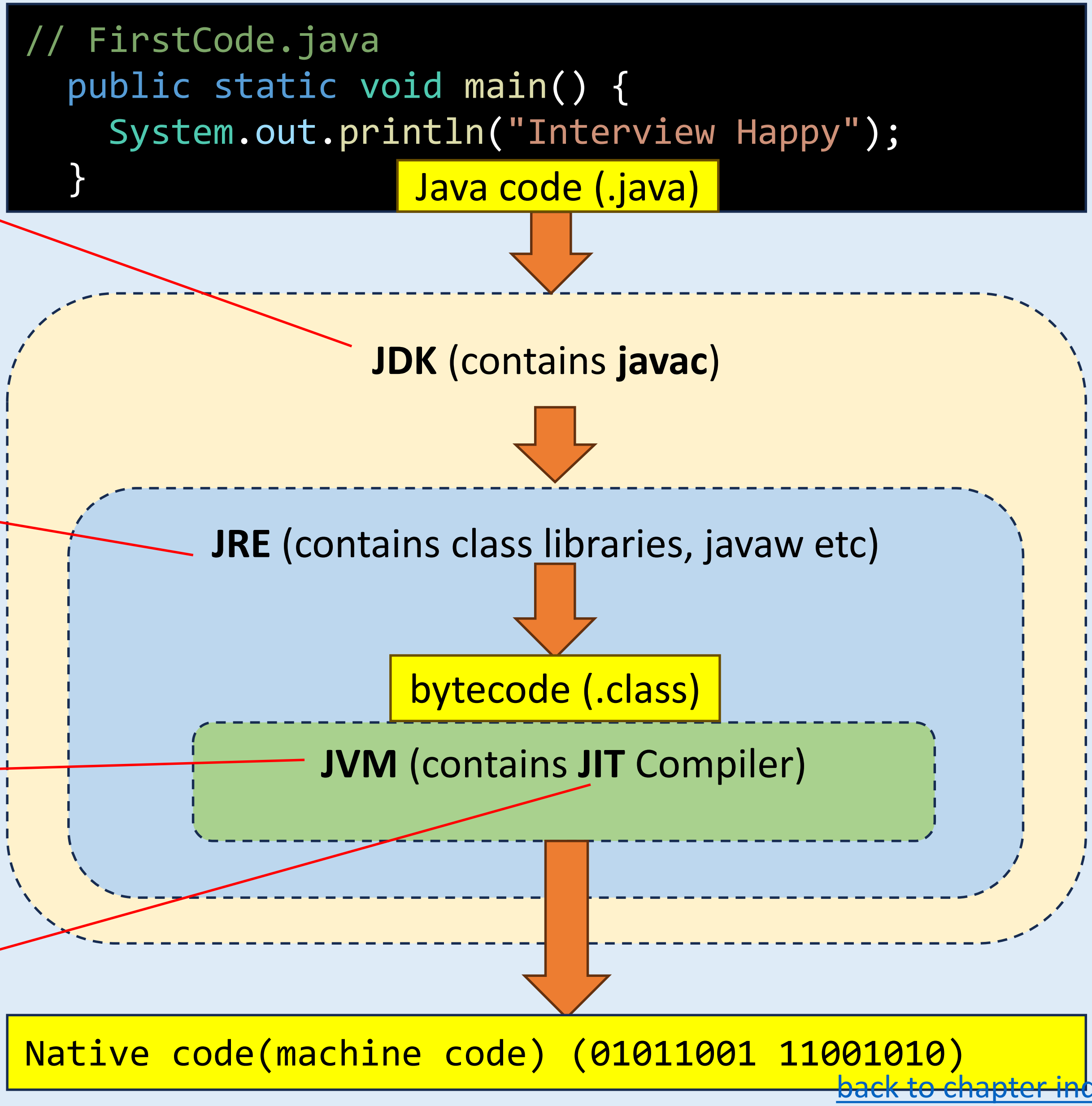
# Q. What are JDK, JRE & JVM? How a Java program compiled or executed? V. IMP.

- **JDK (Java Development Kit):** Includes tools for Java application development, such as javac, jheap, jconsole etc.
    - javac tool compile your .java file into bytecode (.class file)

- **JRE (Java Runtime Environment):** Provides the runtime environment for executing Java applications, including the JVM.
    - contains Java class libraries/ Java API which assist your .java file to execute.

- **JVM (Java Virtual Machine):** Executes Java bytecode, translating it into native machine code. Also do memory management etc.

- **JIT (Just-In-Time)** compiler is a component of the JVM that compiles Java bytecode into native machine code at runtime.



```
// FirstCode.java
  public static void main() {
    System.out.println("Interview Happy");
  }
```

Java code (.java)

**JDK** (contains **javac**)

**JRE** (contains class libraries, javaw etc)

bytecode (.class)

**JVM** (contains **JIT** Compiler)

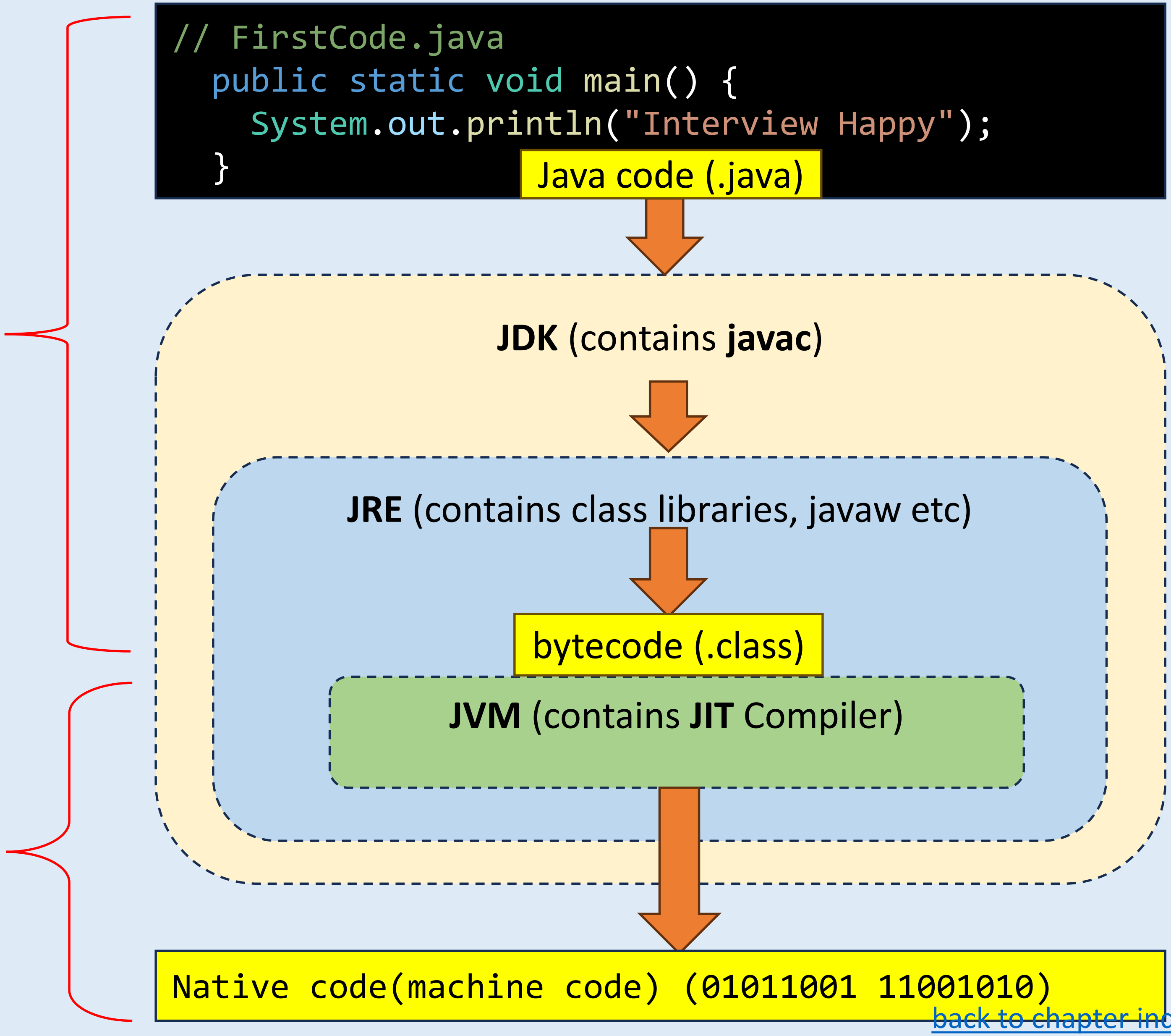
`Native code(machine code) (01011001 11001010)`

back to chapter index

# Q. What is compile-time and run-time in Java?

- **Compile Time:** Compile time is the phase during which the Java source code is translated into bytecode by the Java compiler (javac).

- **Runtime:** Runtime refers to the phase during which the Java Virtual Machine (JVM) executes the compiled bytecode.



```
// FirstCode.java
  public static void main() {
    System.out.println("Interview Happy");
  }
```

Java code (.java)
JDK (contains javac)
JRE (contains class libraries, javaw etc)
bytecode (.class)
JVM (contains JIT Compiler)
Native code(machine code) (01011001 11001010)

back to chapter index

# Q. What are the main features and advantages of Java? V. IMP.

❖ **5 key features and advantages of Java:**

**1. Platform independence**

Java programs run on any device with a Java Virtual Machine (JVM), ensuring compatibility across different platforms(windows, linux etc).

**2. Object-Oriented**

Supports encapsulation, inheritance, and polymorphism, fostering modular, reusable, and maintainable code.

**3. Robust and Secure**

Automatic memory management, exception handling, and type safety enhance reliability and security.

**4. Multi-threading**

Facilitates concurrent execution of tasks, improving performance and responsiveness.

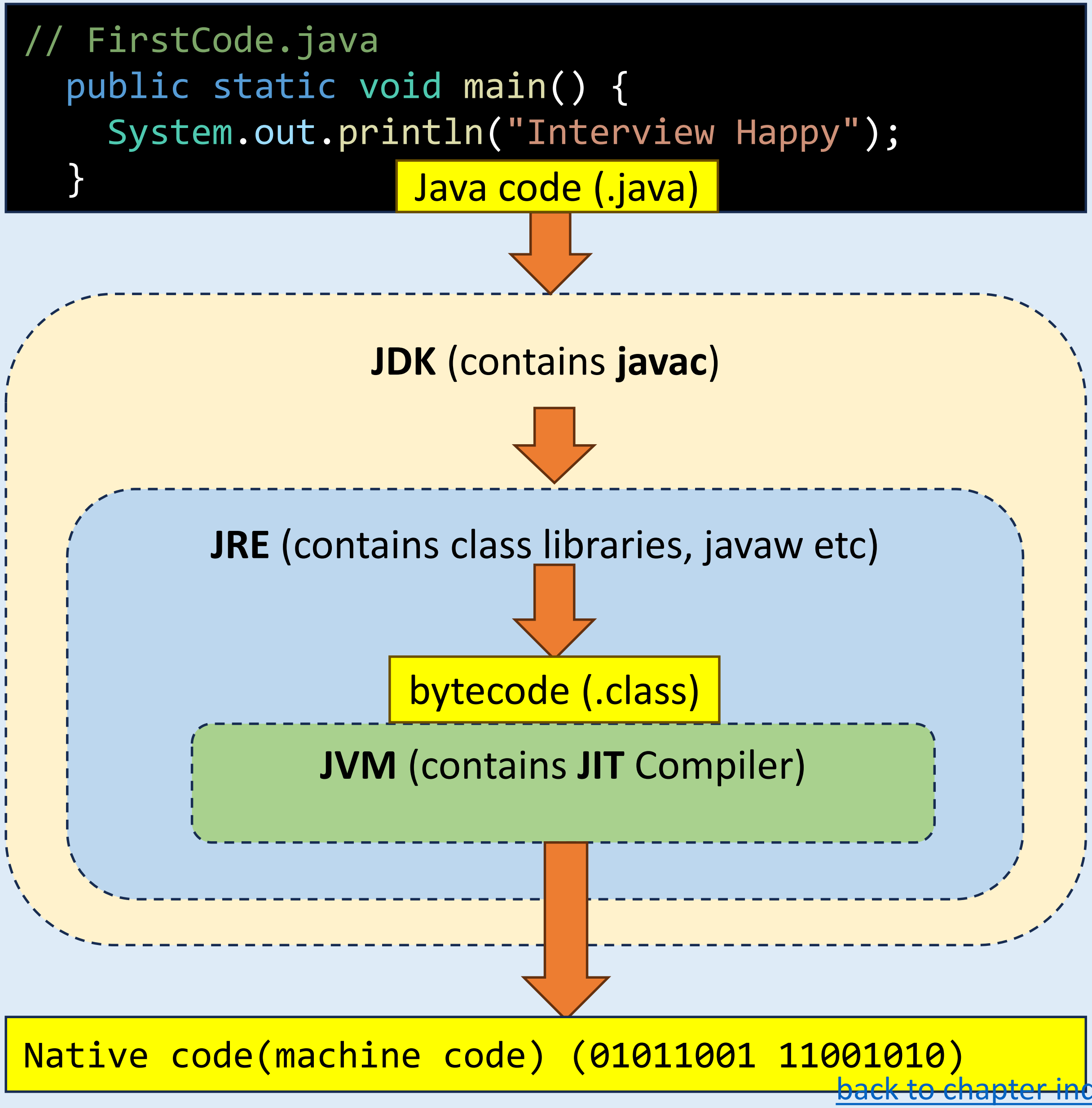
**5. Rich Standard Library**

Comprehensive set of APIs for diverse functionalities like I/O operations, networking, and GUI development, aiding rapid application development.

back to chapter index

# Q. How Java is platform independent? Why convert java code to bytecode?

- Java achieves platform independence by compiling source code into **bytecode**, which can run on any platform with a compatible **JVM**.



```
// FirstCode.java
  public static void main() {
    System.out.println("Interview Happy");
  }
```

Java code (.java)

**JDK** (contains **javac**)

**JRE** (contains class libraries, javaw etc)

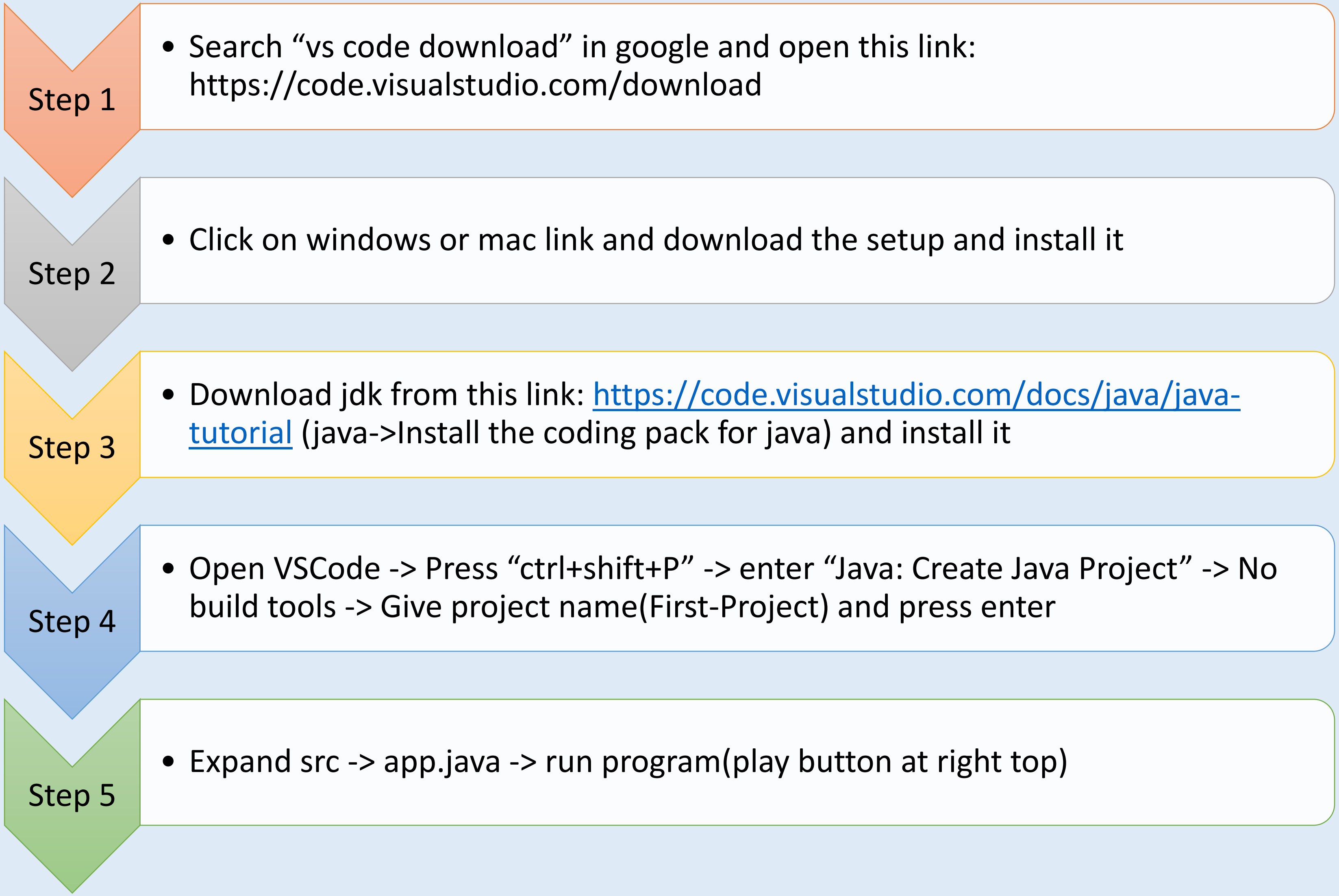
bytecode (.class)

**JVM** (contains **JIT** Compiler)

Native code(machine code) (01011001 11001010)

back to chapter index

# Q. How to setup VS Code for Java? (not an interview question)



Step 1

- Search “vs code download” in google and open this link: https://code.visualstudio.com/download

Step 2

- Click on windows or mac link and download the setup and install it

Step 3

- Download jdk from this link: https://code.visualstudio.com/docs/java/java-tutorial (java->Install the coding pack for java) and install it

Step 4

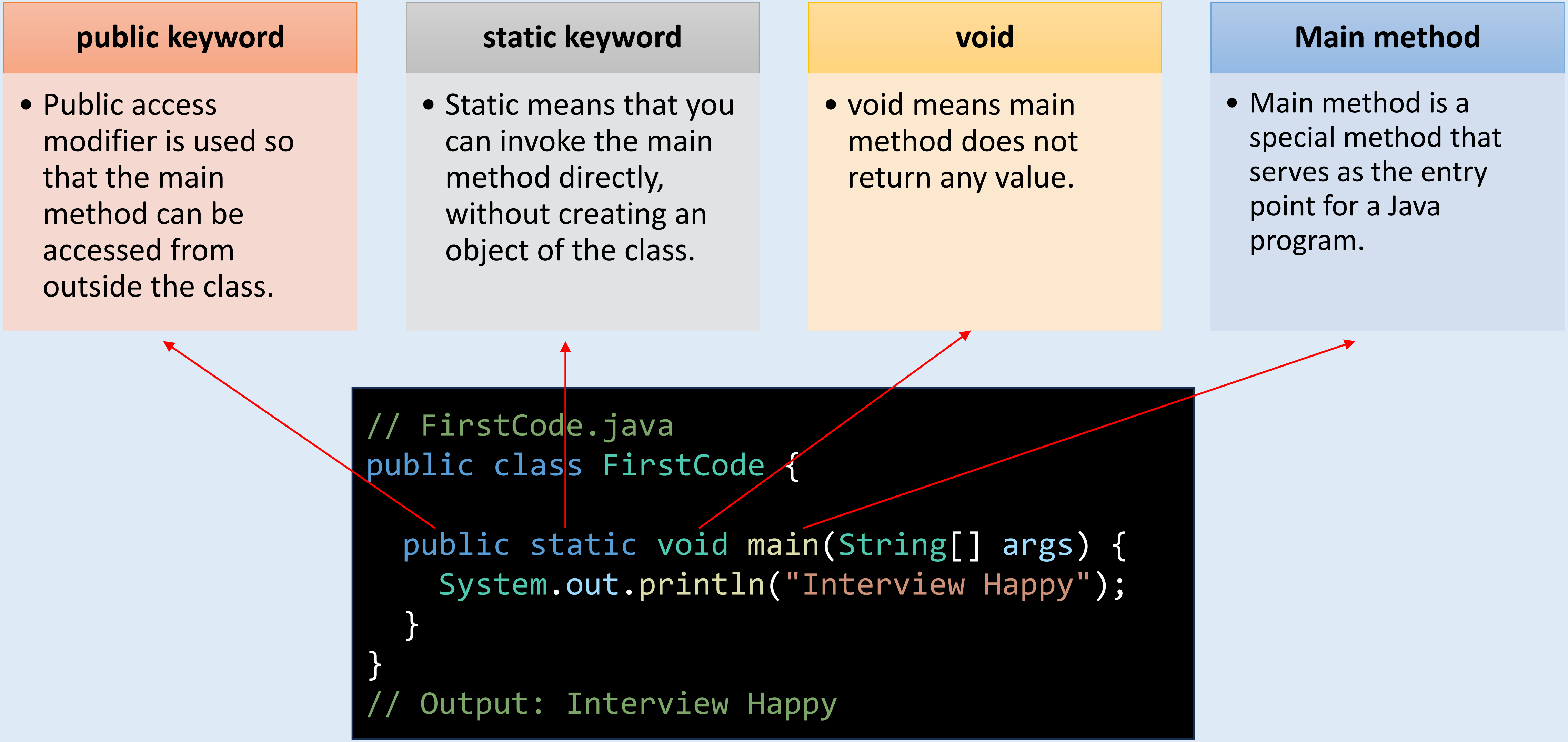
- Open VSCode -> Press “ctrl+shift+P” -> enter “Java: Create Java Project” -> No build tools -> Give project name(First-Project) and press enter

Step 5

- Expand src -> app.java -> run program(play button at right top)

back to chapter index

# Q. What is main method in Java? What is the role of public, static and void in it? V. IMP.



### public keyword

- Public access modifier is used so that the main method can be accessed from outside the class.

### static keyword

- Static means that you can invoke the main method directly, without creating an object of the class.

### void

- void means main method does not return any value.

### Main method

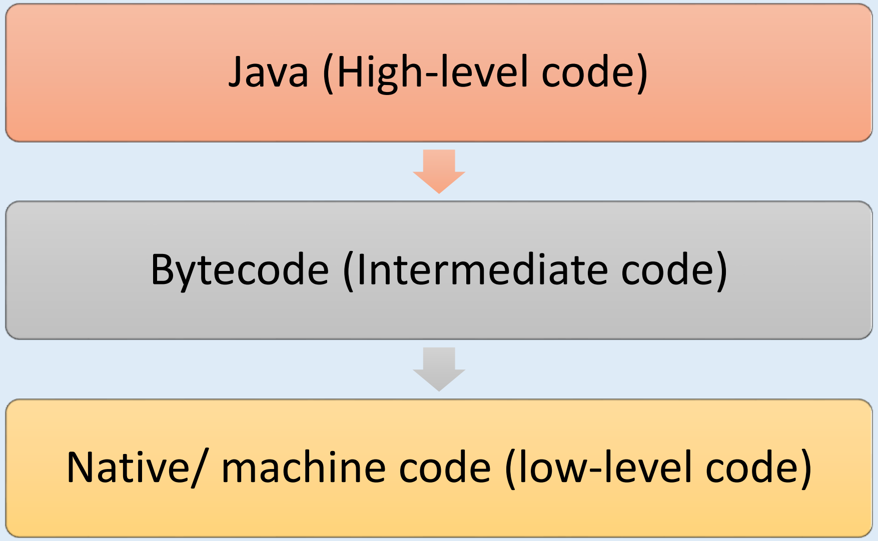
- Main method is a special method that serves as the entry point for a Java program.

```java
// FirstCode.java
public class FirstCode {

  public static void main(String[] args) {
    System.out.println("Interview Happy");
  }
}
// Output: Interview Happy
```

back to chapter index

# Q. What is Java Bytecode? What is high-level, low-level code?

- Java bytecode is a platform-independent intermediate code generated by the Java compiler(javac).

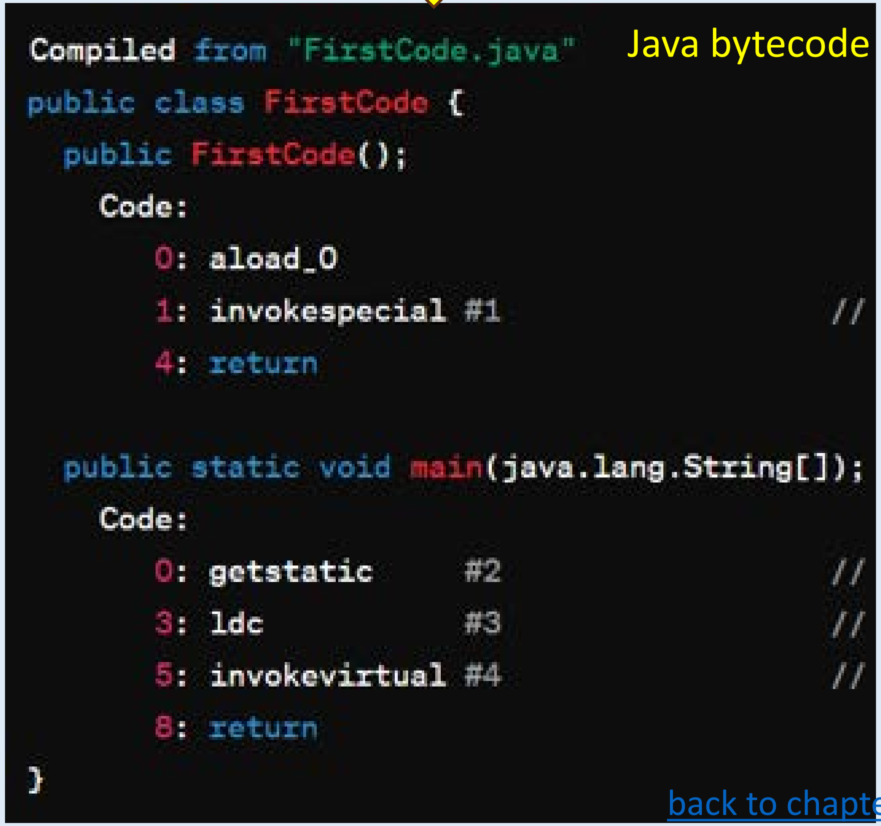

Java (High-level code)
Bytecode (Intermediate code)
Native/ machine code (low-level code)

Java

```java
public class FirstCode {
  public static void main() {
    System.out.println("Interview Happy");
  }
}
```

JDK (javac)

Java bytecode



```
Compiled from "FirstCode.java"
public class FirstCode {
  public FirstCode();
    Code:
       0: aload_0
       1: invokespecial #1                  //
       4: return

  public static void main(java.lang.String[]);
    Code:
       0: getstatic     #2                  //
       3: ldc           #3                  //
       5: invokevirtual #4                  //
       8: return
}
```

back to chapter index

# 2. Variables & Data types

Q. What are variables & data types? What are the types of data types?

Q. What are primitive data types?

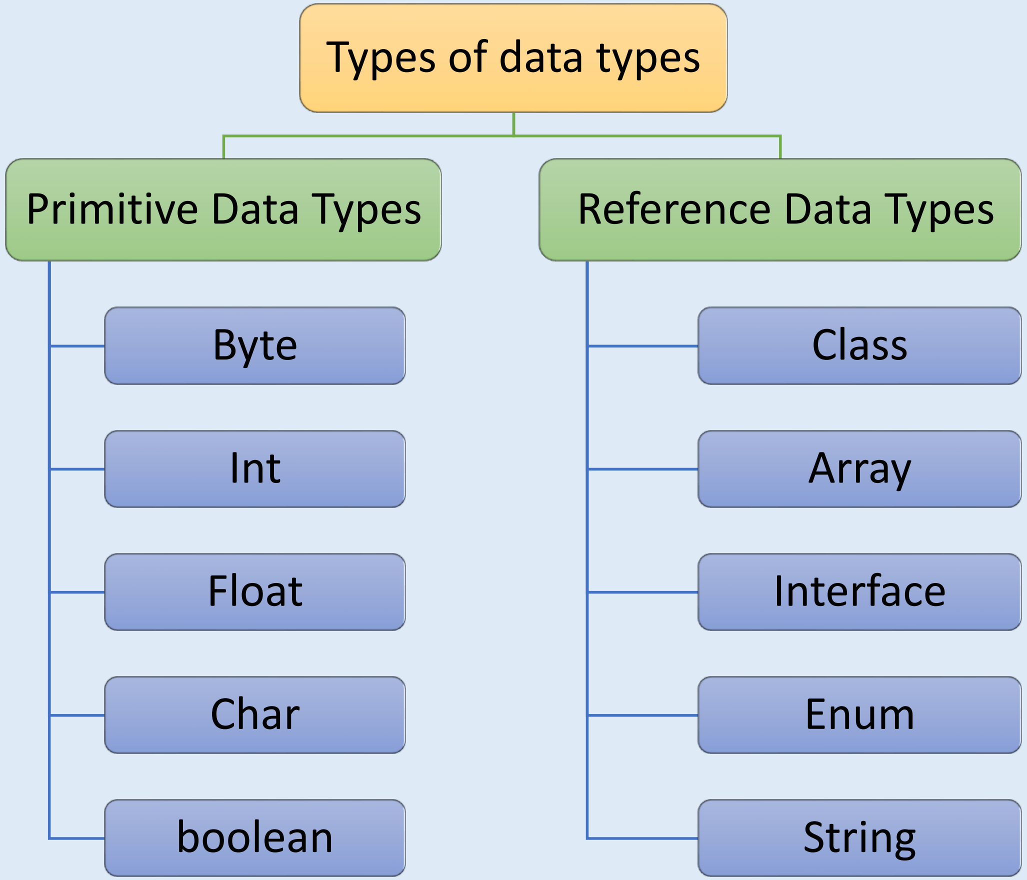
Q. What are reference/ non-primitive data types?

Q. What are the differences between primitive and reference data types? **V. IMP.**

Back to main index

# Q. What are variables & data types? What are the types of data types?

- Variables are used to **store data**.
- Data types define the **type** of variable.



Types of data types

Primitive Data Types

Byte

Int

Float

Char

boolean

Reference Data Types

Class

Array

Interface

Enum

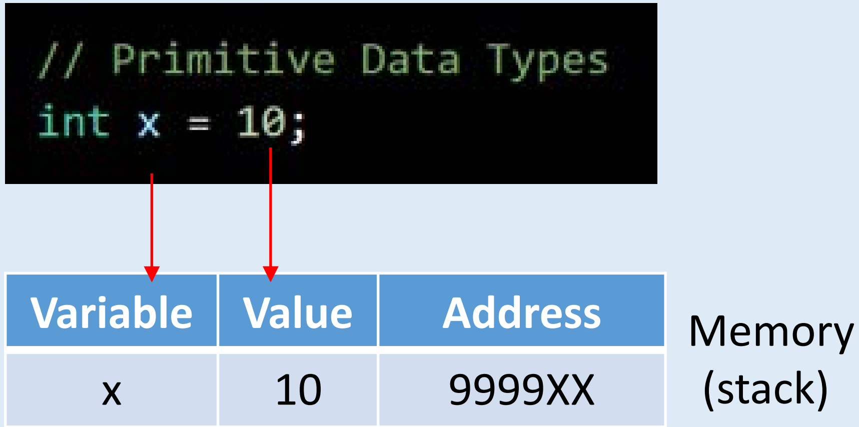
String

```
public class DataTypes {

  public static void main() {
    int count1; // variable declaration
    count1 = 100; // variable initialization

    int count2 = 200; //declaration & initialization
    System.out.println(count1);
    System.out.println(count2);
  }
}

// Output: 100 200
```

back to chapter index

# Q. What are primitive data types?

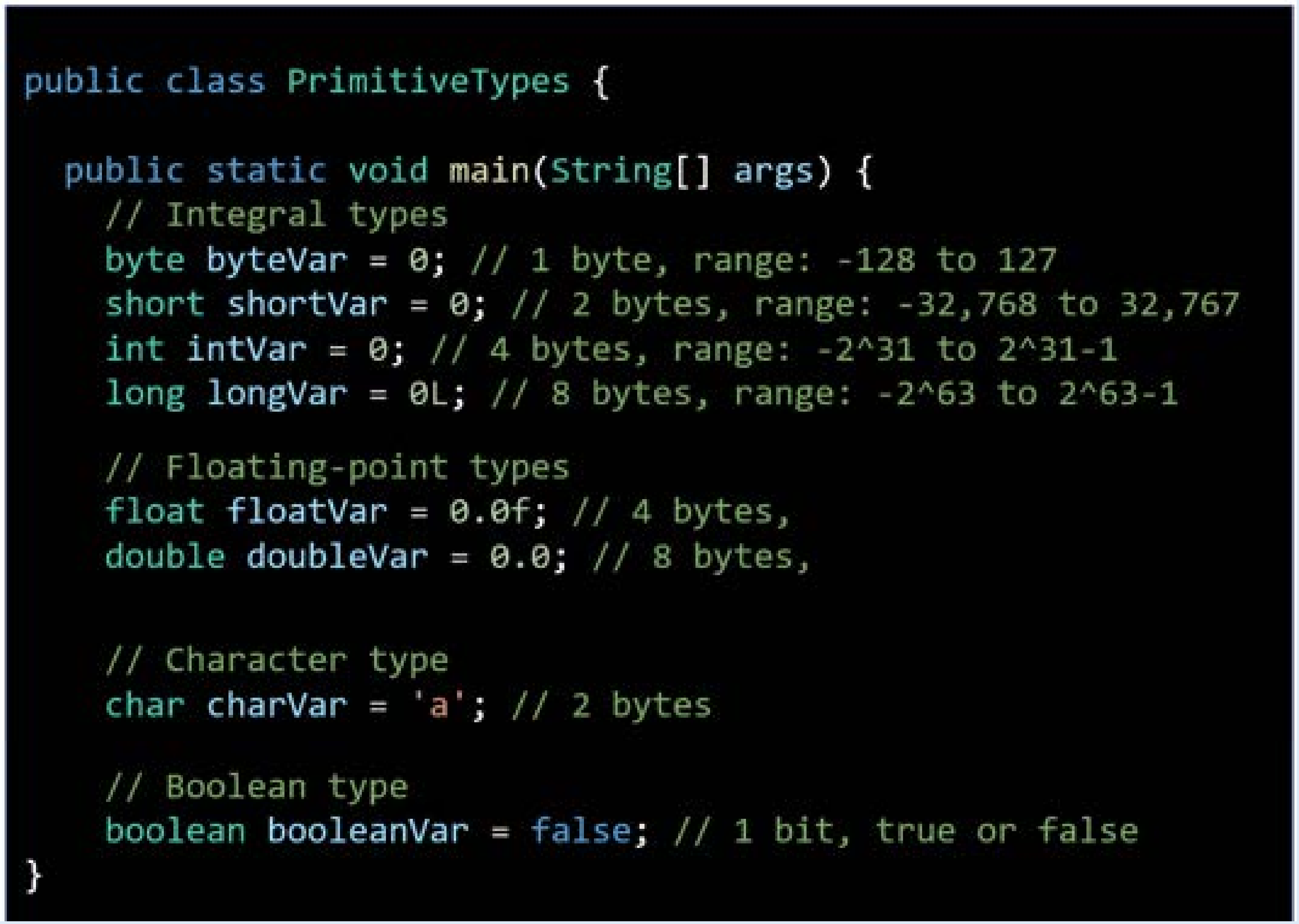
- Primitive data types in Java are basic, **built-in data types that directly store values in memory**.



```
// Primitive Data Types
int x = 10;
```

| Variable | Value | Address |
|---|---|---|
| x | 10 | 9999XX |

Memory (stack)

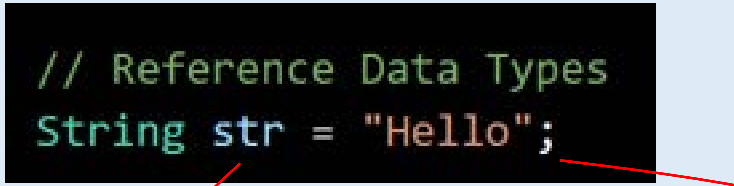


```
public class PrimitiveTypes {

  public static void main(String[] args) {
    // Integral types
    byte byteVar = 0; // 1 byte, range: -128 to 127
    short shortVar = 0; // 2 bytes, range: -32,768 to 32,767
    int intVar = 0; // 4 bytes, range: -2^31 to 2^31-1
    long longVar = 0L; // 8 bytes, range: -2^63 to 2^63-1

    // Floating-point types
    float floatVar = 0.0f; // 4 bytes,
    double doubleVar = 0.0; // 8 bytes,


    // Character type
    char charVar = 'a'; // 2 bytes

    // Boolean type
    boolean booleanVar = false; // 1 bit, true or false
}
```

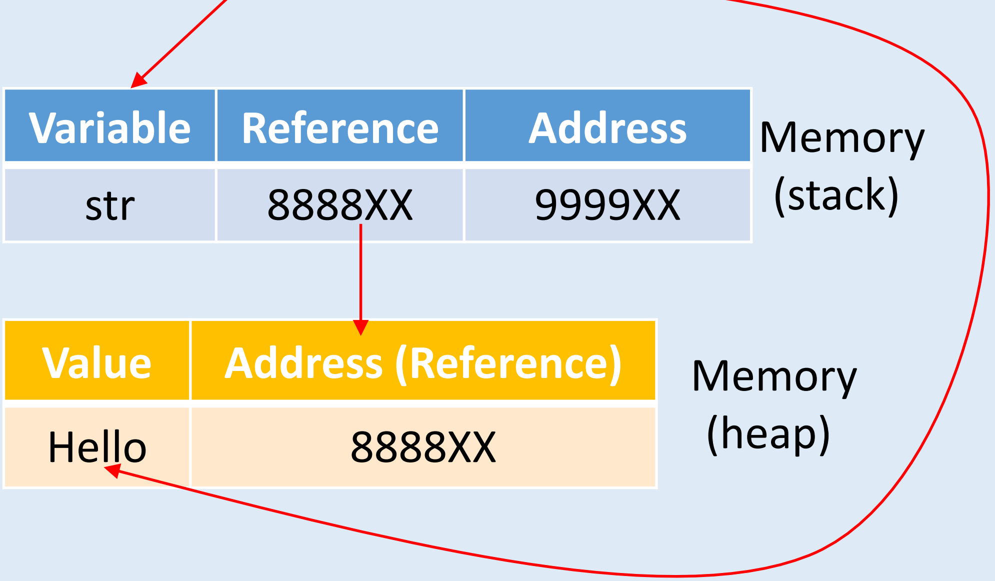
back to chapter index

# Q. What are reference/ non-primitive data types? V. IMP.

- Reference (or non-primitive) data types in Java are data types that **do not store the actual data directly**, but instead store references (heap memory addresses) to values in memory.



```java
// Reference Data Types
String str = "Hello";
```

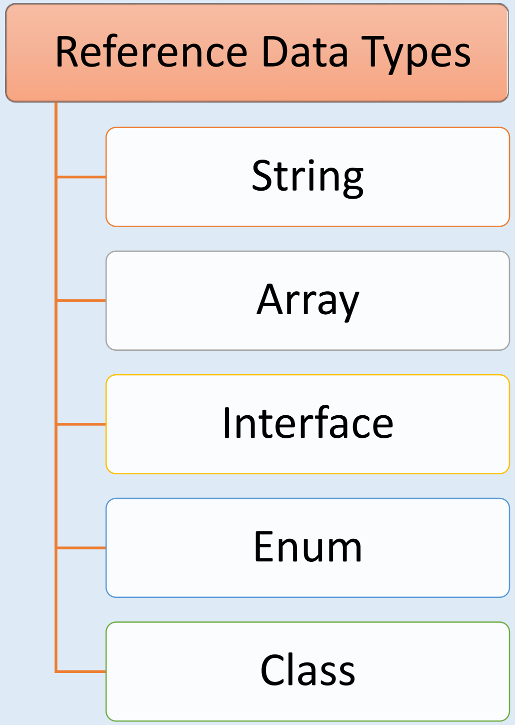


| Variable | Reference | Address |
|---|---|---|
| str | 8888XX | 9999XX |

Memory (stack)

| Value | Address (Reference) |
|---|---|
| Hello | 8888XX |

Memory (heap)

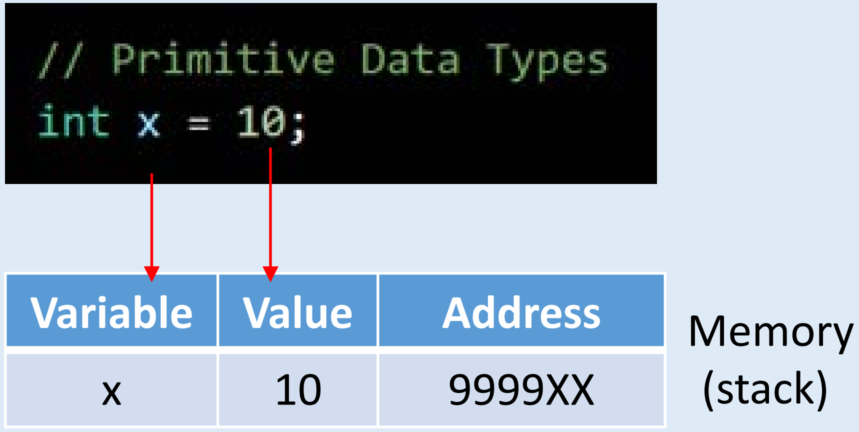


Reference Data Types

- String
- Array
- Interface
- Enum
- Class

back to chapter index

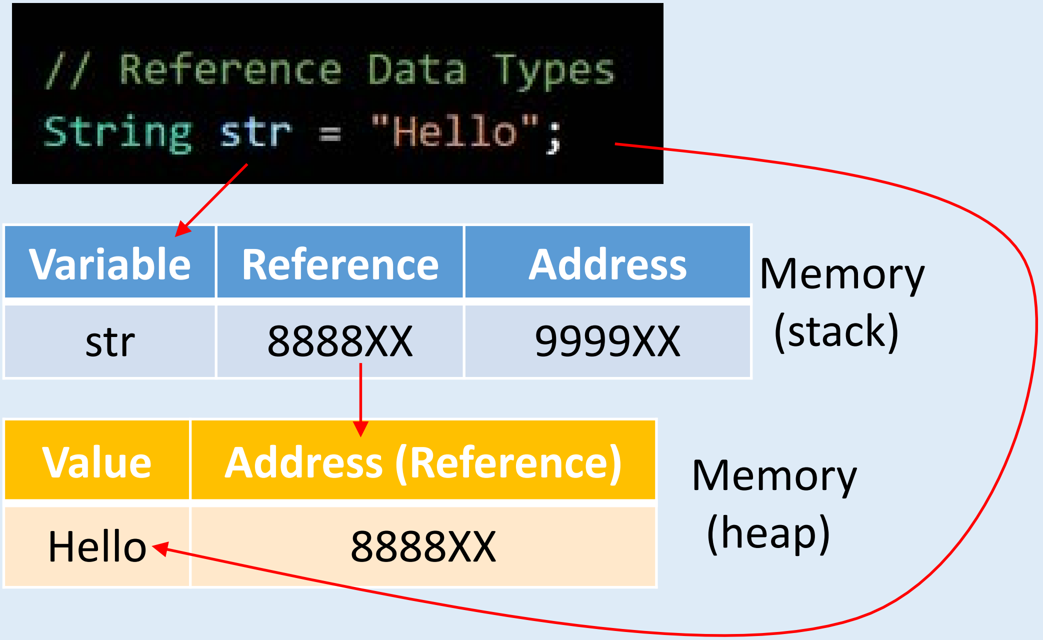
# Q. What are the differences between primitive and reference data types? V. IMP.



```
// Primitive Data Types
int x = 10;
```

| Variable | Value | Address |
|---|---|---|
| x | 10 | 9999XX |

Memory (stack)



```
// Reference Data Types
String str = "Hello";
```

| Variable | Reference | Address |
|---|---|---|
| str | 8888XX | 9999XX |

Memory (stack)

| Value | Address (Reference) |
|---|---|
| Hello | 8888XX |

Memory (heap)

1. Primitive data types store actual values **directly** in memory.
2. Primitive data types mostly hold single values.
3. Primitive data types have **fixed storage**.
4. Primitive data types are stored on the **stack memory**.

1. Reference data types **store references** (heap memory addresses) of the values or objects.
2. Reference data types can hold multiple values.
3. Reference data types have **variable storage sizes**.
4. Reference data types are stored on the **heap memory**, with references stored on the **stack**.

back to chapter index

# 3. Operators

Q. What are Operators? What the types of operators in Java?

Q. What are Arithmetic Operators?

Q. What are Assignment Operators?

Q. What are Comparison Operators? When to use them?

Q. What are Logical Operators? When to use them?

Q. What are Unary Operators?

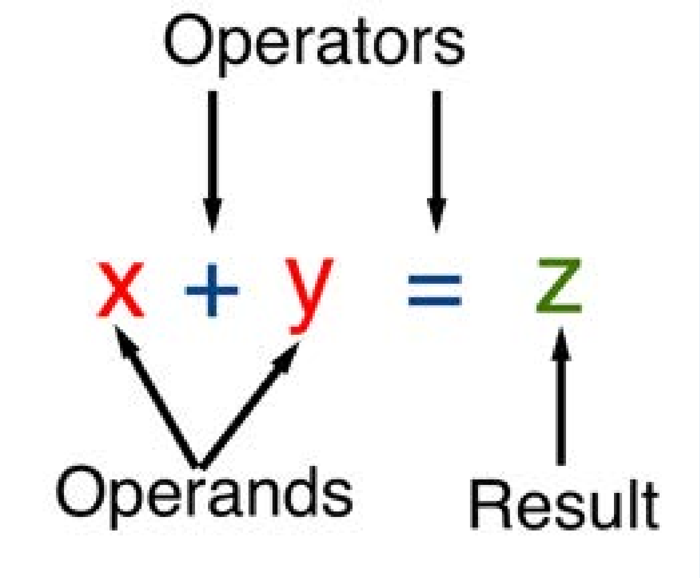
Q. What are Ternary (Conditional) Operators? **V. IMP.**

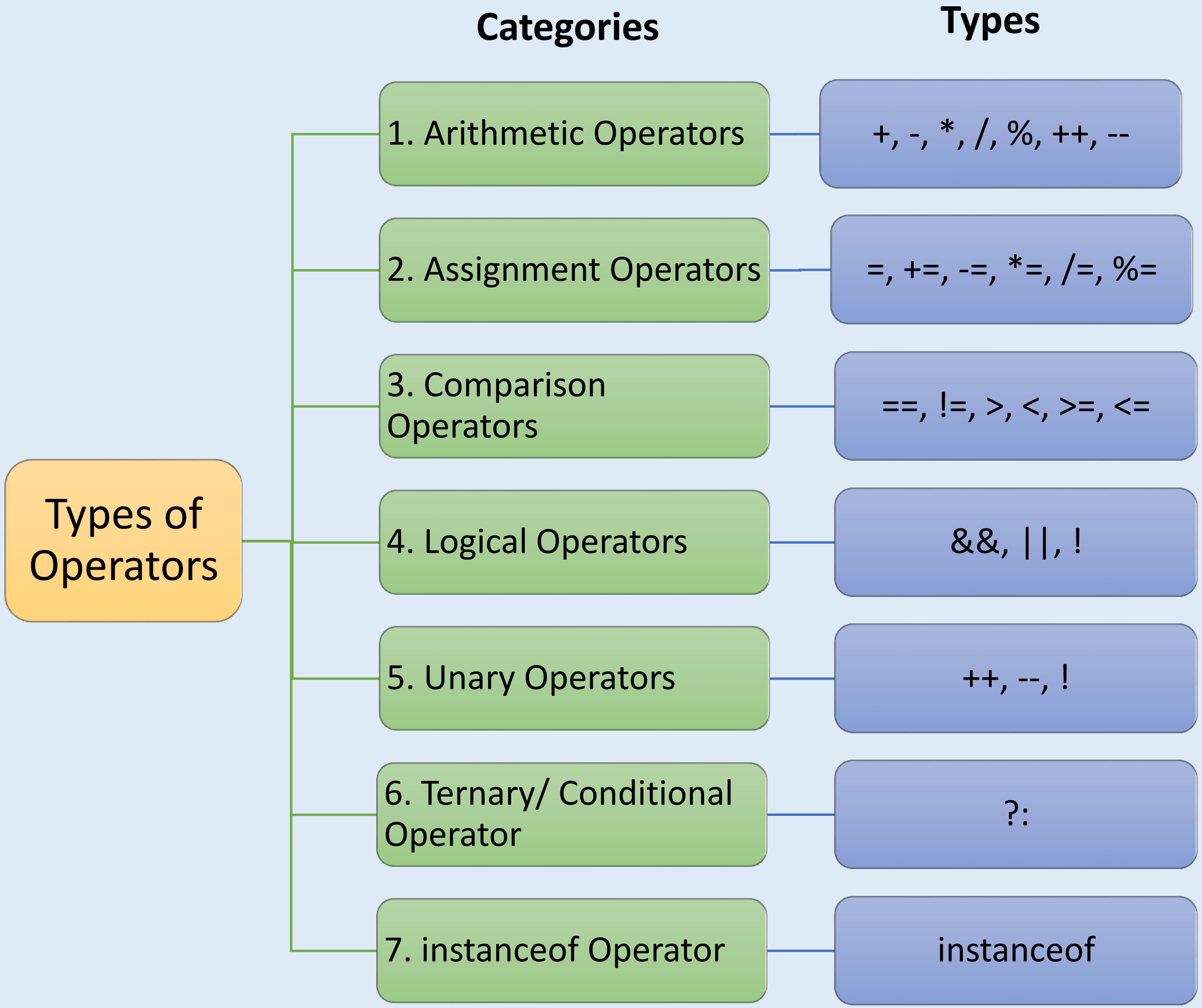
Q. What is instanceOf Operator?

Back to main index

# Q. What are Operators? What the types of operators in Java?

- Operators are **symbols or keywords** used to perform operations on operands.





Types of Operators

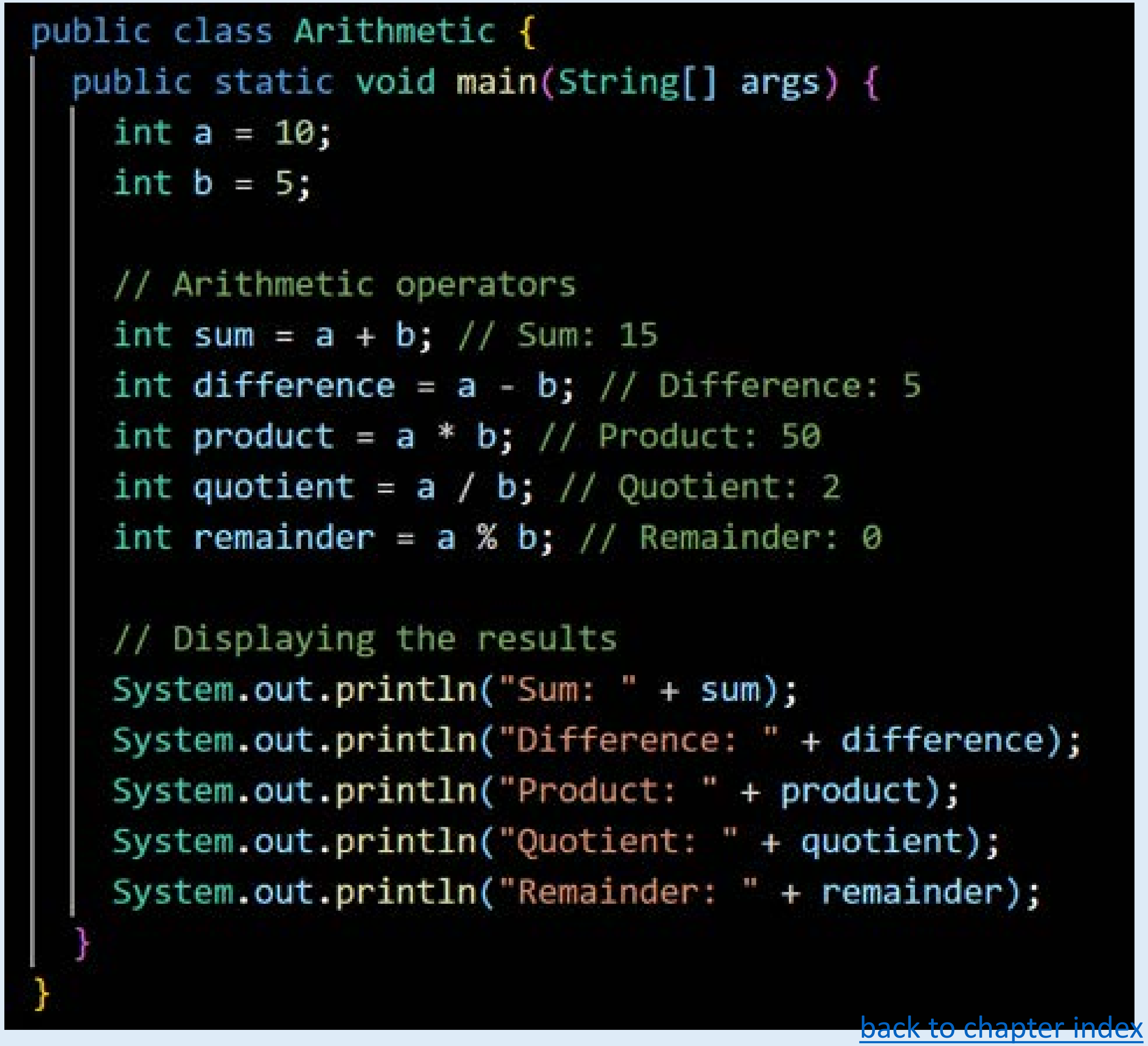
| Categories | Types |
|---|---|
| 1. Arithmetic Operators | +, -, *, /, %, ++, -- |
| 2. Assignment Operators | =, +=, -=, *=, /=, %= |
| 3. Comparison Operators | ==, !=, >, <, >=, <= |
| 4. Logical Operators | &&, \|\|, ! |
| 5. Unary Operators | ++, --, ! |
| 6. Ternary/ Conditional Operator | ?: |
| 7. instanceof Operator | instanceof |

back to chapter index

# Q. What are Arithmetic Operators?

- Arithmetic Operators perform basic arithmetic operations like addition (+), subtraction (-) etc.

| | |
|---|---|
| Types of Operators | Arithmetic Operators |
| | Assignment Operators |
| | Comparison Operators |
| | Logical Operators |
| | Unary Operators |
| | Ternary/ Conditional Operator |
| | instanceof Operator |



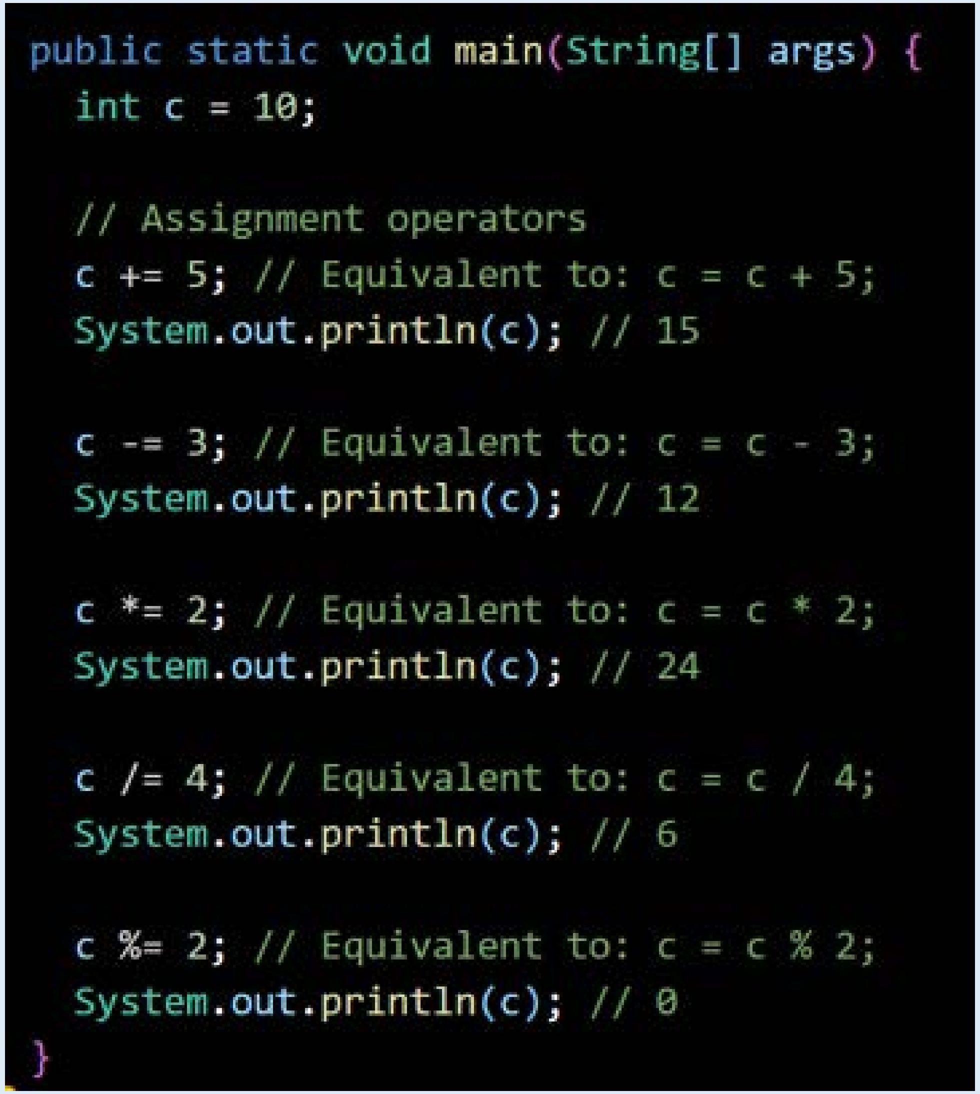
```java
public class Arithmetic {
  public static void main(String[] args) {
    int a = 10;
    int b = 5;

    // Arithmetic operators
    int sum = a + b; // Sum: 15
    int difference = a - b; // Difference: 5
    int product = a * b; // Product: 50
    int quotient = a / b; // Quotient: 2
    int remainder = a % b; // Remainder: 0

    // Displaying the results
    System.out.println("Sum: " + sum);
    System.out.println("Difference: " + difference);
    System.out.println("Product: " + product);
    System.out.println("Quotient: " + quotient);
    System.out.println("Remainder: " + remainder);
  }
}
```

back to chapter index

# Q. What are Assignment Operators?

- Assignment Operators assign values to variables. For example: =, +=, -=, *=, etc.

| | |
|---|---|
| Types of Operators | Arithmetic Operators |
| | Assignment Operators |
| | Comparison Operators |
| | Logical Operators |
| | Unary Operators |
| | Ternary/ Conditional Operator |
| | instanceof Operator |



```
public static void main(String[] args) {
  int c = 10;

  // Assignment operators
  c += 5; // Equivalent to: c = c + 5;
  System.out.println(c); // 15

  c -= 3; // Equivalent to: c = c - 3;
  System.out.println(c); // 12

  c *= 2; // Equivalent to: c = c * 2;
  System.out.println(c); // 24

  c /= 4; // Equivalent to: c = c / 4;
  System.out.println(c); // 6

  c %= 2; // Equivalent to: c = c % 2;
  System.out.println(c); // 0
}
```

back to chapter index

# Q. What are Comparison Operators? When to use them?

- Comparison Operators compare values and return boolean results. For example, ==, !=, >, <, >=, <=. Mostly used to evaluate conditions in control statements.

| | |
|---|---|
| Types of Operators | Arithmetic Operators |
| | Assignment Operators |
| | Comparison Operators |
| | Logical Operators |
| | Unary Operators |
| | Ternary/ Conditional Operator |
| | instanceof Operator |

```
public static void main() {
  int x = 10;
  int y = 5;
  // Comparison operators
  boolean isEqual = x == y;
  System.out.println(isEqual); // false

  boolean isNotEqual = x != y;
  System.out.println(isNotEqual); // true

  boolean isGreater = x > y;
  System.out.println(isGreater); // true

  boolean isLess = x < y;
  System.out.println(isLess); // false

  boolean isGreaterOrEqual = x >= y;
  System.out.println(isGreaterOrEqual); // true

  boolean isLessOrEqual = x <= y;
  System.out.println(isLessOrEqual); // false
}
```

back to chapter index

# Q. What are Logical Operators? When to use them?

- Logical Operators multiple combine boolean expressions and used for decision-making. For example, &&, ||, !, etc.

| | |
|---|---|
| Types of Operators | Arithmetic Operators |
| | Assignment Operators |
| | Comparison Operators |
| | Logical Operators |
| | Unary Operators |
| | Ternary/ Conditional Operator |
| | instanceof Operator |

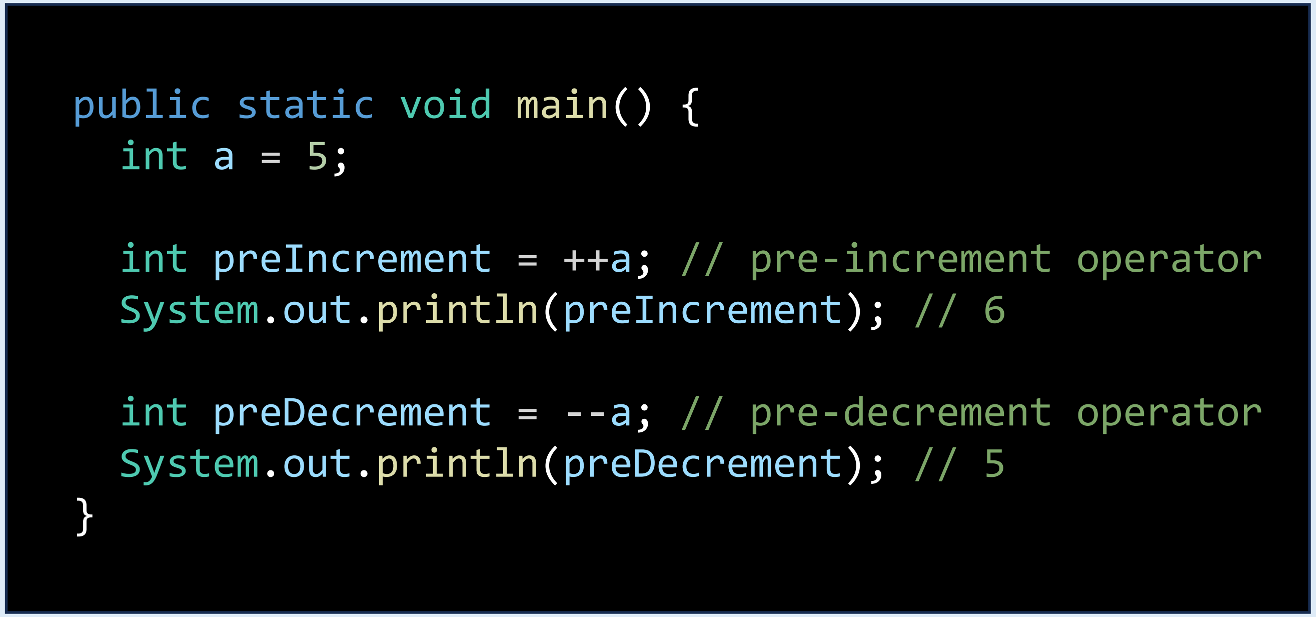
```java
public static void main() {
    boolean x = true;
    boolean y = false;

    // Logical AND
    boolean andResult1 = x && y;
    System.out.println(andResult1); // false

    boolean andResult2 = x && ("abc" == "abc");
    System.out.println(andResult2); // true

    // logical OR
    boolean orResult1 = x || y;
    System.out.println(orResult1); // true

    boolean orResult2 = ("abc" != "abc") || y;
    System.out.println(orResult2); // false
  }
```

back to chapter index

# Q. What are Unary Operators?

- Unary Operators operate on a **single operand**. For example, ++, --,  etc.

| | |
|---|---|
| Types of Operators | Arithmetic Operators |
| | Assignment Operators |
| | Comparison Operators |
| | Logical Operators |
| | Unary Operators |
| | Ternary/ Conditional Operator |
| | instanceof Operator |

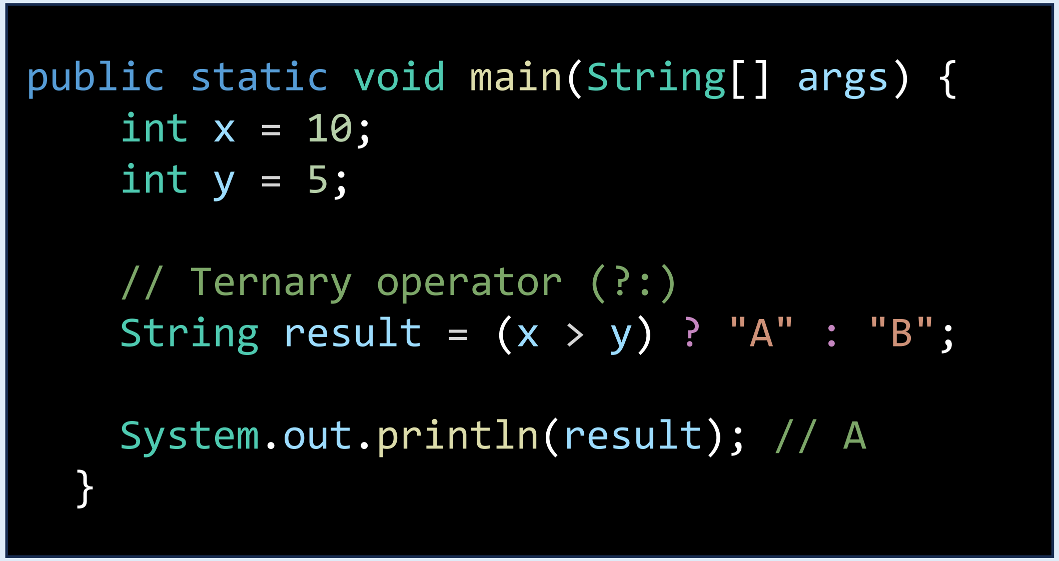


```java
public static void main() {
  int a = 5;

  int preIncrement = ++a; // pre-increment operator
  System.out.println(preIncrement); // 6

  int preDecrement = --a; // pre-decrement operator
  System.out.println(preDecrement); // 5
}
```

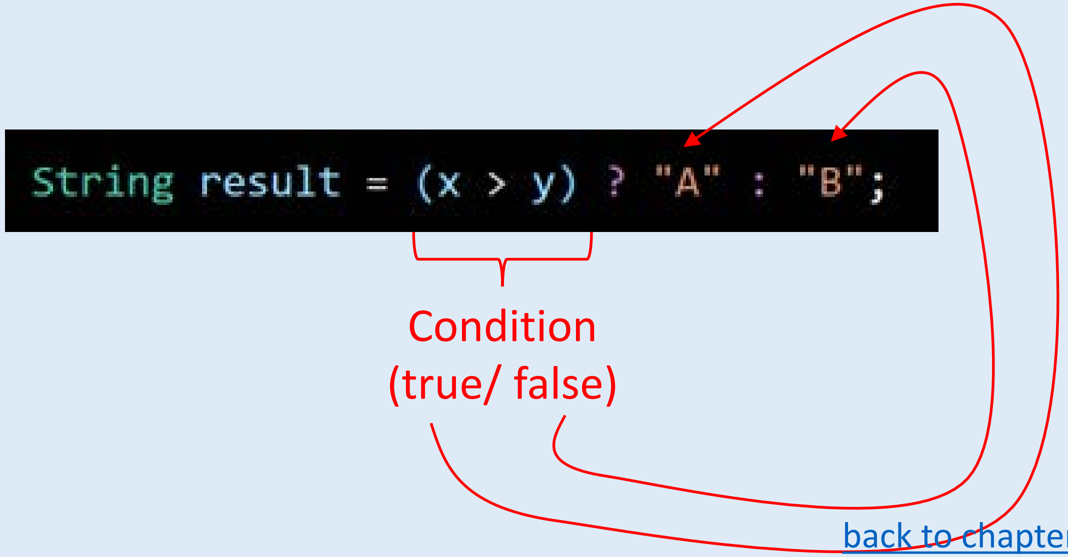
back to chapter index

# Q. What are Ternary (Conditional) Operators? V. IMP.

- Ternary Operator (Conditional Operator) (?:) evaluate a boolean expression and return one of two values based on the result.
- It is mostly used for conditional assignment.
- It is called ternary operator because it operates on three operands.



```
public static void main(String[] args) {
    int x = 10;
    int y = 5;

    // Ternary operator (?:)
    String result = (x > y) ? "A" : "B";

    System.out.println(result); // A
  }
```



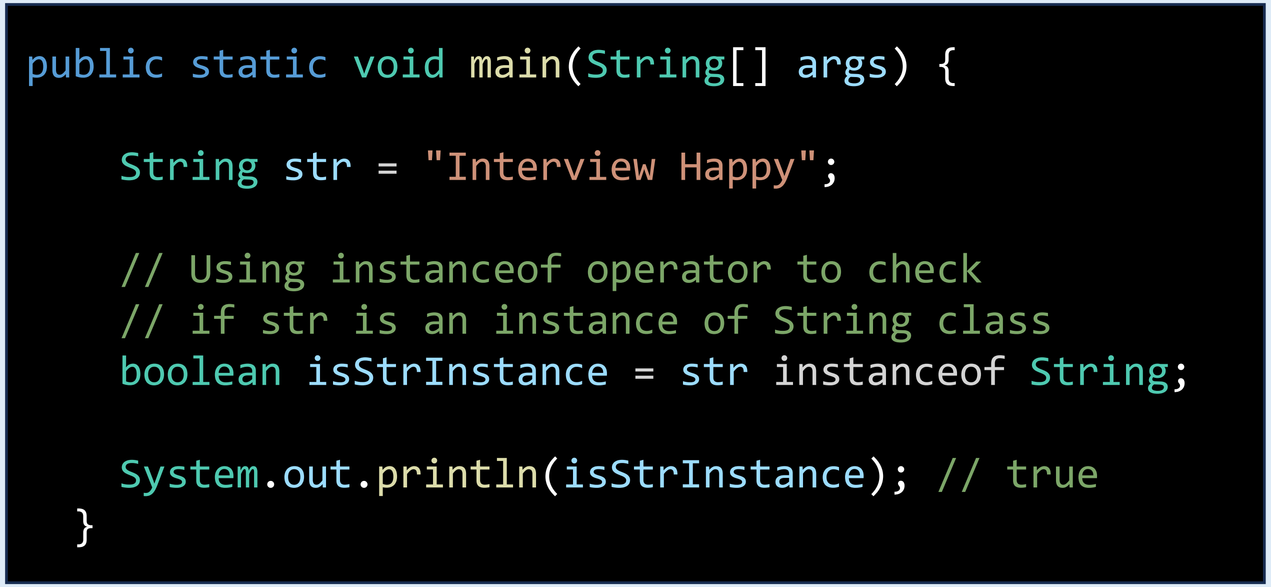
```
String result = (x > y) ? "A" : "B";
```

back to chapter index

# Q. What is instanceOf Operator?

- The instanceof operator is used to check if an object is an instance of a specific class or a type.

| | |
|---|---|
| Types of Operators | Arithmetic Operators |
| | Assignment Operators |
| | Comparison Operators |
| | Logical Operators |
| | Unary Operators |
| | Ternary/ Conditional Operator |
| | instanceof Operator |



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

    String str = "Interview Happy";

    // Using instanceof operator to check
    // if str is an instance of String class
    boolean isStrInstance = str instanceof String;

    System.out.println(isStrInstance); // true
  }
```

back to chapter index

# 4. Control statements - Basics

Q. What are control statements in Java? **V. IMP.**

Q. What are conditional statements? What is if-elseif-else condition?

Q. What are looping statements in Java? What is while loop?

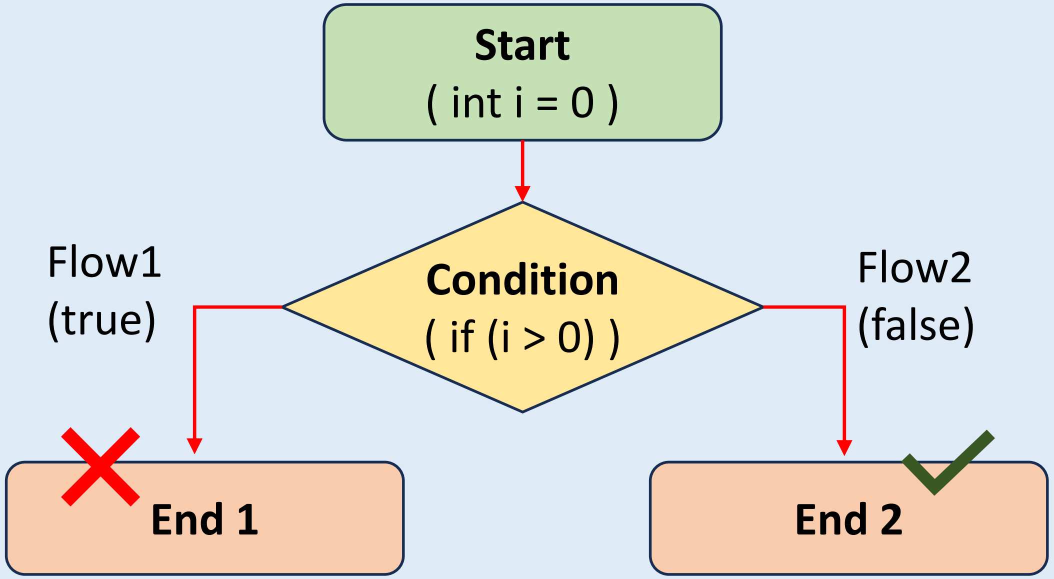
Q. What is the difference between while loop and for loop?

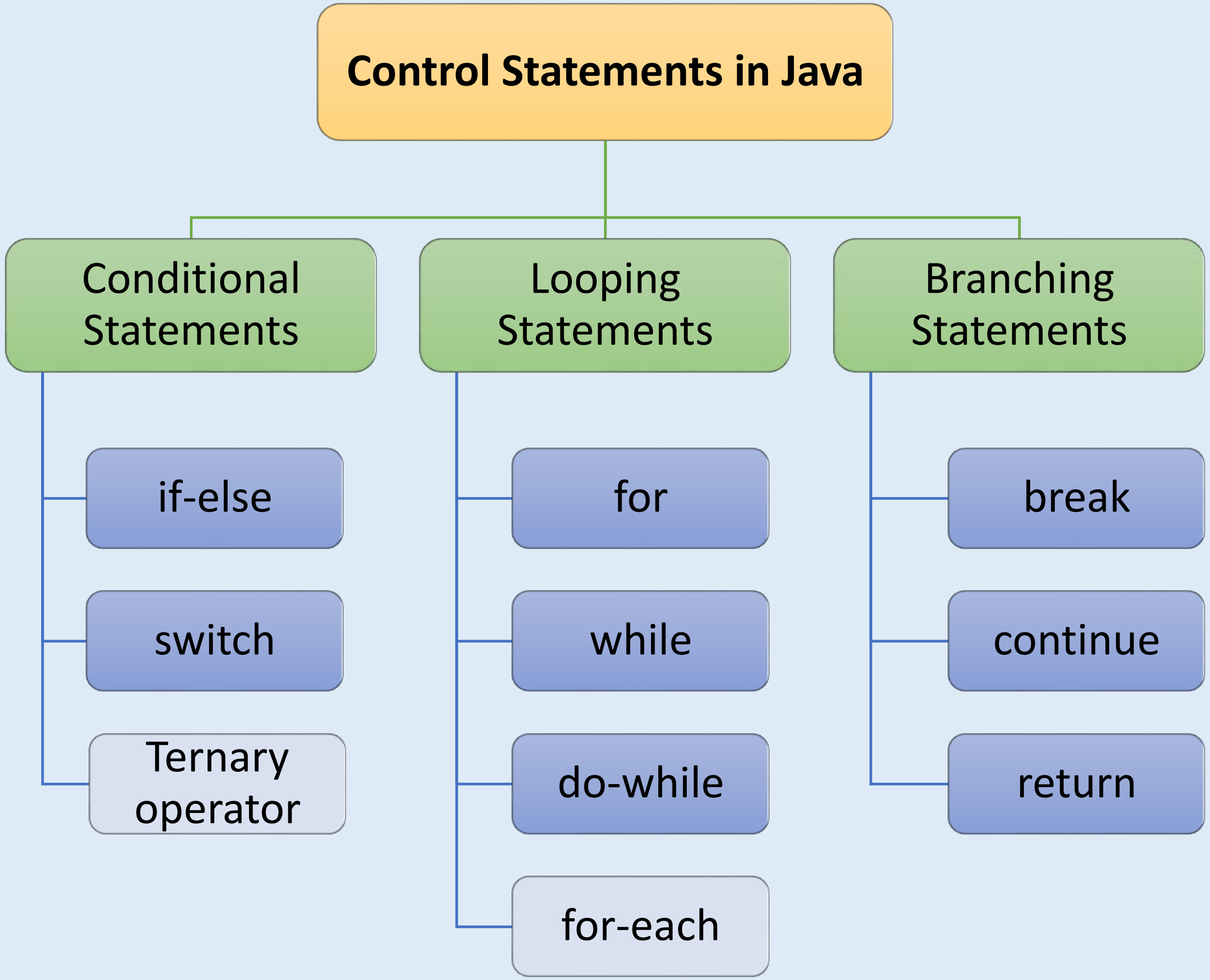
Q. What is the difference between break and continue statement? **V. IMP.**

Back to main index

# Q. What are control statements in Java? V. IMP.

- Control statements **manage the flow** of execution in a program.

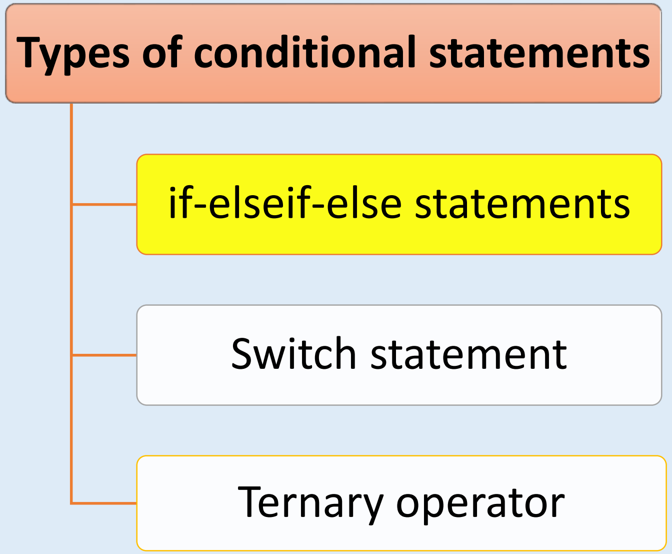




back to chapter index

# Q. What are conditional statements? What is if-elseif-else condition?

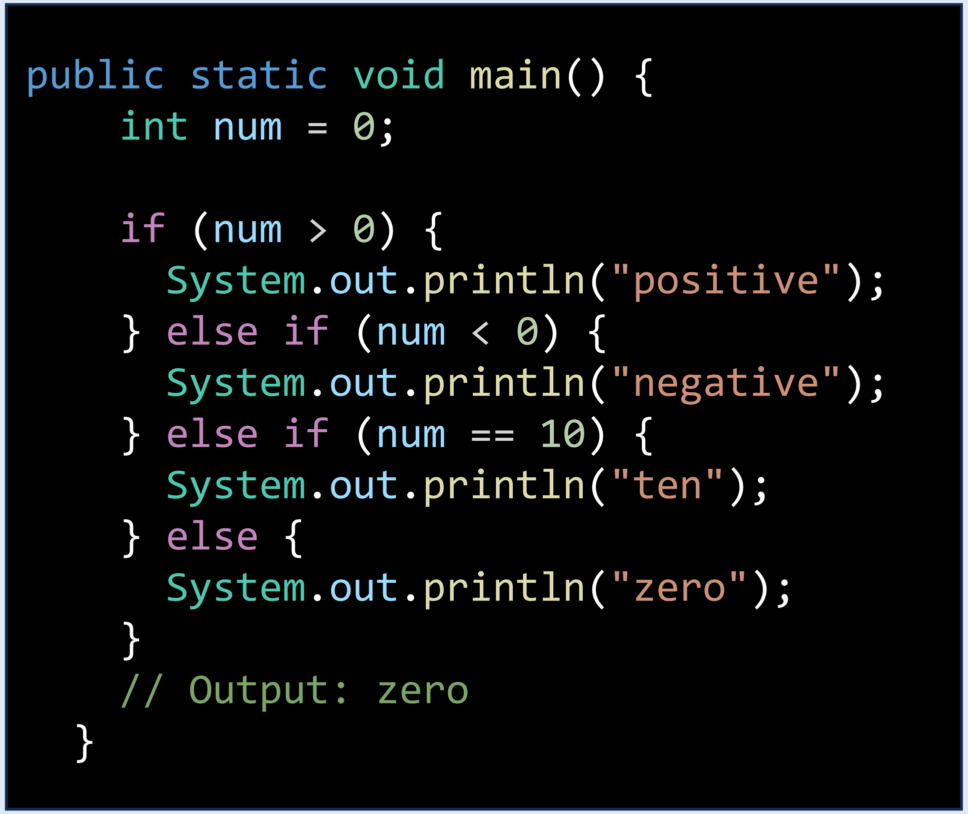
- Conditional statements helps in **decision-making** based on specified conditions.
- There can be multiple else-if blocks, but only one if and else block can be present.
- Only one block will be executed at a time.


Types of conditional statements

if-elseif-else statements

Switch statement

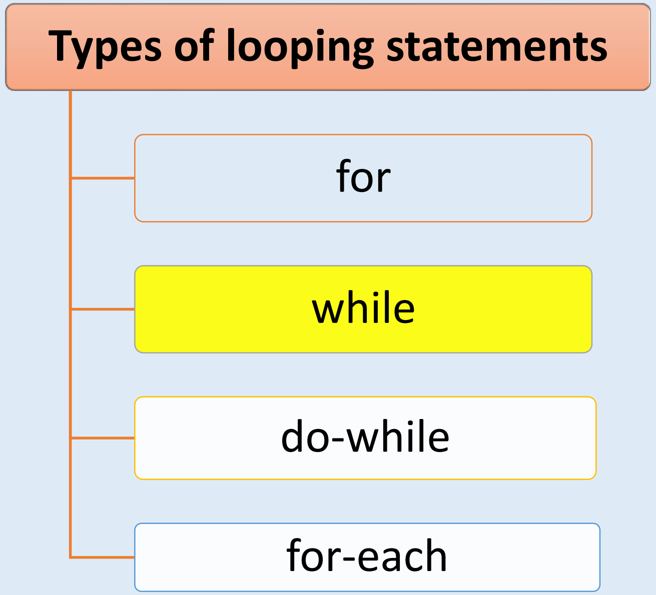
Ternary operator


```
public static void main() {
    int num = 0;

    if (num > 0) {
      System.out.println("positive");
    } else if (num < 0) {
      System.out.println("negative");
    } else if (num == 10) {
      System.out.println("ten");
    } else {
      System.out.println("zero");
    }
    // Output: zero
  }
```

back to chapter index

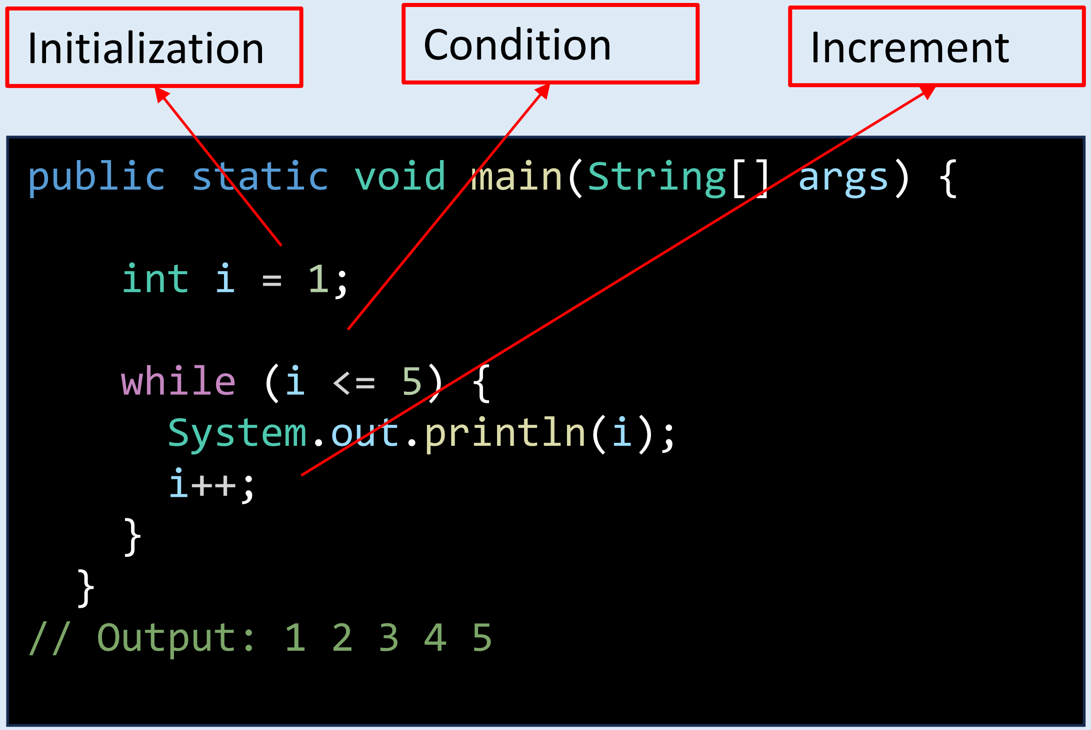
# Q. What are looping statements in Java? What is while loop?

- Looping statements are used to execute a block of **code repeatedly** based on a condition.
- A while loop iterate(repeat) a block of code while a **certain condition is true**.



**Types of looping statements**

- for
- while
- do-while
- for-each



Initialization

Condition

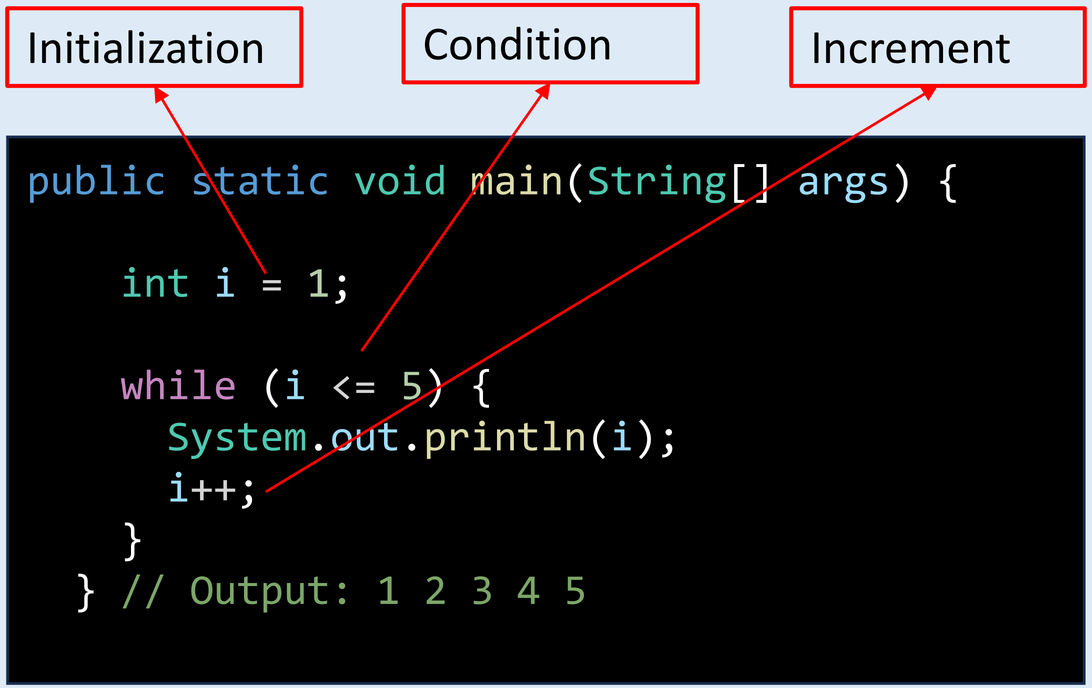
Increment

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

    int i = 1;

    while (i <= 5) {
      System.out.println(i);
      i++;
    }
  }
// Output: 1 2 3 4 5
```

back to chapter index

# Q. What is the difference between while loop and for loop?

- A while loop iterate(repeat) a block of code while a certain **condition is true**.



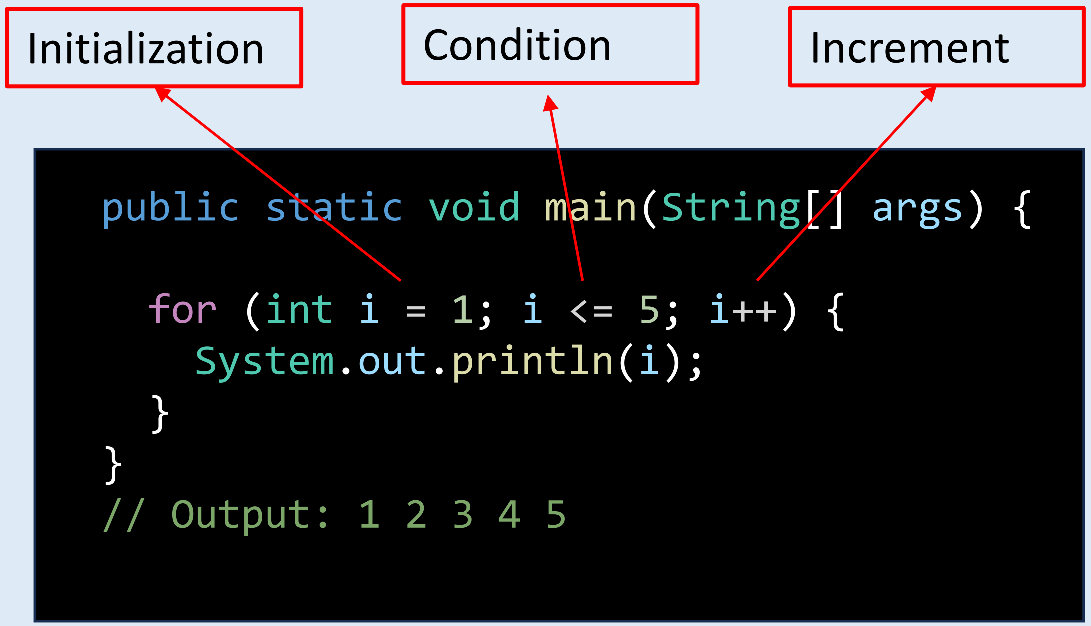
Initialization

Condition

Increment

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

    int i = 1;

    while (i <= 5) {
      System.out.println(i);
      i++;
    }
  } // Output: 1 2 3 4 5
```

- A for loop repeat iterate(repeat) a block of code a certain **number of times**.



Initialization

Condition

Increment

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

  for (int i = 1; i <= 5; i++) {
    System.out.println(i);
  }
}
// Output: 1 2 3 4 5
```

back to chapter index

# Q. What is the difference between break and continue statement? V. IMP.

- The break statement is used to **terminate** the loop completely.

```
public class BreakEx {

  public static void main(String[] args) {
    for (int i = 1; i <= 5; i++) {
      if (i == 3) {
        break; // Exit the loop
      }
      System.out.println(i);
    }
  }
} // Output: 1 2
```

- The continue statement is used to **skip the current iteration** of the loop and move on to the next iteration of the loop.

```
public class ContinueEx {

  public static void main(String[] args) {
    for (int i = 1; i <= 5; i++) {
      if (i == 3) {
        continue; // Skip rest of the loop
      }
      System.out.println(i);
    }
  }
} // Output: 1 2 4 5
```

back to chapter index

# 5. Control statements - Advanced

Q. What is the difference between while loop and do-while loop? **V. IMP.**

Q. How to decide which loop(for, while, do-while) to use in real applications? **V. IMP.**

Q. What is the difference btw for loop and for-each loop? When to use for-each loop?

Q. What is switch statement?

Q. When to use which type of conditional statements in real applications? **V. IMP.**

Back to main index

# Q. What is the difference between while loop and do-while loop? V. IMP.

- A while loop iterate a block of code while a certain **condition is true**.

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

    int i = 1; // Initialization
    while (i <= 5) {// Condition
      System.out.println(i);
      i++; // Increment
    }
  } // Output: 1 2 3 4 5
```

- A do-while loop is like the while loop only, except that the block of code is **executed at least once**, even if the condition is false.

```
public static void main(String[] args) {
  int i = 5; // Initialization

  do {
    System.out.println(i);
    i++; // Increment
  } while (i < 4); // Condition
} // Output: 5
```

back to chapter index

# Q. How to decide which loop(for, while, do-while) to use in real applications? V. IMP.

- Use for loop when you have to initialize a variable, specify a condition, and define an increment **all in one line**, which can make your code more compact and readable.

```
for (int i = 1; i <= 5; i++) {
  System.out.println(i);
}
```

- Use while loop when have **only condition**, without initialization and increment.

```
int i = 1; // Initialization

while (i <= 5) {// Condition
  System.out.println(i);
  i++; // Increment
}
```

```
while("a" == "a") // Only
//Condition
{
  System.out.println("H");
  break;
}
```

- Use do-while when you have to iterate the loop at least one time.

```
int i = 5; // Initialization

do {
  System.out.println(i);
  i++; // Increment
} while (i < 4); // Condition
```

back to chapter index

# Q. What is the difference btw for loop and for-each loop? When to use for-each loop?

- A for loop repeat iterate(repeat) a block of code a certain **number of times**.

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

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

  // Using a traditional for loop
  for (int i = 0; i < numbers.length; i++) {
      int number = numbers[i];
      System.out.println(number);
  }

}
// Output: 1 2 3 4 5
```

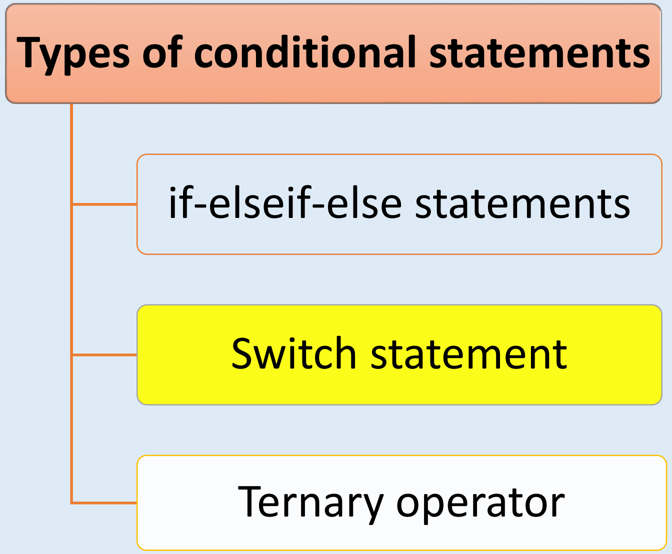
- The enhanced for loop(for-each loop), is used to iterate over elements in an **array or a collection** without the need for an explicit loop counter.

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

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

    for (int number : numbers) {
      System.out.println(number);
    }
  } // Output: 1 2 3 4 5
```

back to chapter index

# Q. What is switch statement?

- The switch statement executes the code block corresponding to the matching case label.
- Use break statements to exit the switch block.
- The default case is optional and executes when none of the case labels match the expression.



Types of conditional statements

if-elseif-else statements

Switch statement

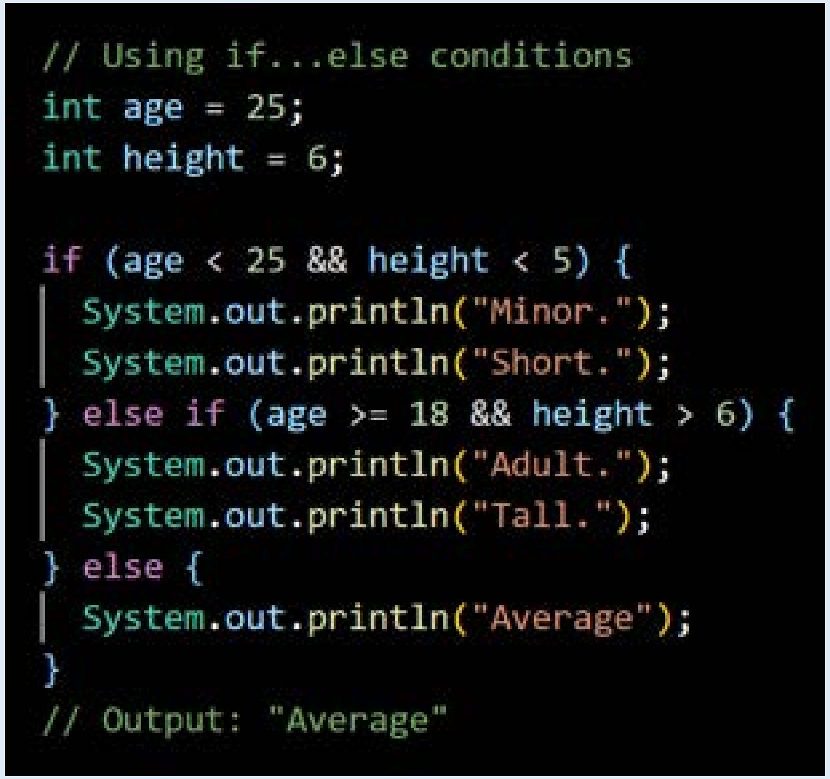
Ternary operator

```
public static void main() {
    int priority = 2;
    String result;

    // Switch statement
    switch (priority) {
      case 1:
        result = "High";
        break;
      case 2:
        result = "Medium";
        break;
      default:
        result = "Low";
        break;
    }
    System.out.println(result);
    // Output: Medium
  }
```

back to chapter index

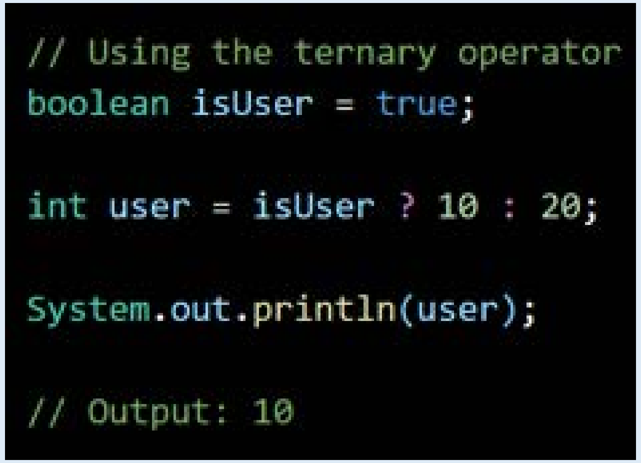
# Q. When to use which type of conditional statements in real applications? V. IMP.

- **If…else** : for complex, different & multiline execution.
- **Benefit**: Cover all scenarios.



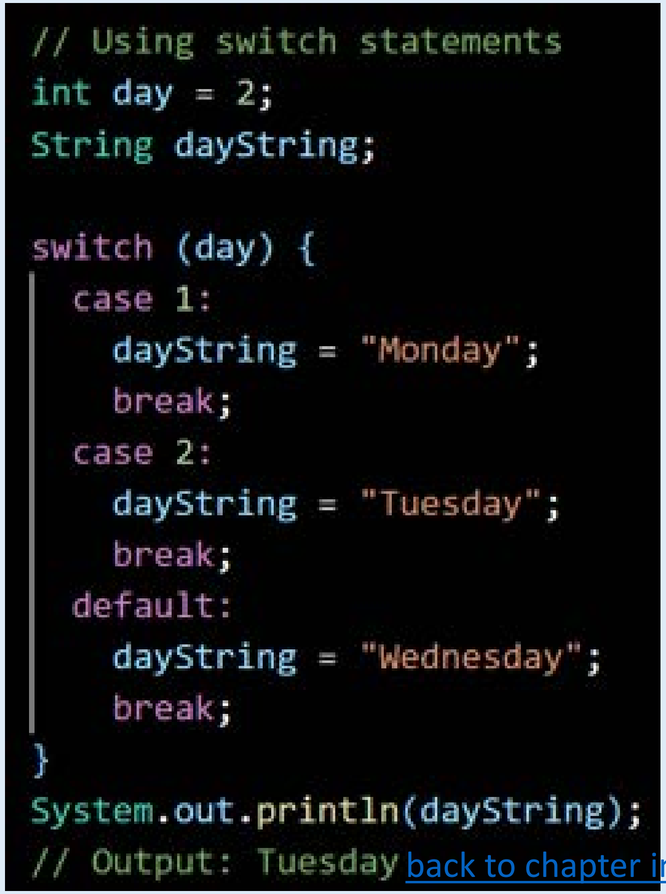
```java
// Using if...else conditions
int age = 25;
int height = 6;

if (age < 25 && height < 5) {
  System.out.println("Minor.");
  System.out.println("Short.");
} else if (age >= 18 && height > 6) {
  System.out.println("Adult.");
  System.out.println("Tall.");
} else {
  System.out.println("Average");
}
// Output: "Average"
```

- **Ternary operators** : for single conditions & single value evaluations.
- **Benefit**: Short one line syntax.



```java
// Using the ternary operator
boolean isUser = true;

int user = isUser ? 10 : 20;

System.out.println(user);

// Output: 10
```

- **Switch case**: For same left side values.
- **Benefit**: More structured code.



```java
// Using switch statements
int day = 2;
String dayString;

switch (day) {
  case 1:
    dayString = "Monday";
    break;
  case 2:
    dayString = "Tuesday";
    break;
  default:
    dayString = "Wednesday";
    break;
}
System.out.println(dayString);
// Output: Tuesday
```

back to chapter index

# 6. String & StringBuilder

Q. What is StringBuilder? What is the difference between String and StringBuilder? **V. IMP.**

Q. When to use String and when to use StringBuilder in real applications?

Q. What is the concept of string pool in Java?

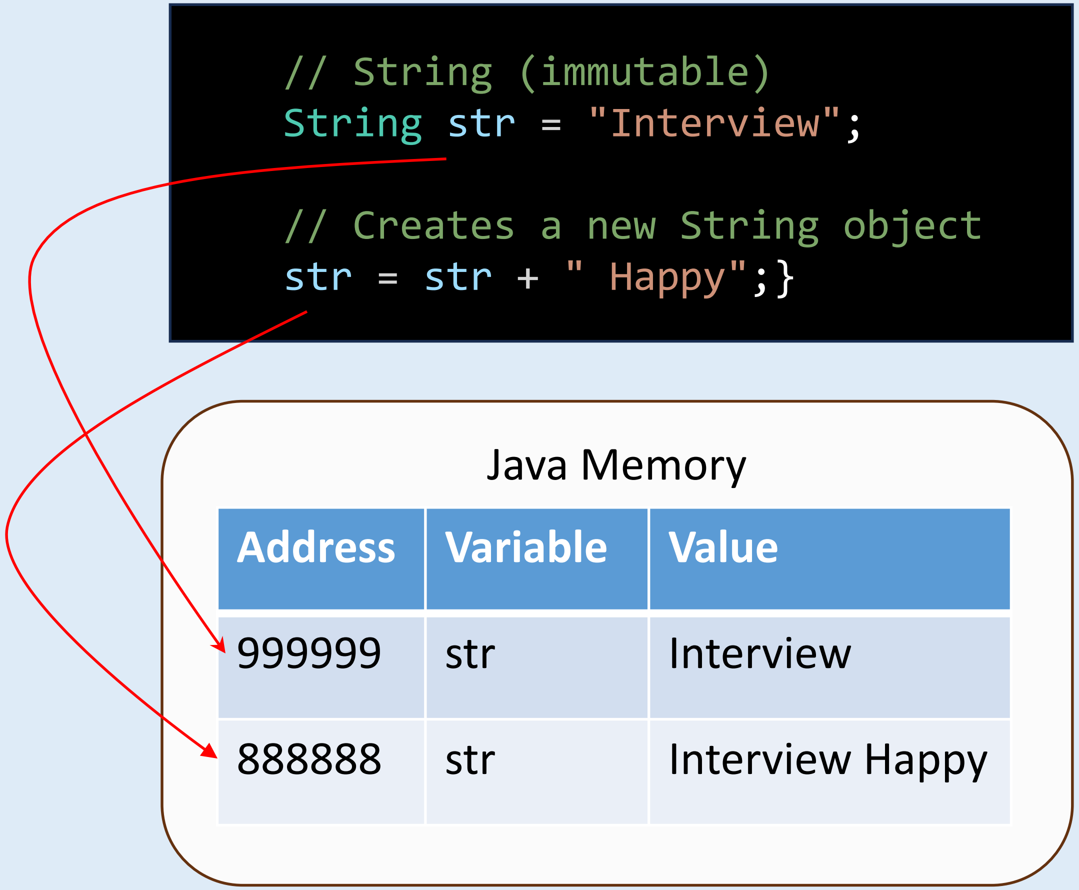
Q. What is the difference between == and equals() method for comparing strings? **V. IMP.**

Q. What are the important methods of String class?

Back to main index

# Q. What is StringBuilder? What is the difference between String and StringBuilder? V. IMP.

- String is **immutable.** It means if you defined one string then you couldn't modify it. Every time you will assign some value to it, a new string is created.

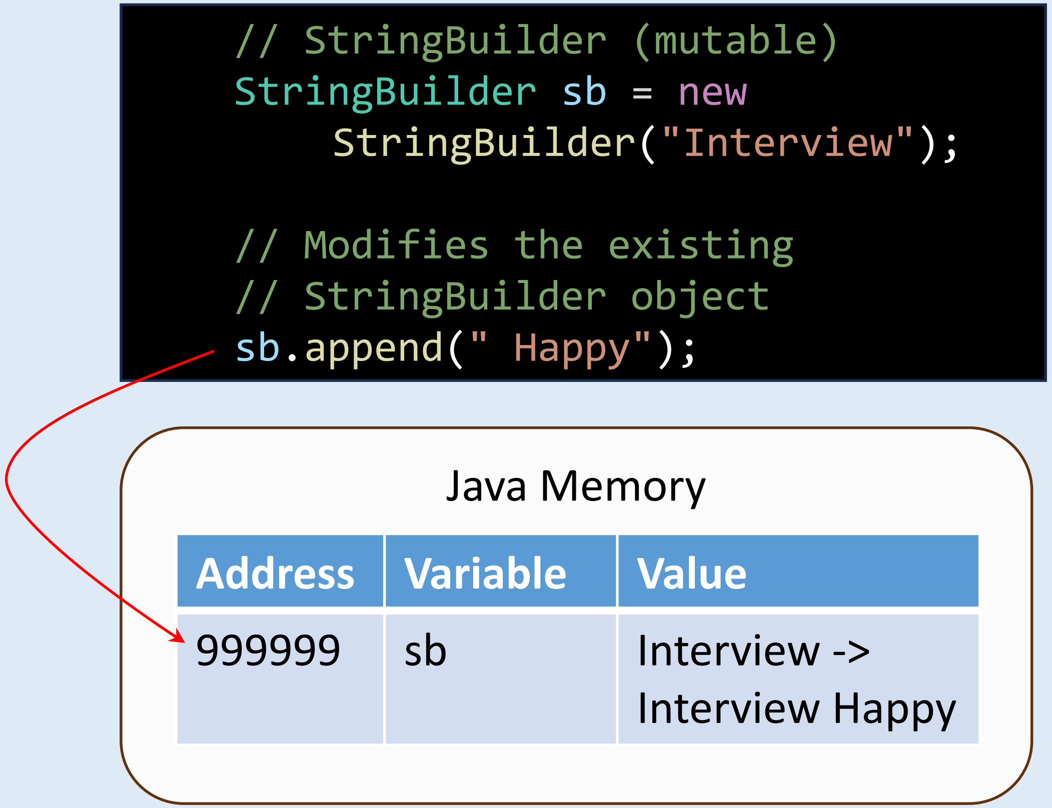


```
// String (immutable)
String str = "Interview";

// Creates a new String object
str = str + " Happy";}
```

Java Memory

| Address | Variable | Value |
|---|---|---|
| 999999 | str | Interview |
| 888888 | str | Interview Happy |

- StringBuilder is **mutable.** It means if you defined one string then you can modify it and a new string is not created.



```
// StringBuilder (mutable)
StringBuilder sb = new
    StringBuilder("Interview");

// Modifies the existing
// StringBuilder object
sb.append(" Happy");
```

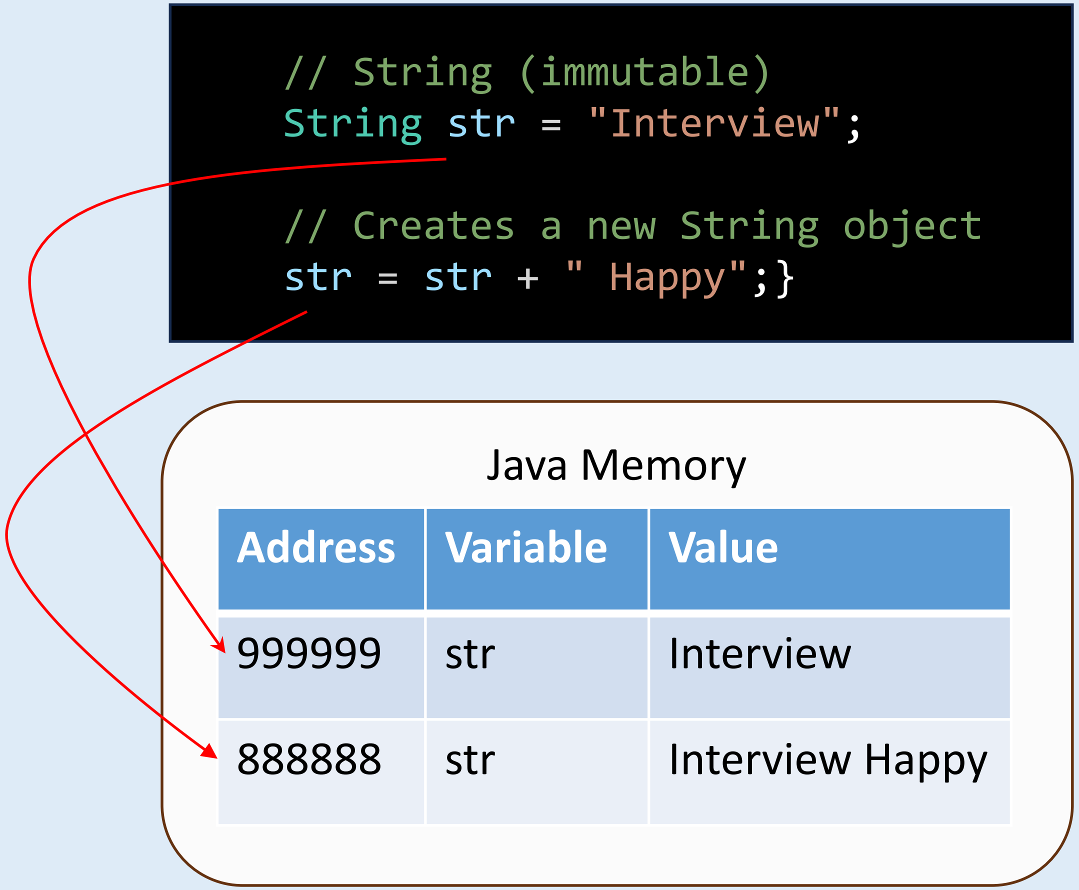
Java Memory

| Address | Variable | Value |
|---|---|---|
| 999999 | sb | Interview -><br>Interview Happy |

back to chapter index

# Q. When to use String and when to use StringBuilder in real applications?

- Use String for **constant values** because Strings are light weight and therefore improve performance.

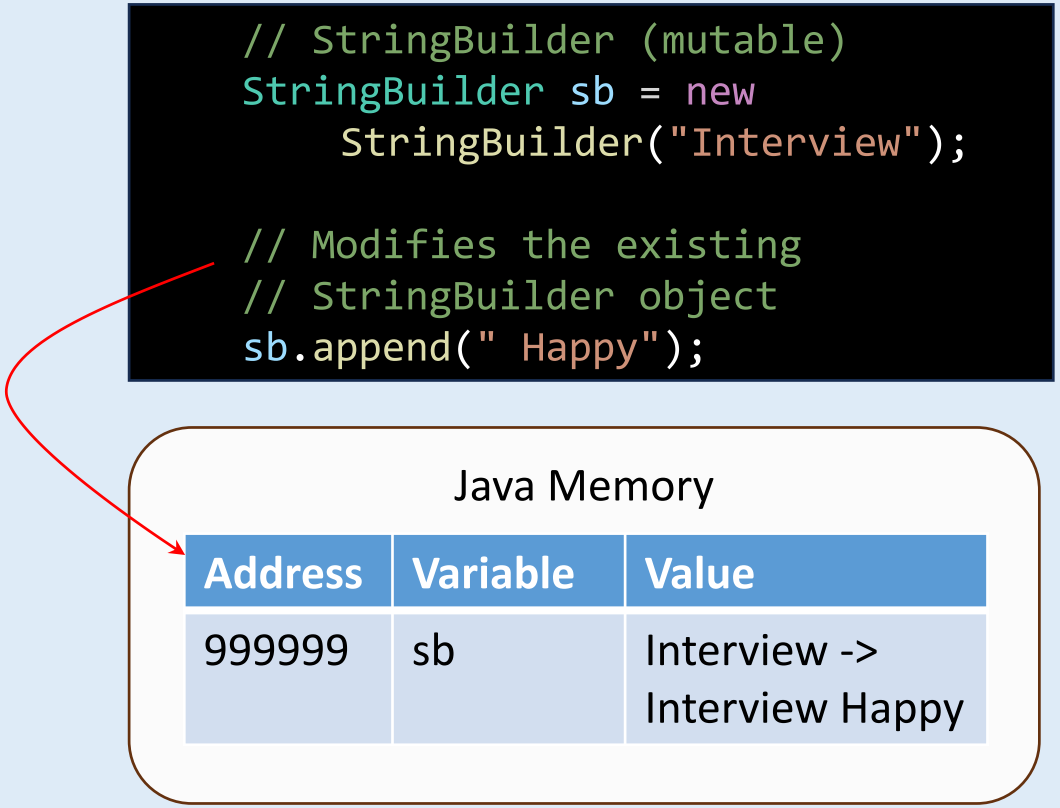


```java
// String (immutable)
String str = "Interview";

// Creates a new String object
str = str + " Happy";}
```

Java Memory

| Address | Variable | Value |
|---|---|---|
| 999999 | str | Interview |
| 888888 | str | Interview Happy |

- Use StringBuilder when concatenating or **modifying strings frequently.**



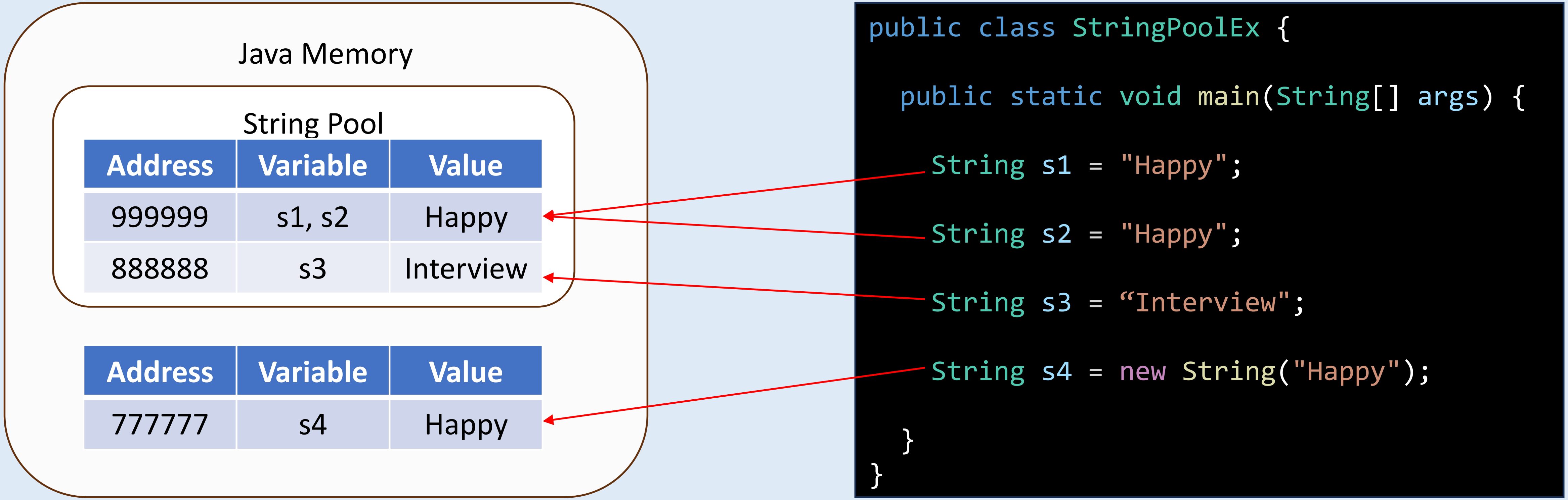
```java
// StringBuilder (mutable)
StringBuilder sb = new
    StringBuilder("Interview");

// Modifies the existing
// StringBuilder object
sb.append(" Happy");
```

Java Memory

| Address | Variable | Value |
|---|---|---|
| 999999 | sb | Interview -> Interview Happy |

back to chapter index

# Q. What is the concept of string pool in Java?



Java Memory

String Pool

| Address | Variable | Value |
|---|---|---|
| 999999 | s1, s2 | Happy |
| 888888 | s3 | Interview |

| Address | Variable | Value |
|---|---|---|
| 777777 | s4 | Happy |

```
public class StringPoolEx {

  public static void main(String[] args) {

    String s1 = "Happy";

    String s2 = "Happy";

    String s3 = “Interview";

    String s4 = new String("Happy");

  }
}
```

- The string constant pool in Java is a memory area where String literals are stored.

- When you use a String literal, Java checks the pool:

a. If the string is already there, Java returns its reference.

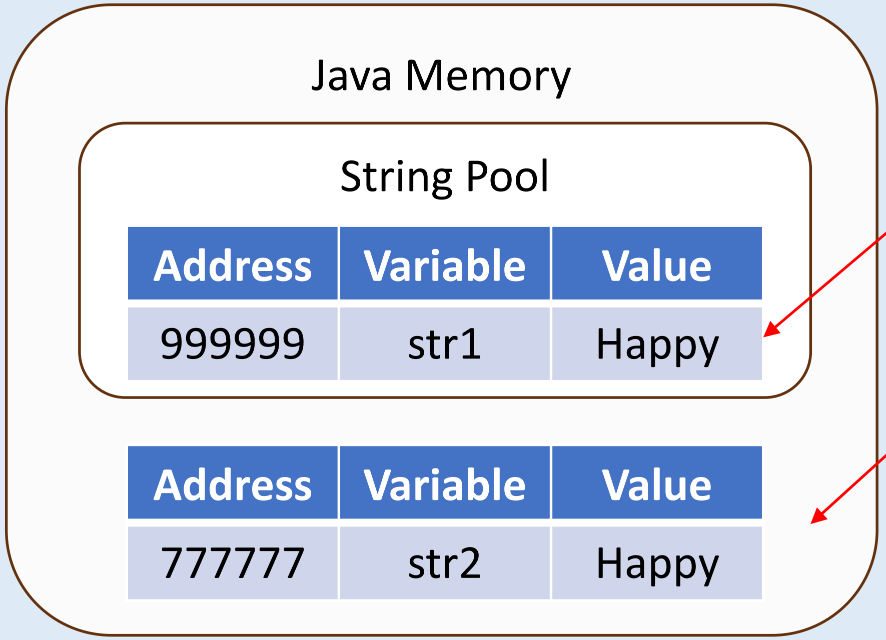
b. If not, Java creates a new String object in the pool and returns its reference.

- This helps conserve memory by reusing existing strings.

back to chapter index

# Q. What is the difference between == and equals() method for comparing strings?

- == compares references (memory addresses) of objects.
- equals() compares the actual contents (characters) of objects.



Java Memory

String Pool

| Address | Variable | Value |
|---|---|---|
| 999999 | str1 | Happy |

| Address | Variable | Value |
|---|---|---|
| 777777 | str2 | Happy |

```java
public class StringComparisonEx {

  public static void main(String[] args) {
    String str1 = "Happy";
    String str2 = new String("Happy");

    // Using == to compare string references
    // different objects in memory
    System.out.println(str1 == str2); // false

    // Using equals() to compare string contents
    System.out.println(str1.equals(str2)); // true
  }
}
```

back to chapter index

# Q. What are the important methods of String class?

```java
public class StringExample {

  public static void main(String[] args) {
    String str1 = "Interview";
    String str2 = "Happy";

    // 1.Length of the string
    int length = str1.length();
    System.out.println(length); // Output: 9

    // 2.Concatenation - joing two strings
    String result = str1.concat(", " + str2);
    System.out.println(result);
    // Output: Interview, Happy
```

```java
// 3. Substring - Retrieves substring from index 5
to 9
    String substring = result.substring(5, 9);
    System.out.println(substring); // Output: view

    // 4. Index of - // Finds index of character 'H'
    int index = result.indexOf('H');
    System.out.println(index); // Output: 11

    // 5. Equality - Checks if strings are equal
    boolean isEqual = str1.equals(str2);
    System.out.println(isEqual); // Output: false
  }
}
```

back to chapter index

# 7. Arrays

Q. What is an Array? Why we need array in real applications? **V. IMP.**

Q. How to declare and initialize an array? How to access array elements?

Q. What is the length property of an array?

Q. How do you iterate over an array in Java?

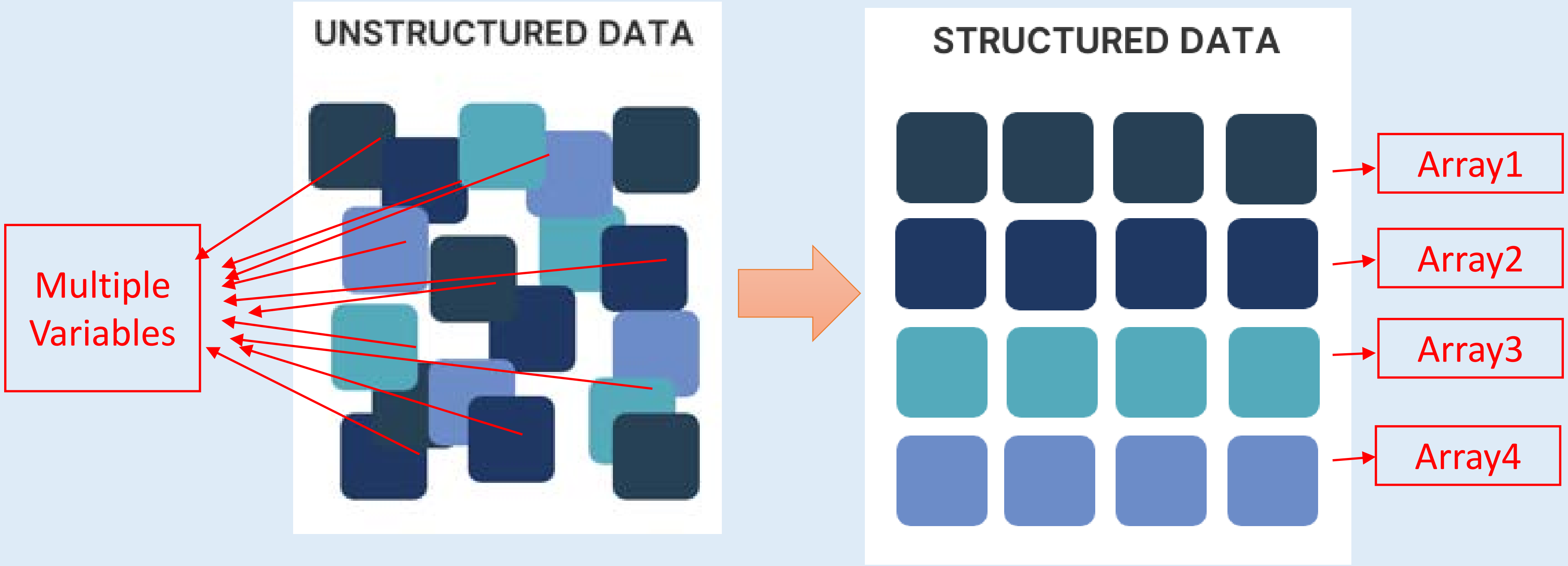
Back to main index

# Q. What is an Array? Why we need array in real applications? V. IMP.

- **Definition:** An array is a **data type** that allows you to **store multiple values** in a single variable.
- **Use:** Arrays are used to **store related data** in a structured way. Arrays also enables sequential access via indexes.

Array (fruits)

| Array Elements | apple | banana | orange |
|---|---|---|---|
| Array Index | 0 | 1 | 2 |



back to chapter index

# Q. How to declare and initialize an array? How to access array elements?

Array

| Array Elements | apple | banana | orange |
|---|---|---|---|
| Array Index | 0 | 1 | 2 |

```java
public class ArrayEx {

  public static void main(String[] args) {

    // Declare and initialize an array of String
    String[] fruits = { "apple", "banana", "orange" };

    //Access one element by passing index
    System.out.println(fruits[2]); // orange
  }
}
```

back to chapter index

# Q. What is the length property of an array?

- The length property of an array is used to determine the number of elements in the array.

```
public class ArrayLengthEx {

  public static void main(String[] args) {

    // Declare and initialize an array of String
    String[] fruits = { "apple", "banana", "orange" };

    // Print the length of the array
    System.out.println(fruits.length); // Output: 3
  }
}
```

back to chapter index

# Q. How do you iterate over an array in Java?

- Use loops to iterate over an array.

Array

| | | | |
|---|---|---|---|
| Array Elements | apple | banana | orange |
| Array Index | 0 | 1 | 2 |

```java
public class ArrayEx {

  public static void main(String[] args) {

    // Declare and initialize an array of String
    String[] fruits = { "apple", "banana", "orange" };

    // Iterate over an array
    for (int i = 0; i < fruits.length; i++) {
      System.out.println(fruits[i]);
    }
    // apple, banana, orange
  }
}
```

back to chapter index

# 8. OOPS - Classes, Objects & Package

Q. What is OOPS? What are the main concepts of OOPS? **V. IMP.**

Q. What are classes and objects? Why use them in applications? **V. IMP.**

Q. How to implement classes and objects in Java?

Q. What are the members of class? **V. IMP.**

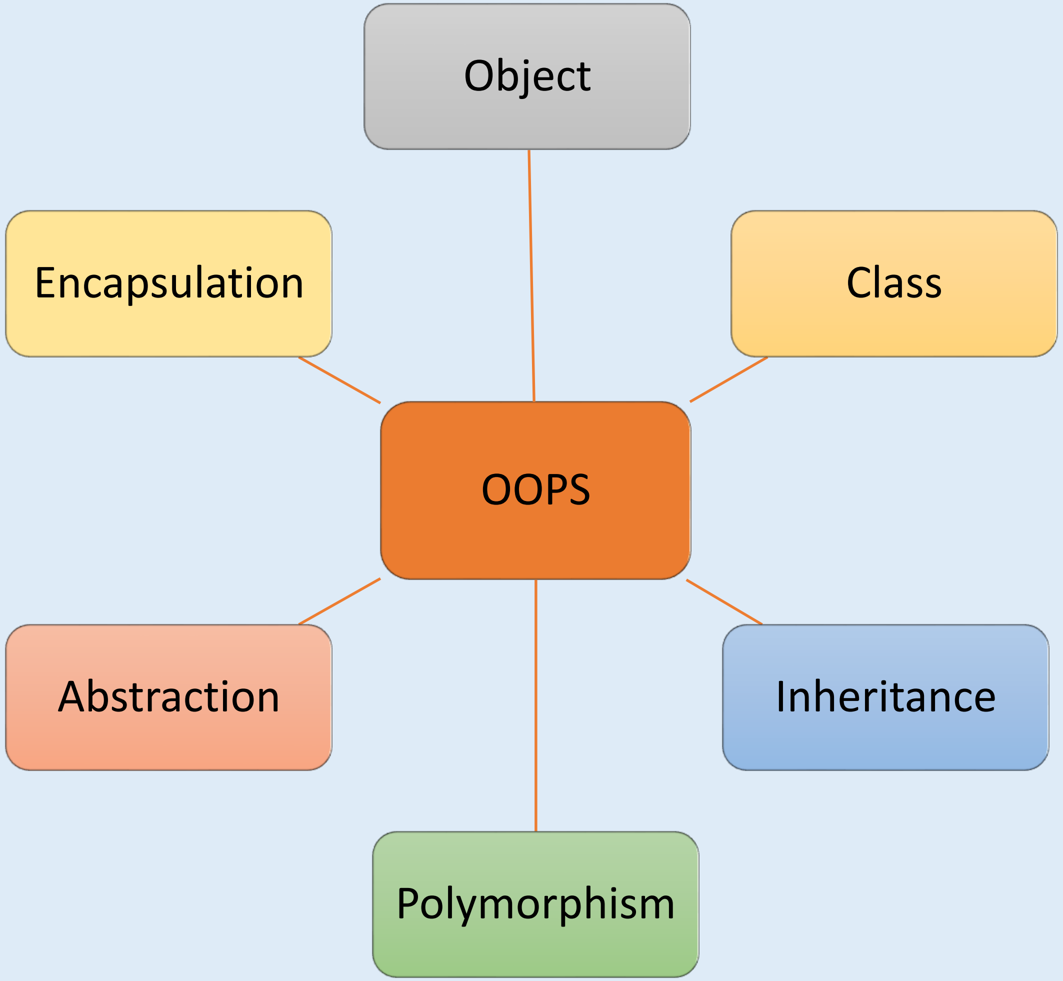
Q. What is the role and benefit of package in Java?

Back to main index

# Q. What is OOPS? What are the main concepts of OOPS? V. IMP.

- OOP stands for **Object-Oriented Programming**, which means OOPS is a way to create software around **objects**.

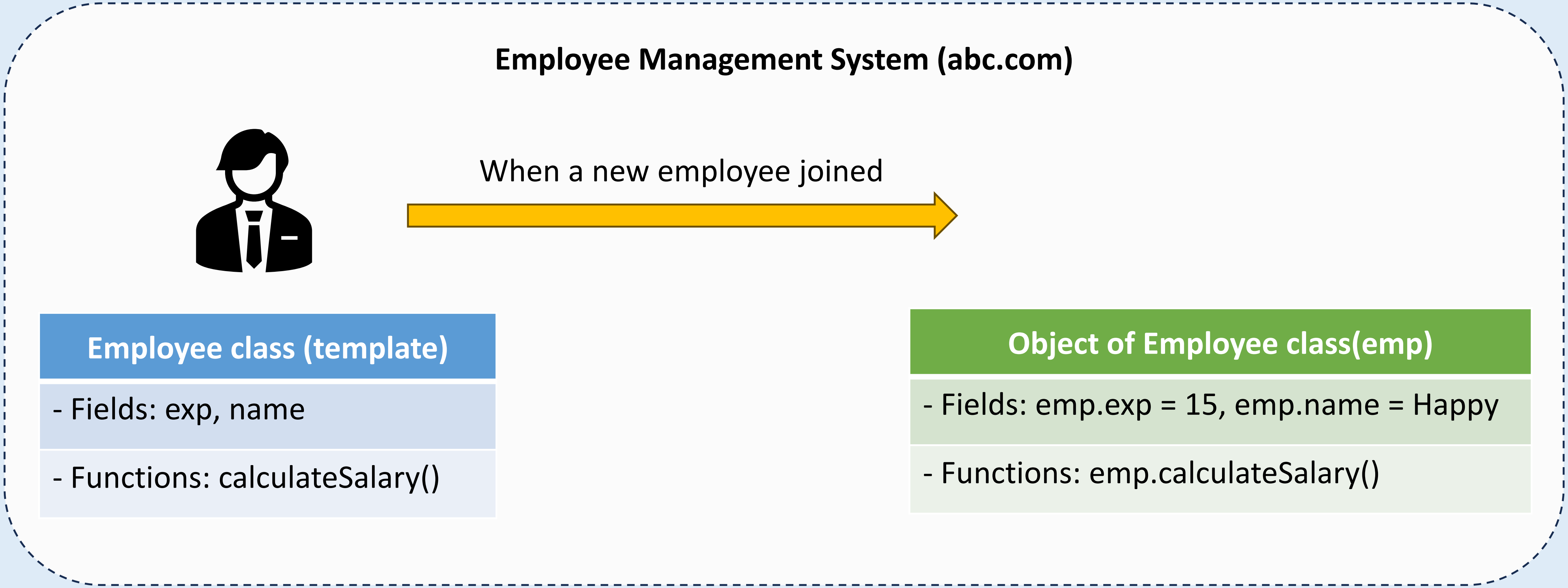
- OOPs provide a **clear structure** for the software's and web applications.



back to chapter index

# Q. What are classes and objects? Why use them in applications? V. IMP.

- A class is a **blueprint/ template** for creating objects.
- An object is an **instance** (real world entity) of a class.



**Employee Management System (abc.com)**

When a new employee joined

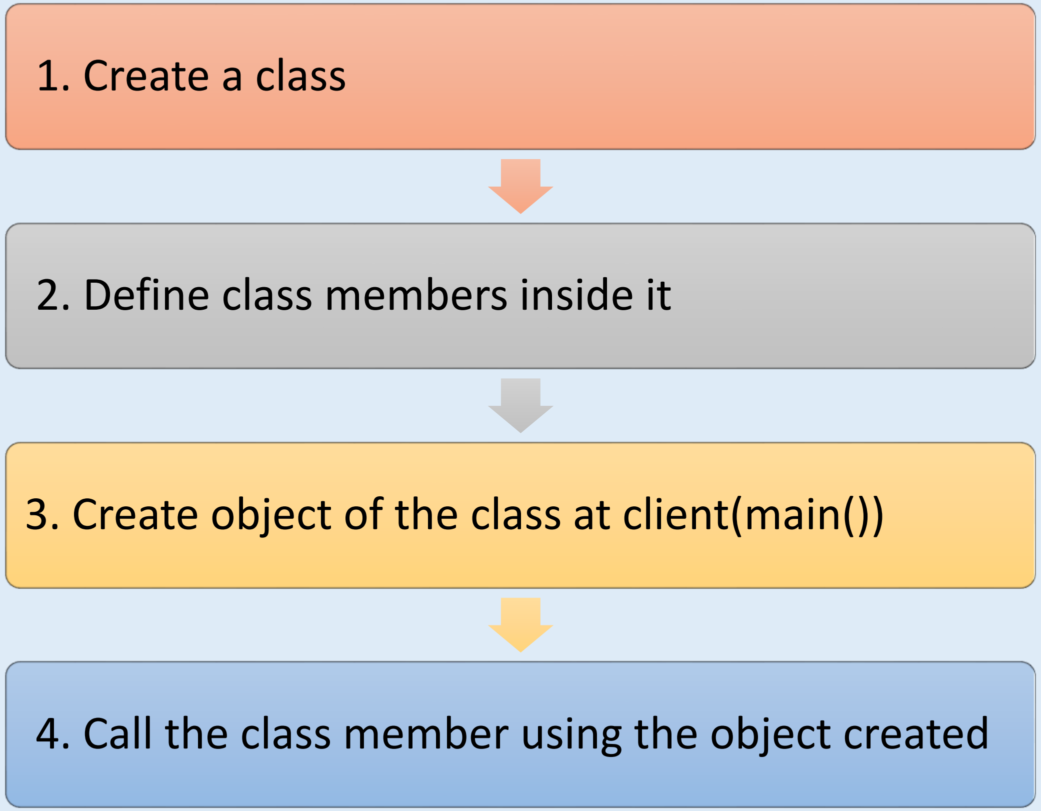
| Employee class (template) |
|---|
| - Fields: exp, name |
| - Functions: calculateSalary() |

| Object of Employee class(emp) |
|---|
| - Fields: emp.exp = 15, emp.name = Happy |
| - Functions: emp.calculateSalary() |

back to chapter index

# Q. How to implement classes and objects in Java?

- Steps to implement class & object:



1. Create a class

2. Define class members inside it

3. Create object of the class at client(main())

4. Call the class member using the object created

```java
public class Employee {  // Class

  private int exp; // 1. Field

  public Employee() {} // 2. Constructor

  public double calculateSalary() { // 3. Method
    int salary = exp * 50000;
    return salary;
  }

  public static void main(String[] args) {
    // Create an Employee(emp) object
    Employee emp = new Employee();

    emp.exp = 5;

    double salary = emp.calculateSalary();

    System.out.println(salary);
    //Output: 250000
  }
}
```

back to chapter index

# Q. What are the members of class?

❖ **Class members are:**

**1. Field**

A field is a variable of any type. It is basically holds the data.

**2. Constructor**

A constructor is a method in the class which gets executed when a class object is created.

**3. Method**

A method is a block of code that performs a specific task.

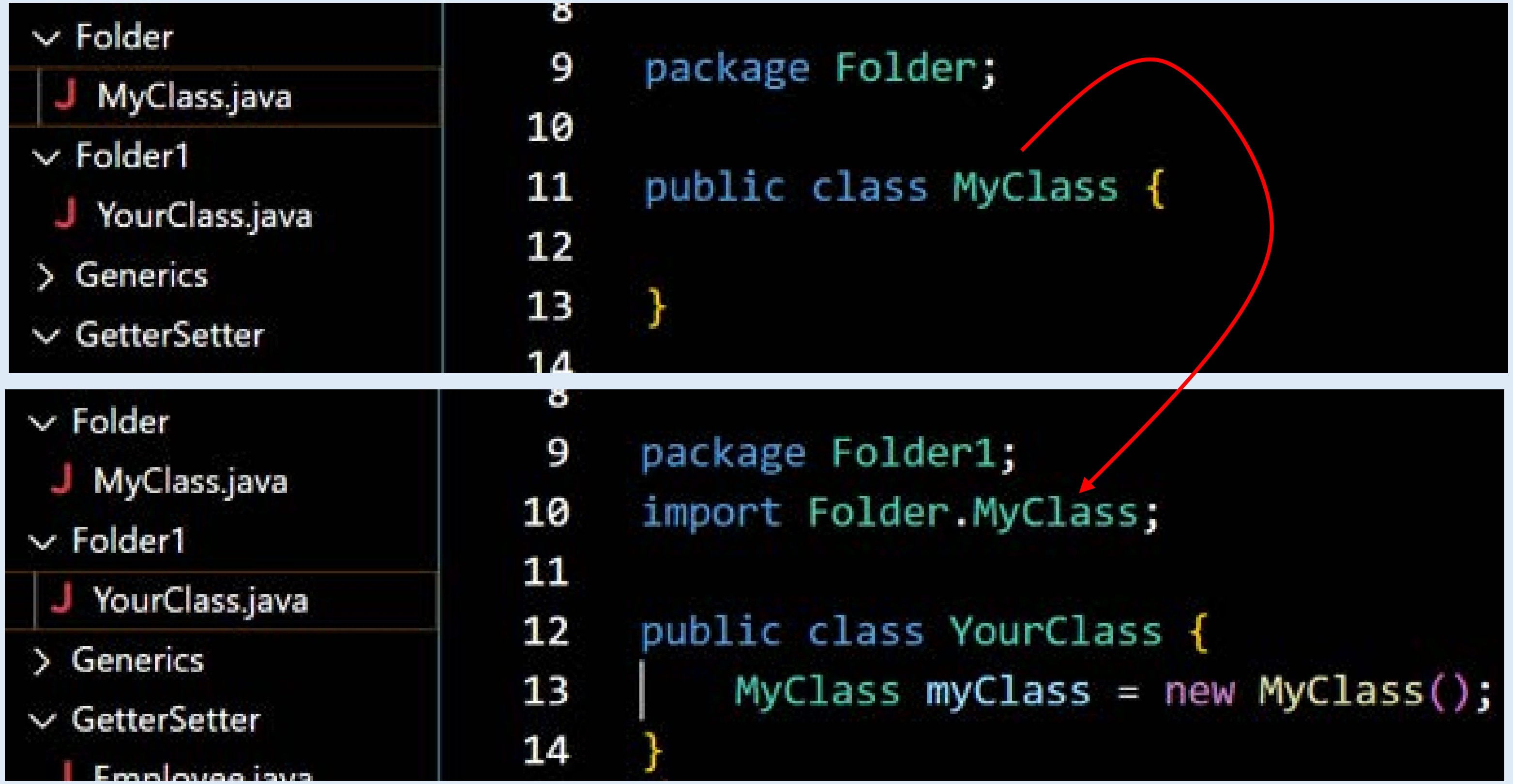
```java
public class Employee {  // Class

  private int exp; // 1. Field

  public Employee() {} // 2. Constructor

  public double calculateSalary() { // 3. Method
    int salary = exp * 50000;
    return salary;
  }

  public static void main(String[] args) {
    // Create an Employee(emp) object
    Employee emp = new Employee();

    emp.exp = 5;

    double salary = emp.calculateSalary();

    System.out.println(salary);
    //Output: 250000
  }
}
```

back to chapter index

# Q. What is the role and benefit of package in Java?

- Definition: Packages create a namespace that helps in **organizing classes** or interfaces.

- Benefit: This **avoids naming conflicts** by allowing the same class name to be used in different packages.



```java
package Folder;

public class MyClass {

}
```

```java
package Folder1;
import Folder.MyClass;

public class YourClass {
    MyClass myClass = new MyClass();
}
```

back to chapter index

# 9. OOPS - Access Specifiers, Getter-Setter & this keyword

Q. What are access specifiers? What are public and private specifiers? **V. IMP.**

Q. What is the role of default access specifier? Difference btw public, private and default?

Q. What is the role of this keyword in java? When to use it? **V. IMP.**

Q. Why to use same names for class fields and parameter name in Setter method?

Q. What are getter and setter methods?

Q. What are the advantages of getter and setter methods?

Q. What are the 4 principles/ pillars of OOPS? **V. IMP.**

Back to main index

# Q. What are access specifiers? What are public and private specifiers? V. IMP.

- Access specifiers (access modifiers) are keywords used to set the **access level** for classes, variables(fields), methods, and constructors.

| Access Specifier | Within Class | Within Package | Subclass (Same Package) | Subclass (Different Package) | Outside Package |
|---|---|---|---|---|---|
| public | Yes | Yes | Yes | Yes | Yes |
| protected | Yes | Yes | Yes | Yes | No |
| Default (no specifier) | Yes | Yes | Yes | No | No |
| private | Yes | No | No | No | No |

```
public class Student {
  public String name = "Happy";
  private String city = "Delhi";
}
```

```
public class ScienceStudent {

  public void GetStudent() {
    Student student = new Student();
    System.out.println(student.name);

    // Error: private not accessible
    System.out.println(student.city);
  }
}
```

back to chapter index

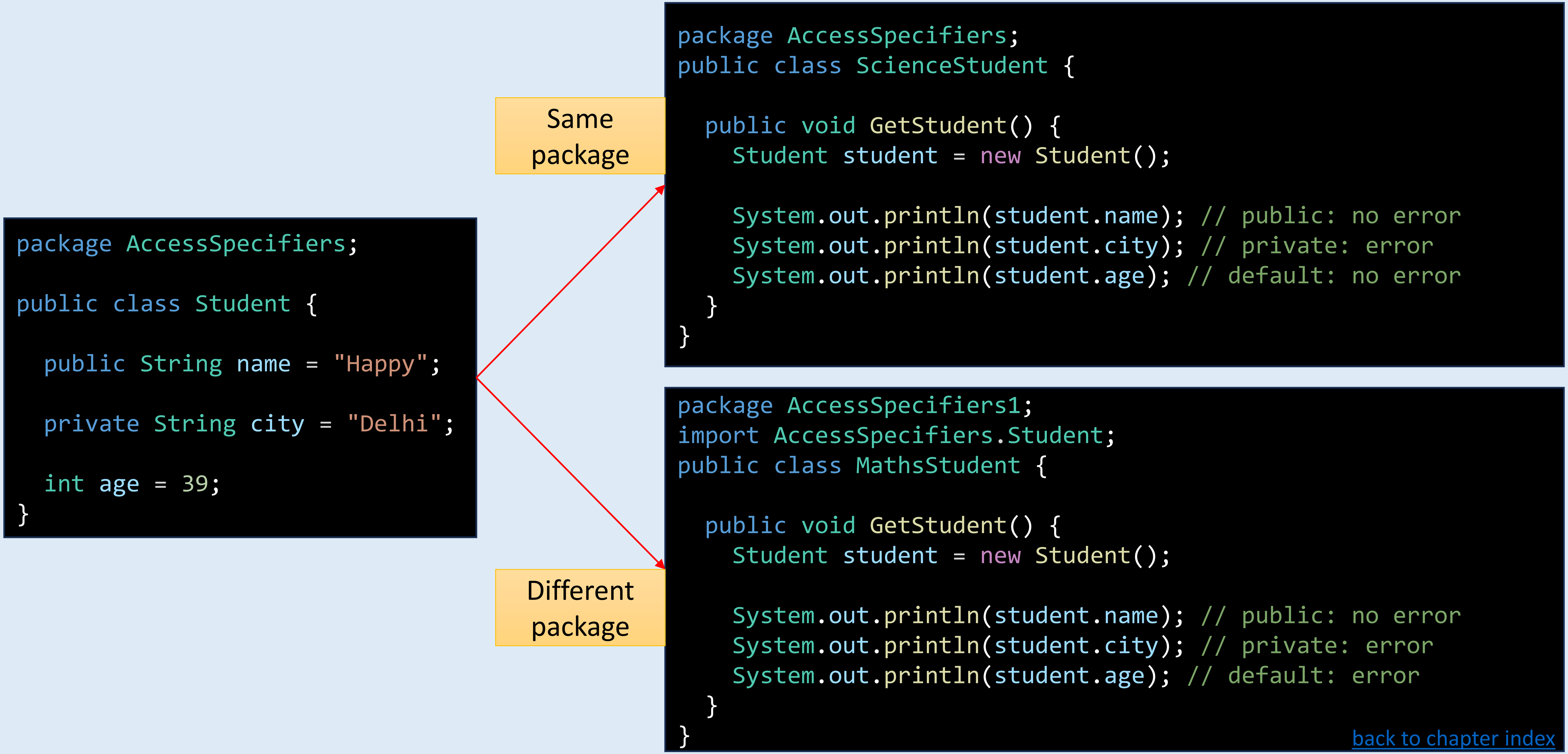
# Q. What is the role of default access specifier? Difference btw public, private and default?

- The default access specifier(package-private access) is used when no explicit access specifier is provided. It controls the accessibility of member within the same package.

| Access Specifier | Within Class | Within Package | Subclass (Same Package) | Subclass (Different Package) | Outside Package |
|---|---|---|---|---|---|
| public | Yes | Yes | Yes | Yes | Yes |
| protected | Yes | Yes | Yes | Yes | No |
| Default (no specifier) | Yes | Yes | Yes | No | No |
| private | Yes | No | No | No | No |

back to chapter index

# Q. What is the role of default access specifier? Difference btw public, private and default?



```java
package AccessSpecifiers;

public class Student {

  public String name = "Happy";

  private String city = "Delhi";

  int age = 39;
}
```

Same package

```java
package AccessSpecifiers;
public class ScienceStudent {

  public void GetStudent() {
    Student student = new Student();

    System.out.println(student.name); // public: no error
    System.out.println(student.city); // private: error
    System.out.println(student.age); // default: no error
  }
}
```

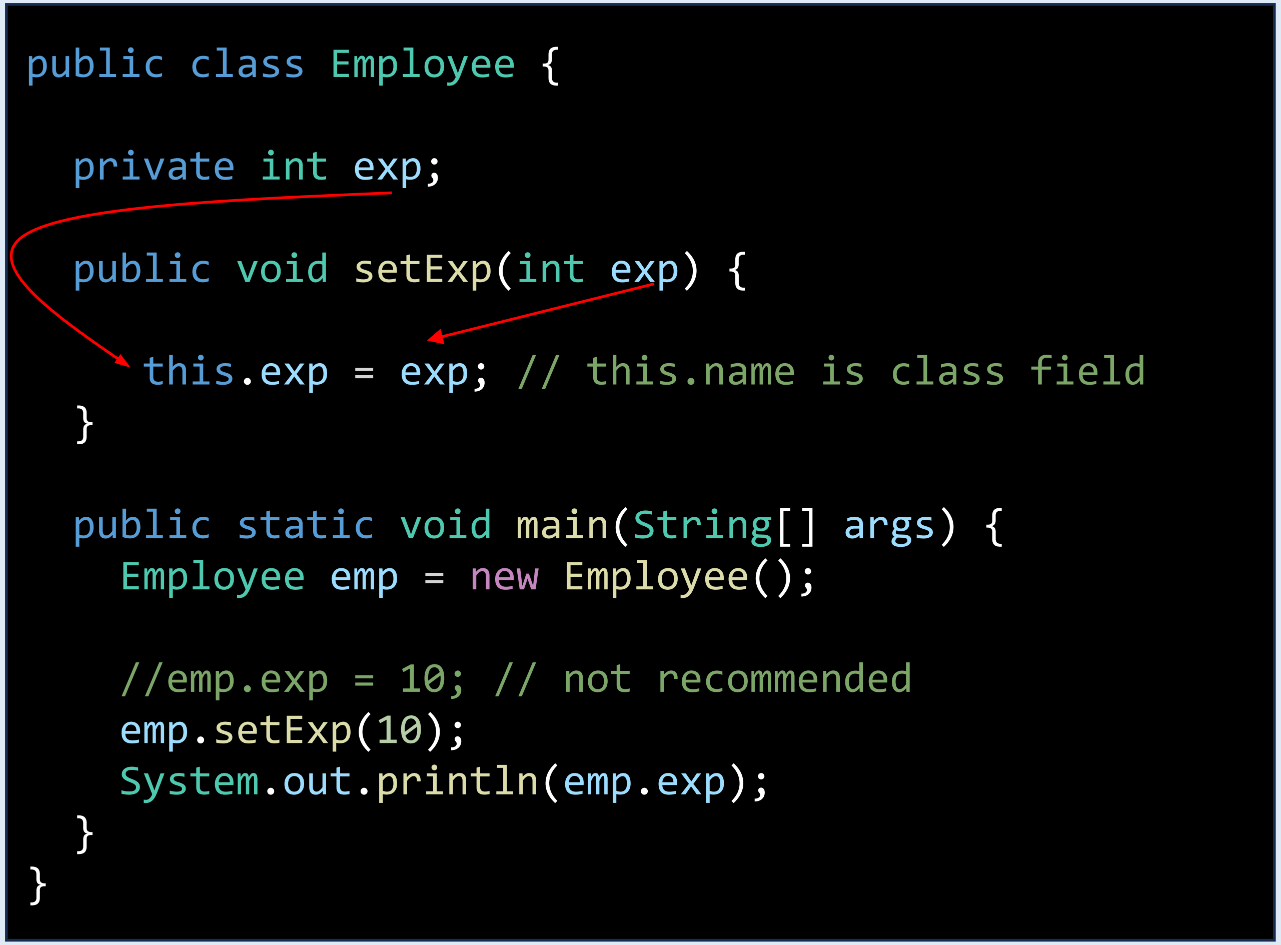
Different package

```java
package AccessSpecifiers1;
import AccessSpecifiers.Student;
public class MathsStudent {

  public void GetStudent() {
    Student student = new Student();

    System.out.println(student.name); // public: no error
    System.out.println(student.city); // private: error
    System.out.println(student.age); // default: error
  }
}
```

back to chapter index

# Q. What is the role of this keyword in java? When to use it? V. IMP.

- this keyword refers to the current instance of the class.
- Use of this keywords: Differentiate between instance variables and parameters with the same name in setter methods.

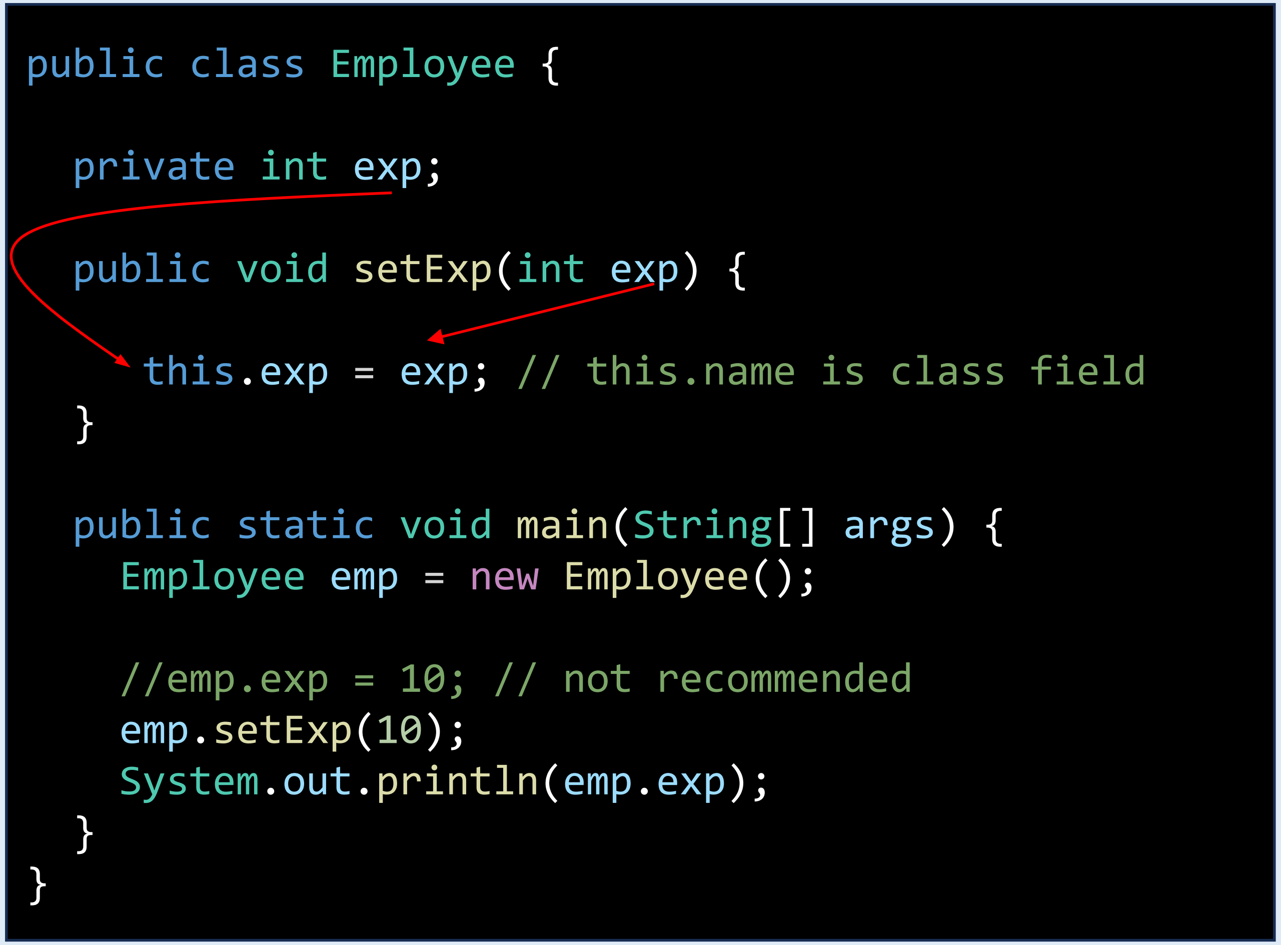


```java
public class Employee {

  private int exp;

  public void setExp(int exp) {

     this.exp = exp; // this.name is class field
  }

  public static void main(String[] args) {
    Employee emp = new Employee();

    //emp.exp = 10; // not recommended
    emp.setExp(10);
    System.out.println(emp.exp);
  }
}
```

back to chapter index

# Q. Why to use same names for class fields and parameter name in Setter method?

- Using the same names for class fields and parameter names in setter methods can lead to cleaner, more readable code



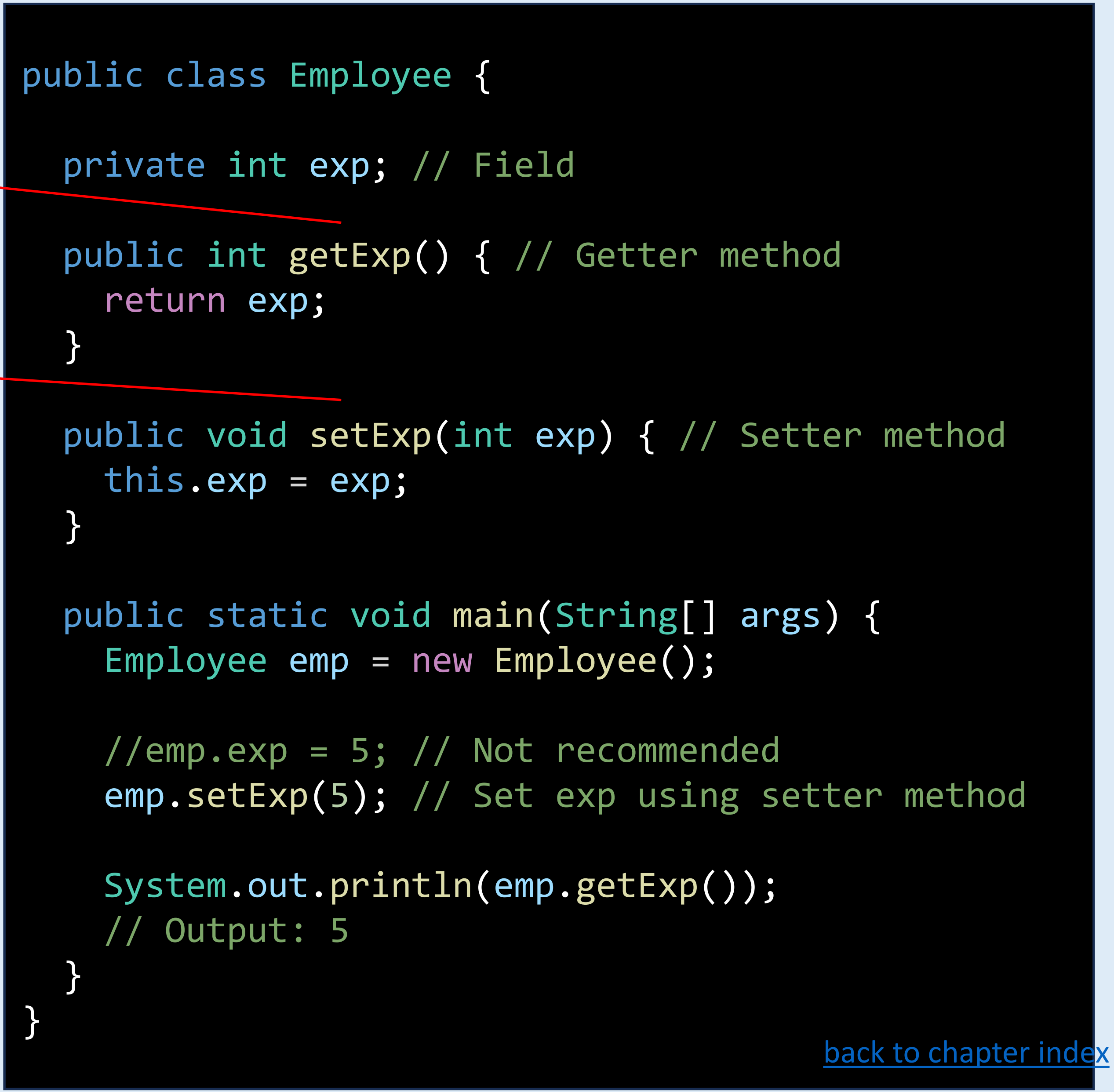
```
public class Employee {

  private int exp;

  public void setExp(int exp) {

     this.exp = exp; // this.name is class field
  }

  public static void main(String[] args) {
    Employee emp = new Employee();

    //emp.exp = 10; // not recommended
    emp.setExp(10);
    System.out.println(emp.exp);
  }
}
```

back to chapter index

# Q. What are getter and setter methods?

- Getter methods are used to **retrieve the values** of private fields of a class.

- Setter methods are used to **modify or set the value** of a private field of a class.



```
public class Employee {

  private int exp; // Field

  public int getExp() { // Getter method
    return exp;
  }

  public void setExp(int exp) { // Setter method
    this.exp = exp;
  }

  public static void main(String[] args) {
    Employee emp = new Employee();

    //emp.exp = 5; // Not recommended
    emp.setExp(5); // Set exp using setter method

    System.out.println(emp.getExp());
    // Output: 5
  }
}
```

back to chapter index

# Q. What are the advantages of getter and setter methods?

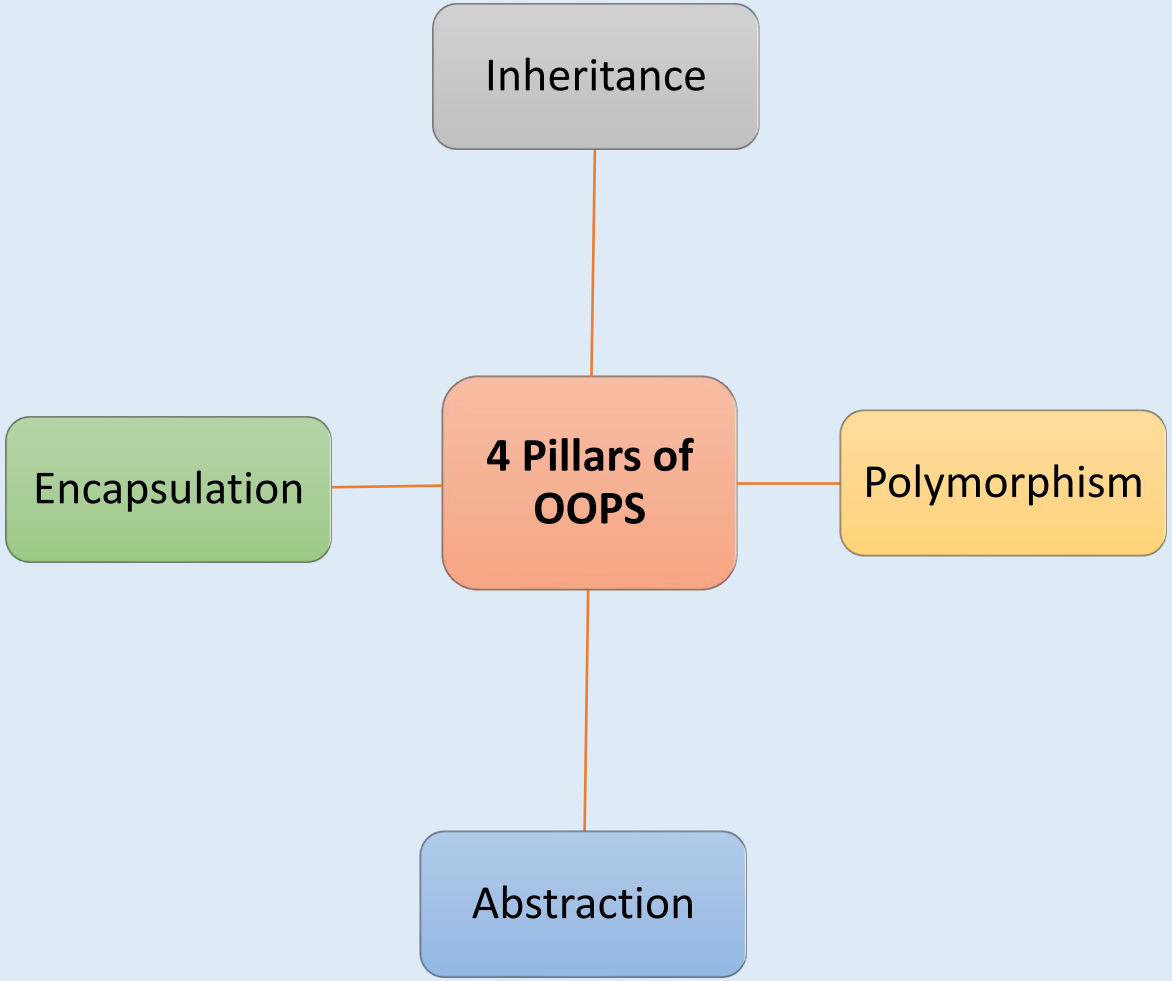
❖ **Advantages of getter and setter methods:**

1. **Data Validation:** Setter methods can include validation logic to ensure that the assigned values are valid.
2. **Data Access:** Getter methods can control how a field is accessed.
3. **Encapsulation/ Abstraction:** Getter and setter methods **hide the internal implementation** details of a class and expose only necessary information.

```java
public class Employee {
  private int exp; // Field

  public int getExp() { // Getter method
    return exp * 2;
  }

  public void setExp(int exp) { // Setter method
    if (exp < 0) { // Validate experience
      System.out.println("Incorrect Experience");
      return;
    }
    this.exp = exp;
  }

  public static void main(String[] args) {
    Employee emp = new Employee();

    //emp.exp = 5; // Not recommended
    emp.setExp(5); // Set exp using setter method

    System.out.println(emp.getExp()); // Output: 10
  }
}
```

back to chapter index

# Q. What are the 4 principles/ pillars of OOPS? V. IMP.



back to chapter index

# 10. OOPS - Inheritance

Q. What is inheritance and when to use inheritance in real applications? **V. IMP.**

Q. How to implement inheritance in Java?

Q. What are the different types of Inheritance? When to use what? **V. IMP.**

Q. What is multiple inheritance? Does Java support it?

Q. Why Java does not support multiple inheritance of classes? What is diamond problem?

Q. What is the alternative of multiple inheritance in Java? **V. IMP.**

Q. How to prevent a class from being inherited?

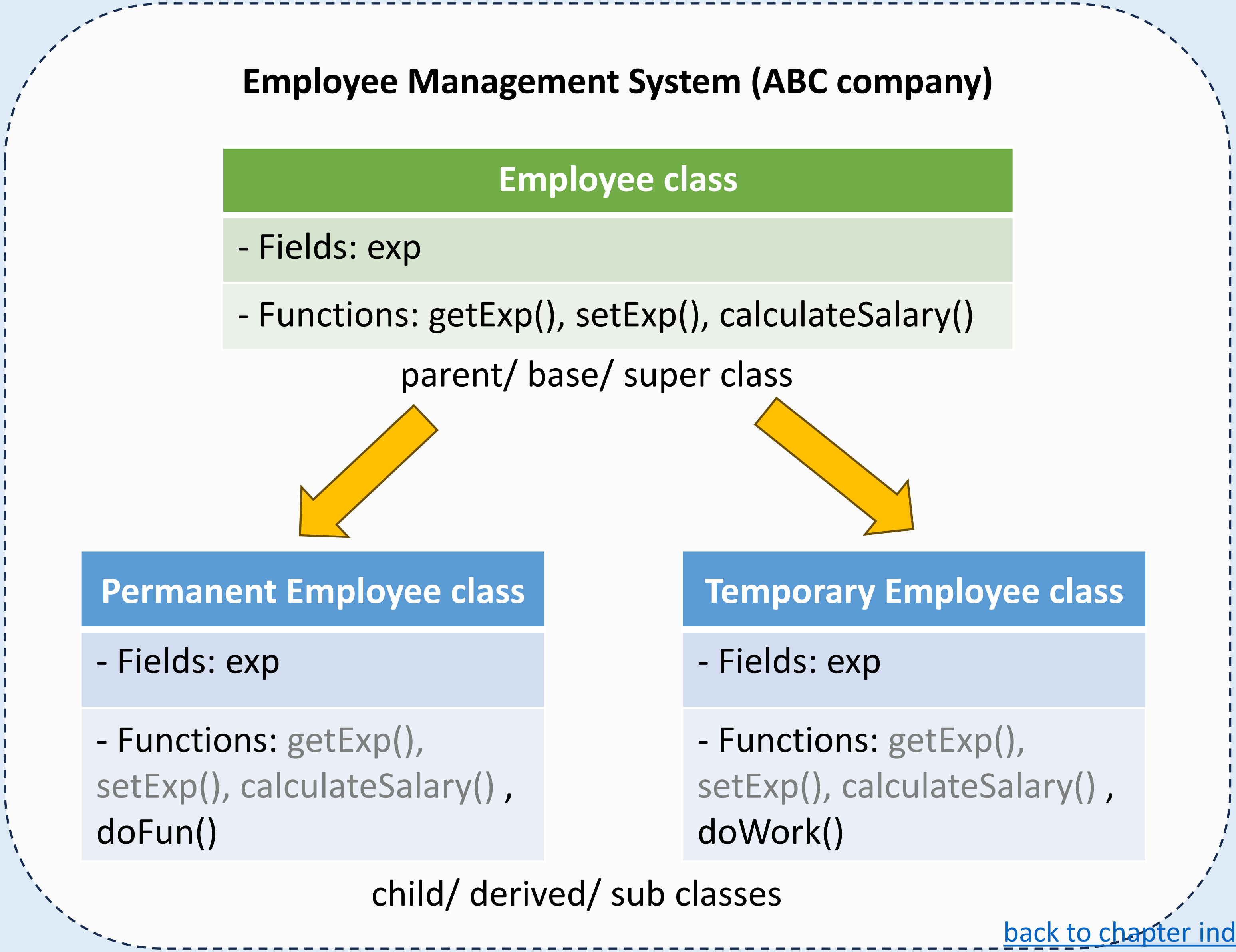
Back to main index

# Q. What is inheritance and when to use inheritance in real applications? V. IMP.

- **Definition:** Inheritance is creating a **parent-child** relationship between two classes, where child class will automatically get the properties and methods of the parent.

- **When to use of inheritance?**

- **Code Reusability:** Multiple Subclasses can reuse fields and methods from a superclass.



**Employee Management System (ABC company)**

**Employee class**

- Fields: exp

- Functions: getExp(), setExp(), calculateSalary()

parent/ base/ super class

**Permanent Employee class**

- Fields: exp

- Functions: getExp(), setExp(), calculateSalary() , doFun()

**Temporary Employee class**

- Fields: exp

- Functions: getExp(), setExp(), calculateSalary() , doWork()

child/ derived/ sub classes

back to chapter index

# Q. How to implement inheritance in Java? V. IMP.

- Use **extends keyword** to setup parent-child relationship(inheritance) between two classes.

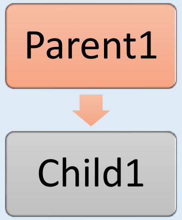
```java
// Inheritance example
// Employee class (parent/ base/ super)
public class Employee {

  private int exp;

  public int getExp() {
    return exp;
  }

  public void setExp(int exp) {
    this.exp = exp;
  }

  public double calculateSalary() {
    int salary = exp * 50000;
    return salary;
  }
}
```

```java
// Permanent Employee class (child/ derived/ sub)
public class PermEmployee extends Employee {

  public void doFun() {
    System.out.println("having fun");
  }

  public static void main(String[] args) {
    // Create an Employee object
    PermEmployee permEmp = new PermEmployee();

    permEmp.doFun(); // Call sub class own method

    // superclass methods automatically available
    // for subclass object
    permEmp.setExp(5);

    double salary = permEmp.calculateSalary();
    System.out.println(salary);
    //Output: having fun    250000
  }
}
```

back to chapter index

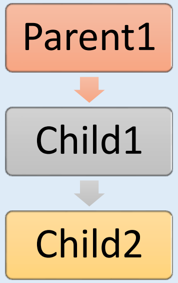
# Q. What are the different types of Inheritance? When to use what? V. IMP.

## 1. Single Inheritance



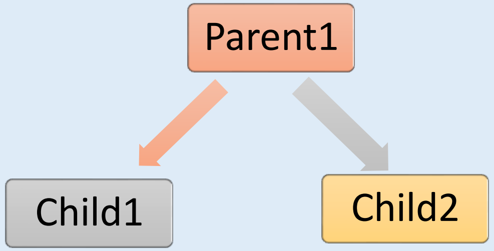
```java
// Single Inheritance
class Parent1 {
  public void display() {
    System.out.println("Parent1");
  }
}

class Child1 extends Parent1 {
  public void show() {
    System.out.println("Child1");
  }
}
```

## 2. Multilevel Inheritance



```java
// Multilevel Inheritance
class Parent1 {
  public void display() {
    System.out.println("Parent1");
  }
}

class Child1 extends Parent1 {
  public void show() {
    System.out.println("Child1");
  }
}

class Child2 extends Child1 {
  public void print() {
    System.out.println("Child2");
  }
}
```

## 3. Hierarchical Inheritance

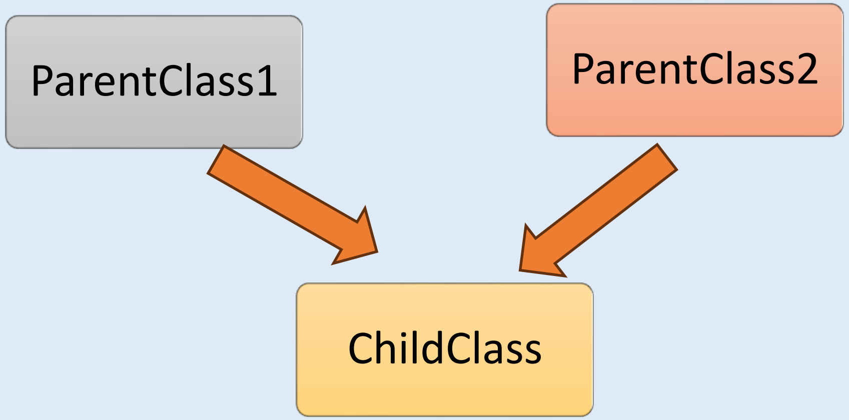


```java
// Hierarchical Inheritance
class Parent1 {
  public void display() {
    System.out.println("Parent1");
  }
}

class Child1 extends Parent1 {
  public void show() {
    System.out.println("Child1");
  }
}

class Child2 extends Parent1 {
  public void print() {
    System.out.println("Child2");
  }
}
```

back to chapter index

# Q. What is multiple inheritance? Does Java support it?

- In multiple inheritance a class can inherit attributes and methods from more than one parent class, but java does not support it.



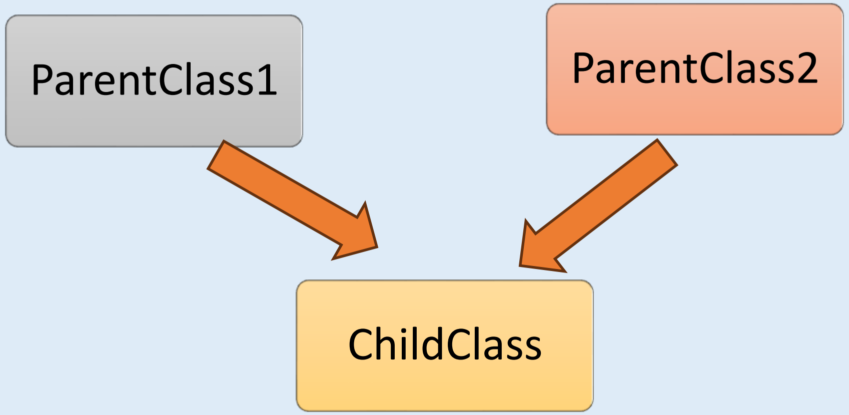
Multiple Inheritance
(Not allowed in Java)

```java
class ParentClass1 {
    void method1() {
        System.out.println("method1");
    }
}

class ParentClass2 {
    void method2() {
        System.out.println("method2");
    }
}

// Attempt at multiple inheritance
// (which is not allowed in Java)
class ChildClass extends ParentClass1, ParentClass2
{
    void childMethod() {
        System.out.println("Child class method");
    }
}
```

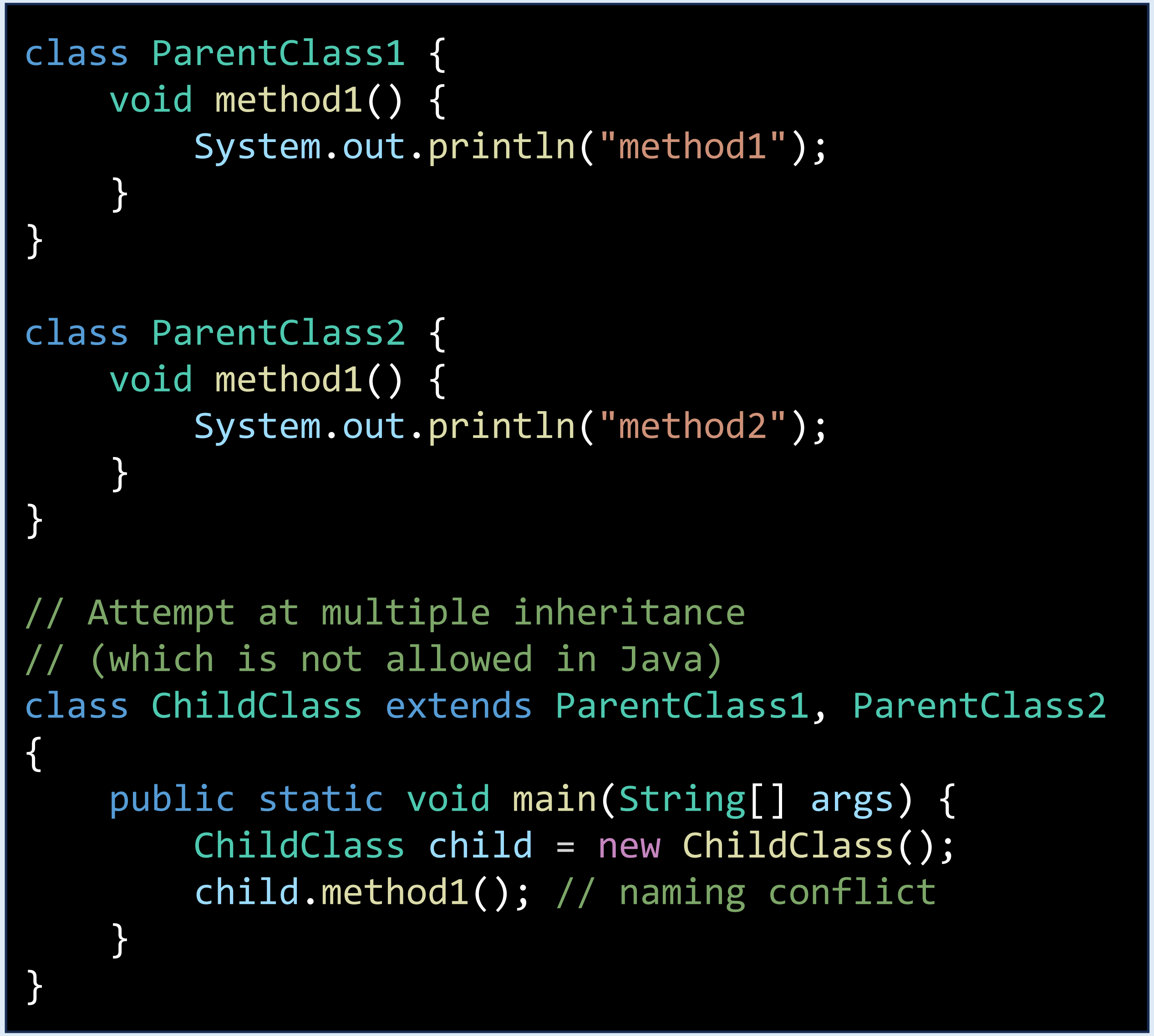
back to chapter index

# Q. Why Java does not support multiple inheritance of classes? What is diamond problem?

- Multiple inheritance can lead to the "diamond problem“. In diamond problem, confusion arises if a child class inherits from two parent classes that have a same name method.



Multiple Inheritance
(Not allowed in Java)

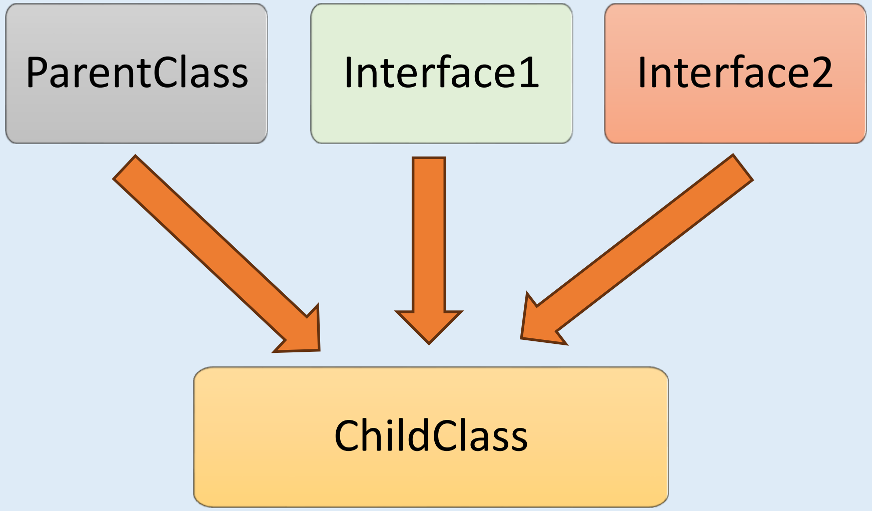


```java
class ParentClass1 {
    void method1() {
        System.out.println("method1");
    }
}

class ParentClass2 {
    void method1() {
        System.out.println("method2");
    }
}

// Attempt at multiple inheritance
// (which is not allowed in Java)
class ChildClass extends ParentClass1, ParentClass2
{
    public static void main(String[] args) {
        ChildClass child = new ChildClass();
        child.method1(); // naming conflict
    }
}
```

back to chapter index

# Q. What is the alternative of multiple inheritance in Java? V. IMP.

- Java support multiple inheritance of interfaces. So multiple interfaces and maximum one parent class can be inherited.



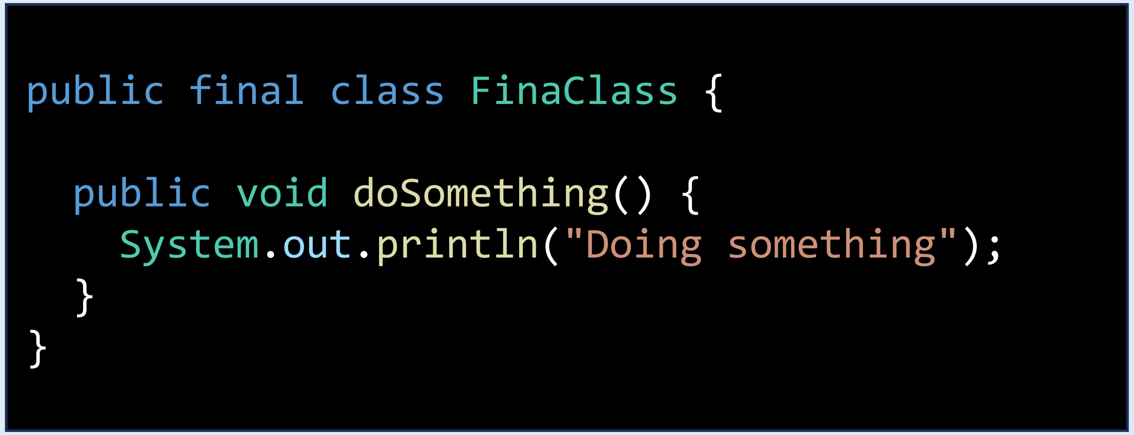
Allowed in Java

```java
class ParentClass { // Parent class (superclass)
  public void parentMethod() {
    System.out.println("ParentClass method");
  }
}

interface Interface1 { // Interface 1
  void method1();
}

interface Interface2 { // Interface 2
  void method2();
}

// Child class (subclass)
class ChildClass extends ParentClass
                implements Interface1, Interface2 {
  public void method1() {
    System.out.println("Implementation of method1");
  }
  public void method2() {
    System.out.println("Implementation of method2");
  }
}
```

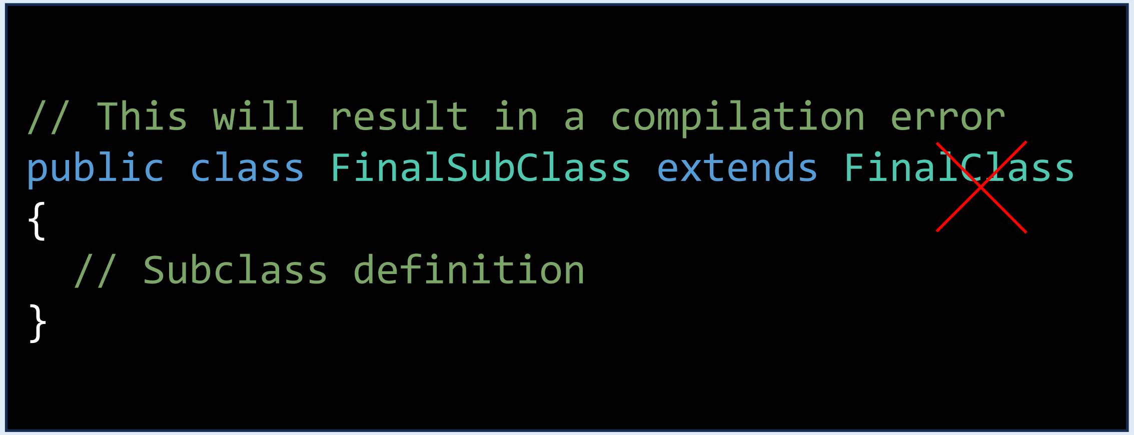
back to chapter index

# Q. How to prevent a class from being inherited?

- In Java, you can prevent a class from being inherited (subclassed) by using the **final keyword**.



```
public final class FinaClass {

  public void doSomething() {
    System.out.println("Doing something");
  }
}
```



```
// This will result in a compilation error
public class FinalSubClass extends FinalClass
{
  // Subclass definition
}
```

back to chapter index

# 11. OOPS - Polymorphism

Q. What is polymorphism?

Q. What are the types of polymorphism? **V. IMP.**

Q. What is Method Overloading? How to implement it and when to use it?

Q. Why do we call method overloading as a type of compile-time or early binding?

Q. In how many ways can a method be overloaded? **V. IMP.**

Q. If two same methods have different return type, then are methods are overloaded?

Q. What is Method Overriding? **V. IMP.**

Q. Why to use method overriding? Why don’t we have different name methods?

Q. How to implement method overriding?

Q. Why we call method overriding as a run-time or late binding?

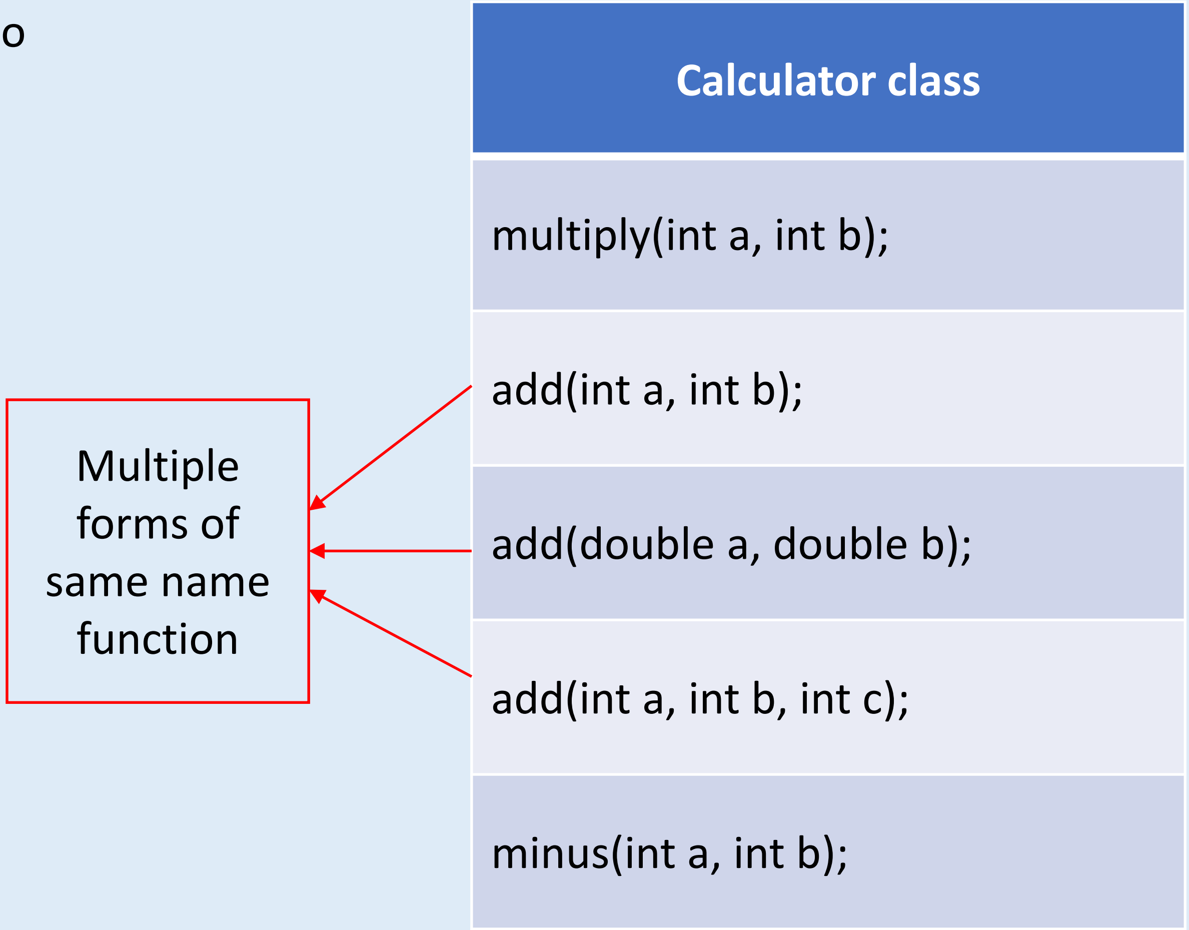
Q. What are the 5 differences between Overloading and Overriding? **V. IMP.**

Q. What are Annotations in Java?

Back to main index

# Q. What is polymorphism?

- Polymorphism is the ability of a function to take on **multiple forms**.

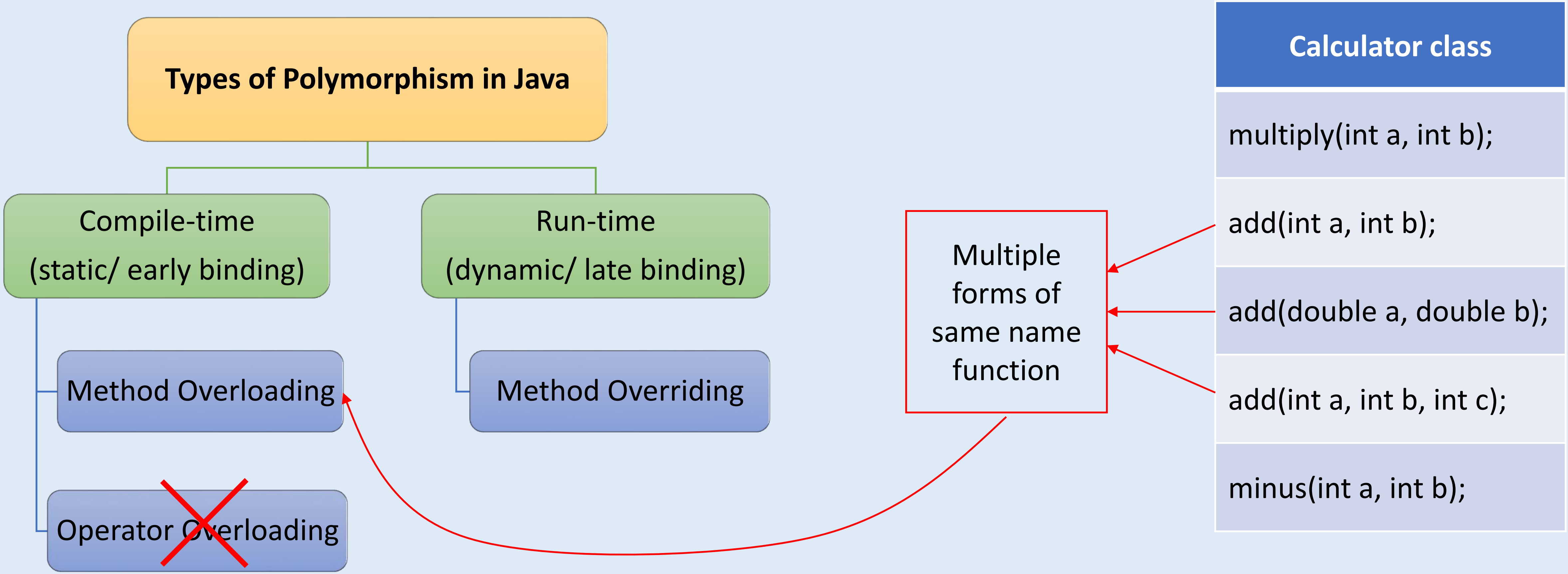


| Calculator class |
|---|
| multiply(int a, int b); |
| add(int a, int b); |
| add(double a, double b); |
| add(int a, int b, int c); |
| minus(int a, int b); |

Multiple forms of same name function

back to chapter index

# Q. What are the types of polymorphism? V. IMP.



Types of Polymorphism in Java

Compile-time (static/ early binding)

- Method Overloading
- ~~Operator Overloading~~

Run-time (dynamic/ late binding)

- Method Overriding

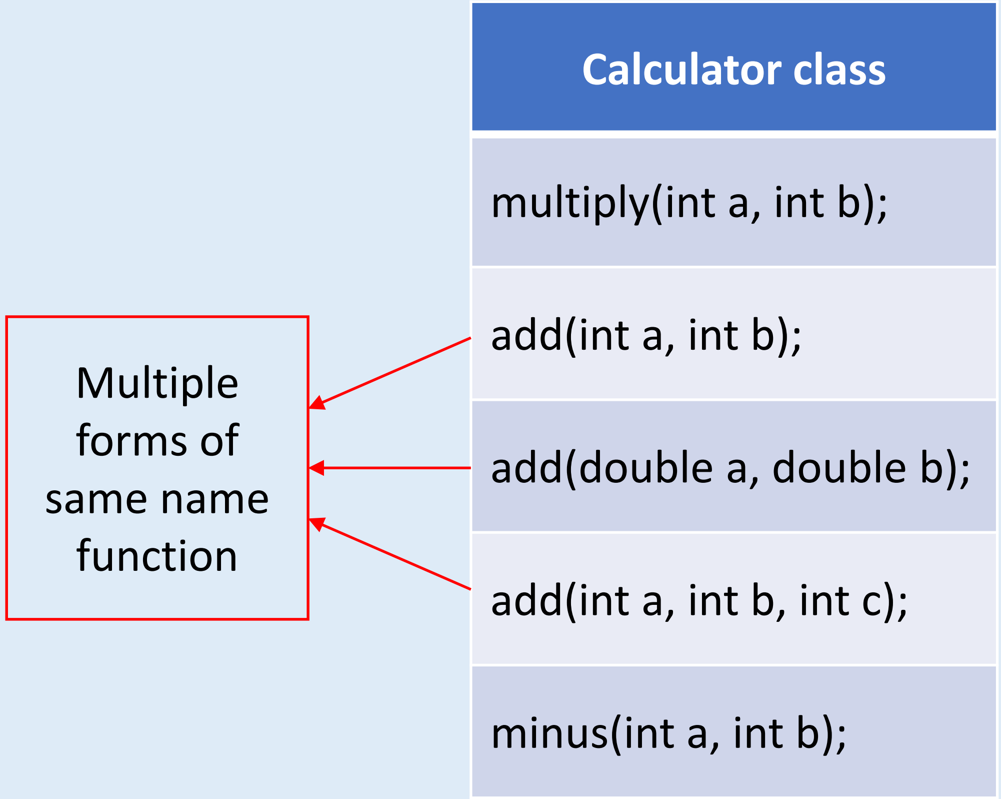
Multiple forms of same name function

| Calculator class |
|---|
| multiply(int a, int b); |
| add(int a, int b); |
| add(double a, double b); |
| add(int a, int b, int c); |
| minus(int a, int b); |

back to chapter index

# Q. What is Method Overloading? How to implement it and when to use it?

- Method Overloading occurs when there are multiple methods with the same name within the same class, but with **different parameter lists**.



Calculator class

multiply(int a, int b);

add(int a, int b);

add(double a, double b);

add(int a, int b, int c);

minus(int a, int b);

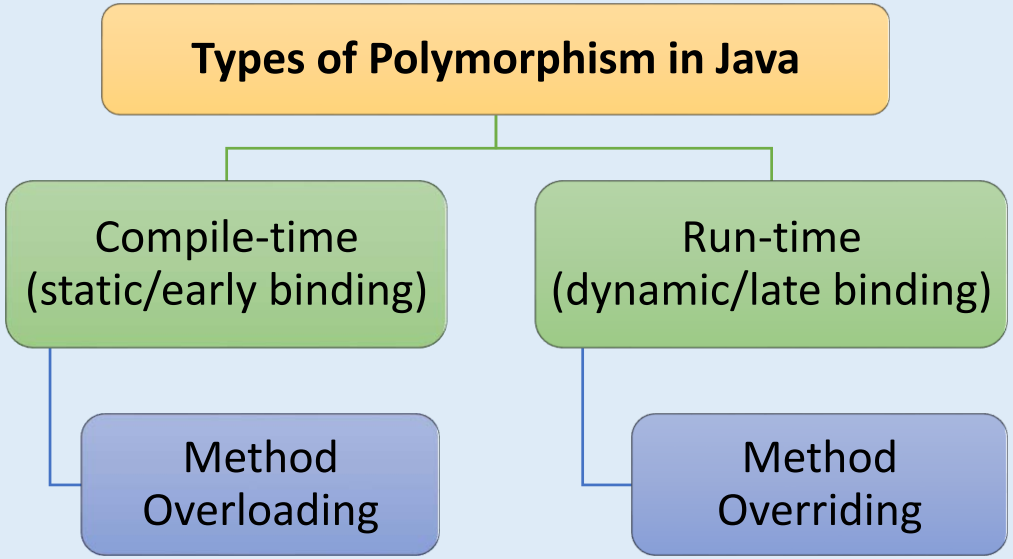
Multiple forms of same name function

```java
public class Calculator {

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

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

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

  public static void main(String[] args) {
    Calculator calc = new Calculator();

    System.out.println(calc.add(5, 10)); // 15
    System.out.println(calc.add(3.5, 2.5)); // 6.0
    System.out.println(calc.add(2, 4, 6)); // 12
  }
}
```

back to chapter index

# Q. Why do we call method overloading as a type of compile-time or early binding?

- Method overloading in Java is referred to as compile-time polymorphism (or early binding) because the decision of which method to call **happens at compile time** based on the method signature(method parameters), before the program is executed.



Types of Polymorphism in Java

Compile-time (static/early binding)

Run-time (dynamic/late binding)

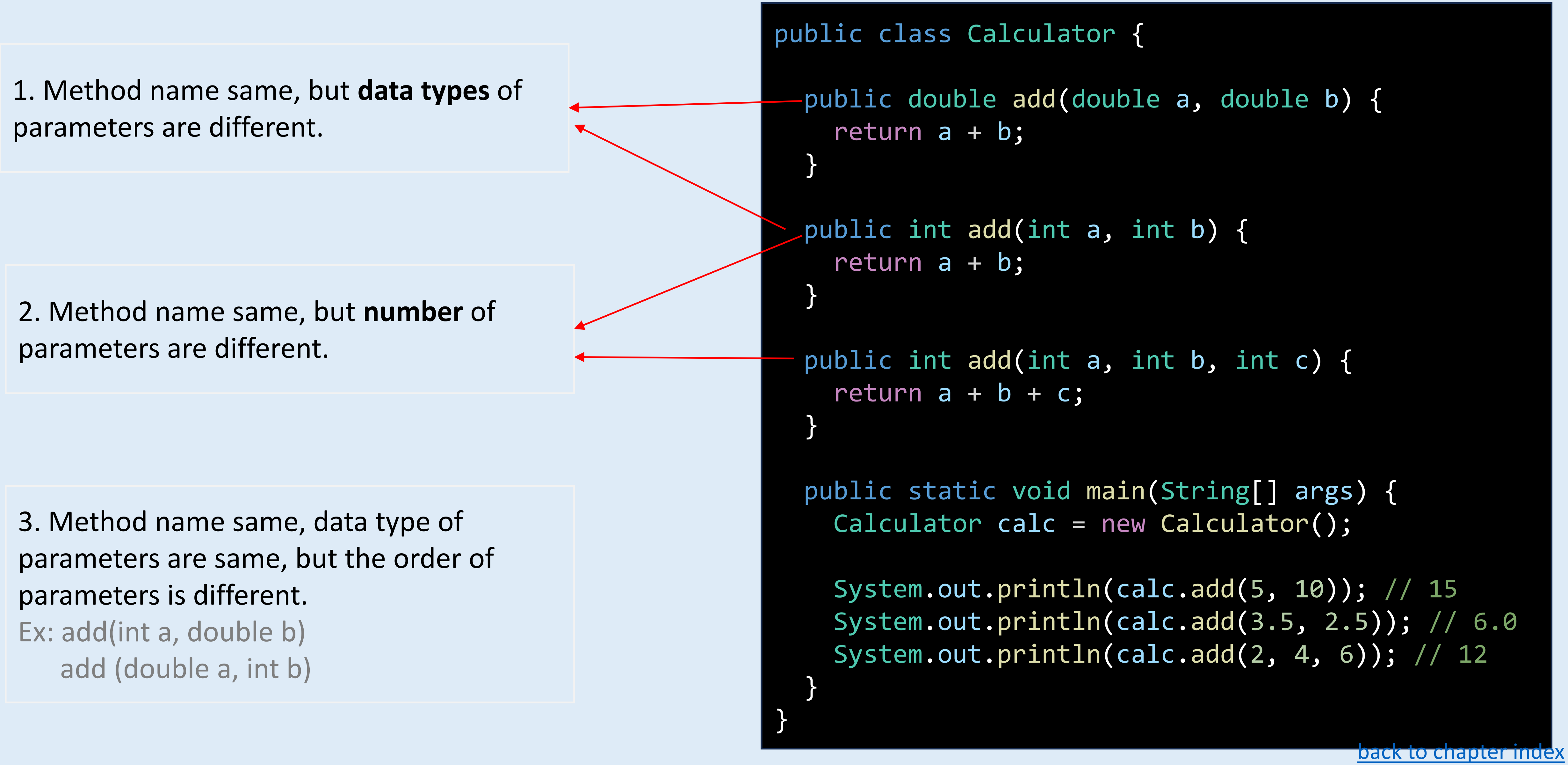
Method Overloading

Method Overriding

```java
public class Calculator {

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

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

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

  public static void main(String[] args) {
    Calculator calc = new Calculator();

    System.out.println(calc.add(5, 10)); // 15
    System.out.println(calc.add(3.5, 2.5)); // 6.0
    System.out.println(calc.add(2, 4, 6)); // 12
  }
}
```

back to chapter index

# Q. In how many ways can a method be overloaded? V. IMP.



1. Method name same, but **data types** of parameters are different.

2. Method name same, but **number** of parameters are different.

3. Method name same, data type of parameters are same, but the order of parameters is different.
Ex: add(int a, double b)
add (double a, int b)

```
public class Calculator {

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

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

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

  public static void main(String[] args) {
    Calculator calc = new Calculator();

    System.out.println(calc.add(5, 10)); // 15
    System.out.println(calc.add(3.5, 2.5)); // 6.0
    System.out.println(calc.add(2, 4, 6)); // 12
  }
}
```

back to chapter index

# Q. If two same methods have different return type, then are methods are overloaded?

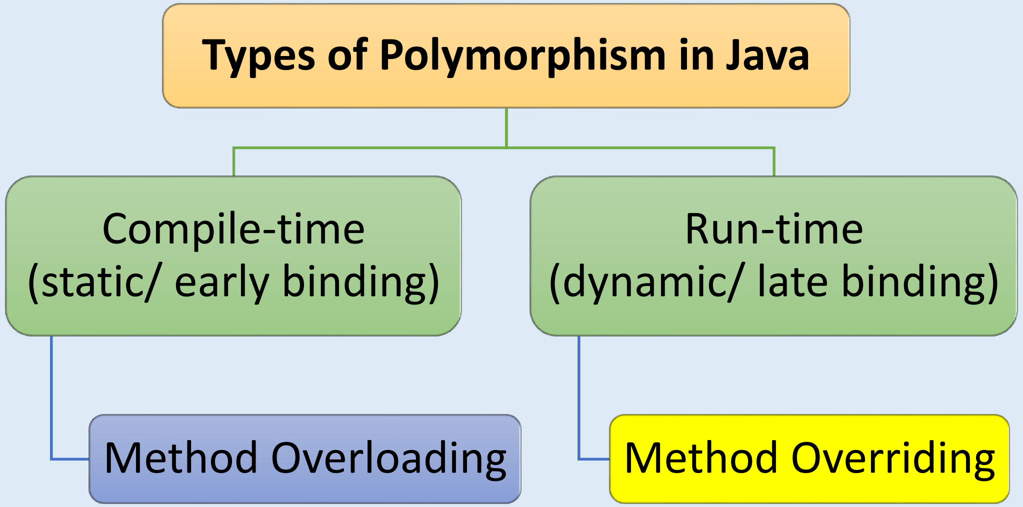
- **No**, methods are not overloaded, this will show compile time error.

```
public class DifferentReturnType {

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

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

    // Compile time error: Duplicate methods

}
```

back to chapter index

# Q. What is Method Overriding? V. IMP.

- Method overriding in Java allows a subclass to provide a specific implementation of a method that is already defined in its superclass.



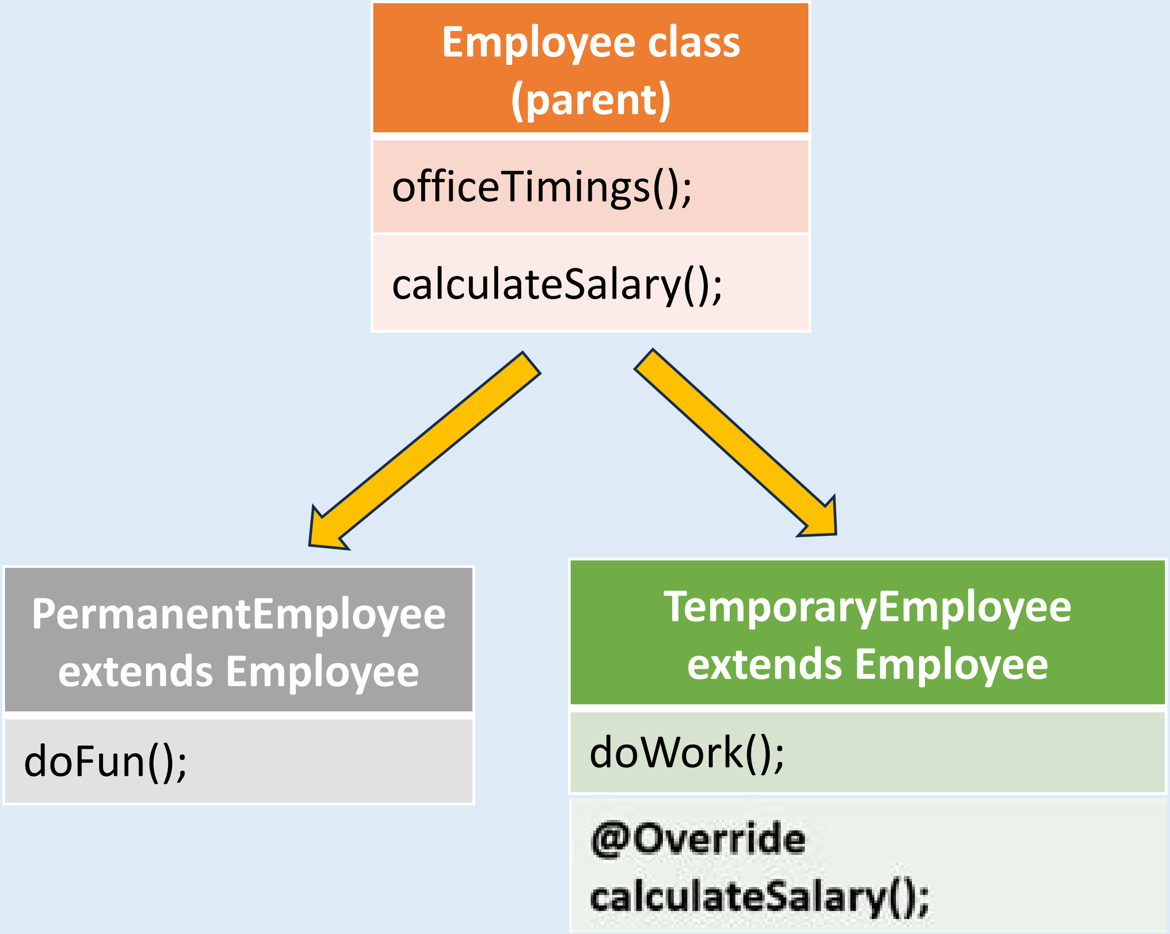
Types of Polymorphism in Java

Compile-time (static/ early binding)

Method Overloading

Run-time (dynamic/ late binding)

Method Overriding



Employee class (parent)

officeTimings();

calculateSalary();

PermanentEmployee extends Employee

doFun();

TemporaryEmployee extends Employee

doWork();

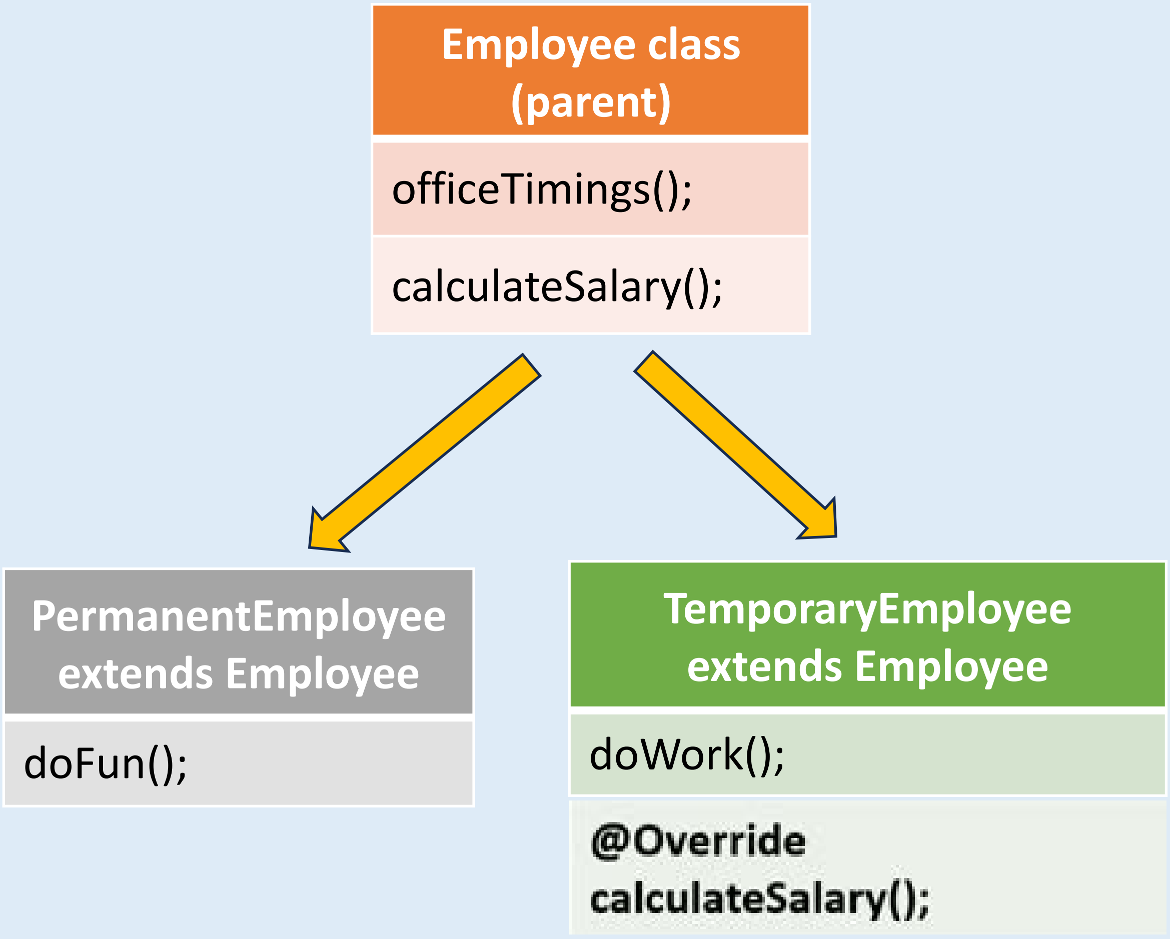
@Override calculateSalary();

Method Overriding

back to chapter index

# Q. Why to use method overriding? Why don’t we have different name methods?

- Method-overriding concept for making our application more structured, readable, and organized



Employee class (parent)

officeTimings();

calculateSalary();

PermanentEmployee extends Employee

doFun();

TemporaryEmployee extends Employee

doWork();

**@Override calculateSalary();**

Method Overriding

back to chapter index

# Q. How to implement method overriding?

- Use the **@Override annotation** before the subclass method declaration to override the same name and same signature method of the parent class.

```
public class Employee { // Parent Class

    public void calculateSalary() {
        System.out.println(100000);
    }

    public void officeTimings() {
        System.out.println("9am-6pm");
    }
}
```

```
public class TemporaryEmp extends
Employee {

  @Override //annotation used to
override parent class method
  public void calculateSalary() {
    System.out.println(75000);
  }
}
```

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

    TemporaryEmp tEmp = new TemporaryEmp();

    tEmp.calculateSalary(); // call overrided method

    tEmp.officeTimings(); // call parent class method
  }
}
// Output: 75000   9am-6pm
```

[back to chapter index]

# Q. Why we call method overriding as a run-time or late binding?

- Method overriding in Java is called run-time polymorphism (or late binding) because the decision of which method to execute is made dynamically at runtime based on the actual object.

```java
public class Employee { // Parent Class

    public void calculateSalary() {
        System.out.println(100000);
    }

    public void officeTimings() {
        System.out.println("9am-6pm");
    }
}
```

```java
public class TemporaryEmp extends
Employee {

  @Override //annotation used to
override parent class method
  public void calculateSalary() {
    System.out.println(75000);
  }
}
```

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

    TemporaryEmp tEmp = new TemporaryEmp();

    tEmp.calculateSalary(); // call overrided method

    tEmp.officeTimings(); // call parent class method
  }
}
// Output: 75000   9am-6pm
```

back to chapter index

# Q. What are the 5 differences between Overloading and Overriding? V. IMP.

## Method Overloading

```java
public class Calculator {

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

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

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

## Method Overriding

```java
// Parent Class
public class Employee {

    public void calculateSalary() {
        System.out.println(100000);
    }
}
```

```java
// Child Class
public class TemporaryEmp extends Employee
{
  @Override //annotation
  public void calculateSalary() {
    System.out.println(75000);
  }
}
```

back to chapter index

# Q. What are the 5 differences between Overloading and Overriding? V. IMP.

| Overloading | Overriding |
|---|---|
| 1. Multiple methods with same name are in the **same class**. | Multiple methods with same name are in the **different classes**. |
| 2. Methods have the same name but different parameter lists. | Method name and signature (parameter list) must be same. |
| 3. Compile-time polymorphism. | Runtime polymorphism. |
| 4. No annotation or special keyword is required. | @Override annotation is used over method of child class to specify that it's overriding a superclass method. |
| 5. Constructors can also be overloaded. | Constructors cannot be overridden. |

back to chapter index

# Q. What are Annotations in Java?

- Annotations in Java are a form of **metadata** that provide information about a program/ function to the compiler.
- Annotations are declared using the **@** symbol.

```
public class Employee {

    public void calculateSalary() {
        System.out.println(100000);
    }

}
```

```
public class TemporaryEmp extends Employee {

  @Override
  public void calculateSalary() {
    System.out.println(75000);
  }
}
```

- **Two types annotations:**

**1. Built-in Annotations:** Ex: @Override, @Deprecated, @SuppressWarnings, etc.

**2. Custom Annotations** Developers can define custom annotations to add metadata to their code.

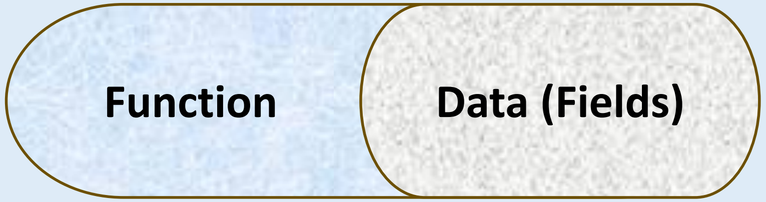
back to chapter index

# 12. OOPS - Encapsulation & Abstraction

Back to main index

# Q. What is Encapsulation? V. IMP.

- Encapsulation is the bundling of data(attributes or fields) and functions into a single unit(class).



Encapsulation

back to chapter index

# Q. How to achieve Encapsulation in Java? V. IMP.

- Encapsulation can be achieved by using **access modifier(private)** and **getters and setters methods**.

```java
public class Employee {

  private int exp; // Data(Field)

  public int getExp() { // Getter method
    return exp;
  }

  public void setExp(int exp) { // Setter method
    this.exp = exp;
  }

  public static void main(String[] args) {

    Employee emp = new Employee();

    //emp.exp = 5; // Not recommended
    emp.setExp(5); // Set exp using setter method

    System.out.println(emp.getExp()); // Output: 10
  }
}
```

back to chapter index

# Q. What are the advantages of Encapsulation in Java?

**1. Data hiding**

Encapsulation allows an object to hide its internal state(data) because all interaction will be performed through its methods.

**2. Modularity**

By encapsulating data and functions within an object, you can create self-contained modules.

**3. Increased Security**

By restricting access to data directly, encapsulation helps in protecting sensitive data.

**4. Reusability**

Encapsulation allows for components to be reused across different parts of a program or even in different programs.

**5. Controlled Access**

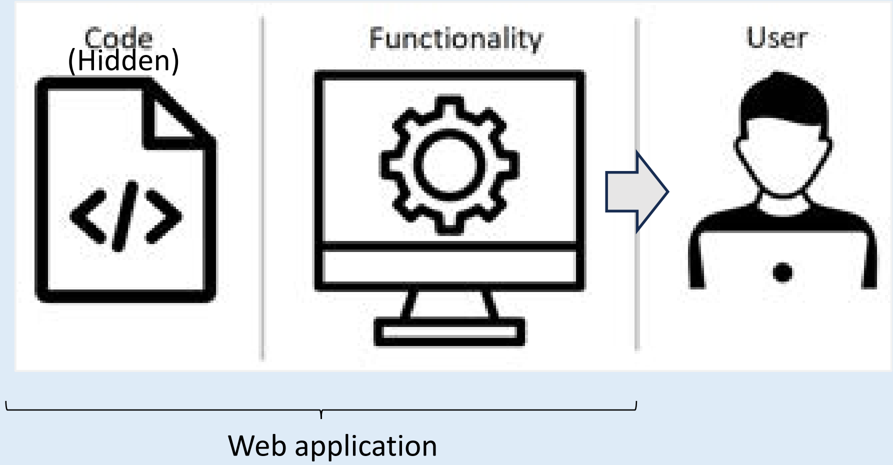
By defining methods to access and modify the data, you can enforce rules and constraints on data.

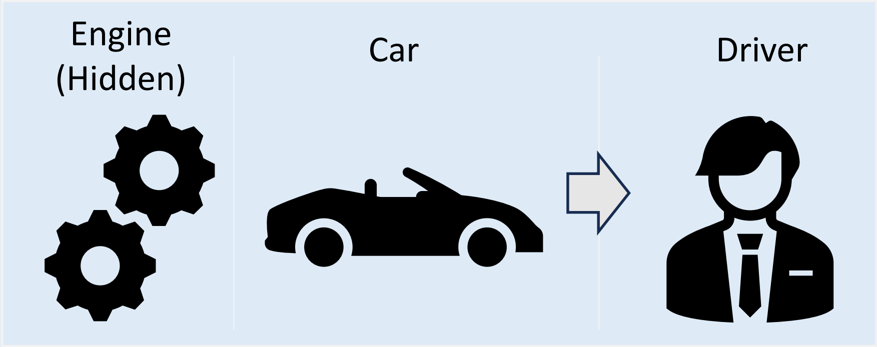
back to chapter index

# Q. What is Abstraction? How to implement abstraction?

- Abstraction means showing only required things and **hide the background(complex implementation) details**.

- Mostly we use **abstract classes** and **interfaces** for implementing abstraction.





back to chapter index

# Q. What is the difference between abstraction and encapsulation? V. IMP.

| Abstraction | Encapsulation |
|---|---|
| 1. Abstraction means showing only required things and hide the background(complex implementation) details. | Encapsulation is the bundling of data(attributes or fields) and functions into a single unit(class). |
| 2. It simplify complexity by hiding unnecessary details. | It protects the data by restricting its access only via the functions. |
| 3. It can be achieved by using abstract classes and interfaces. | It can be achieved by using access modifier and  getters-setters' methods. |

back to chapter index

# 13. Abstract class & Interface

Q. What is abstract class In Java? How to implement it?

Q. When to use abstract class in real applications? **V. IMP.**

Q. What are interfaces in Java? How to implement it?

Q. When to use interfaces in real applications? **V. IMP.**

Q. What are the differences between an Abstract class & an Interface? **V. IMP.**

Q. What are default methods? When to use default methods?

Q. Can you create an instance of an Abstract class or an Interface?

Q. Do abstract class can have Constructors? What is the use of that constructor?

Q. Do Interface can have a constructor?

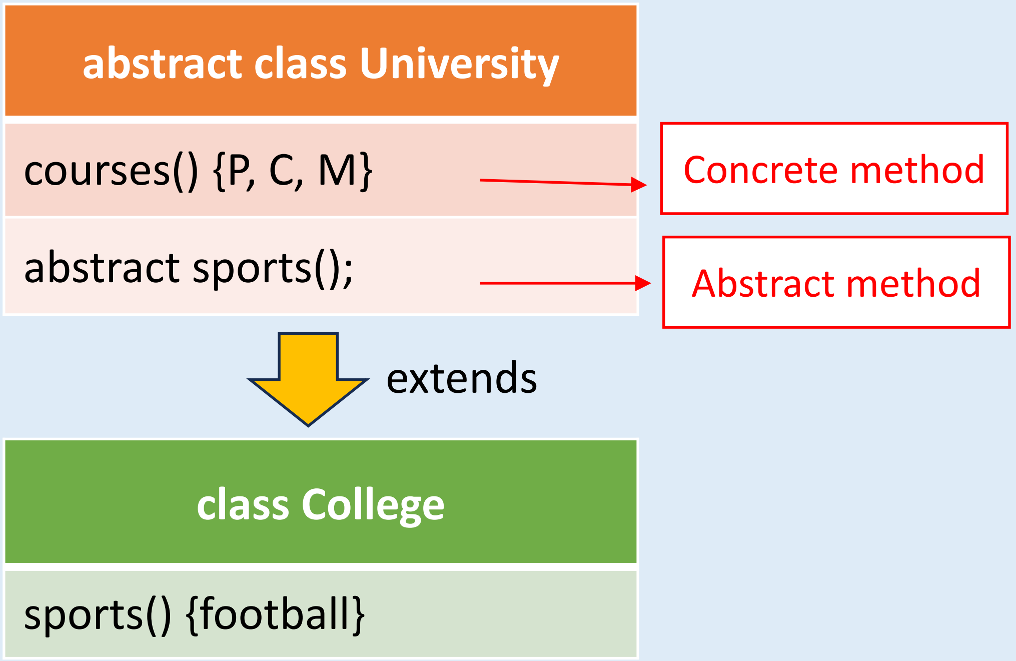
Q. When to use Interface and when Abstract class in real applications? **V. IMP.**

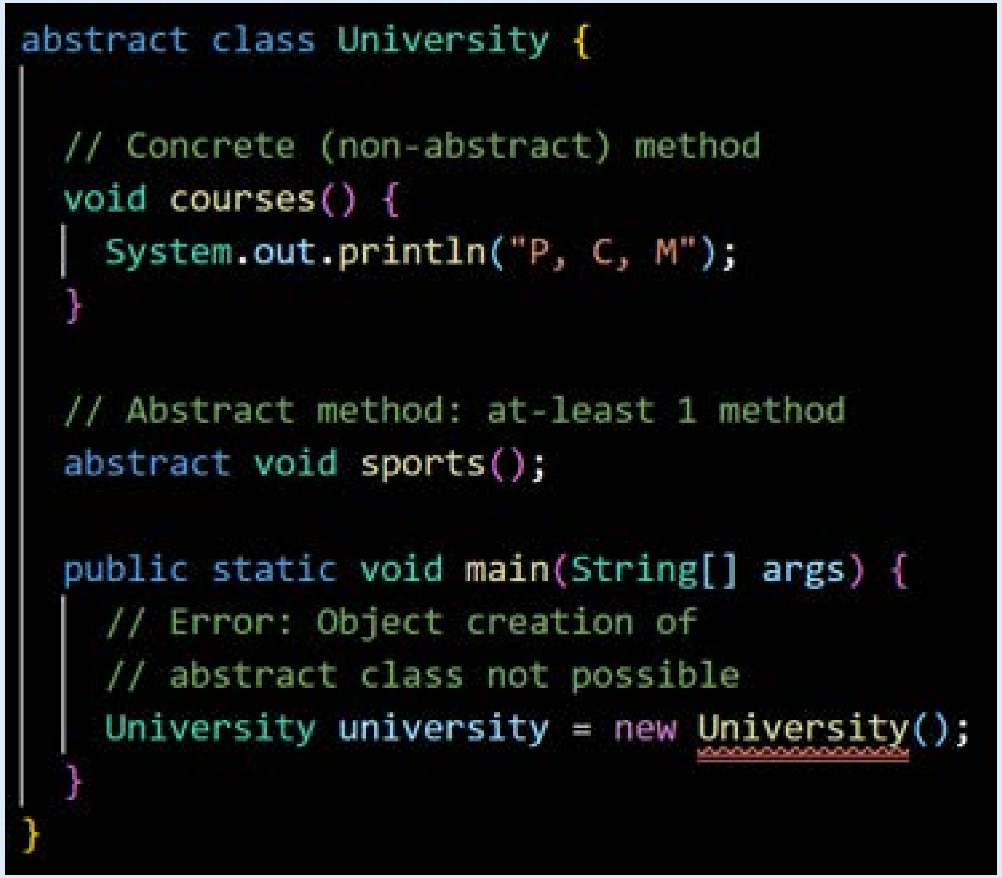
Q. How to achieve abstraction? Difference btw abstraction and abstract class?

Back to main index

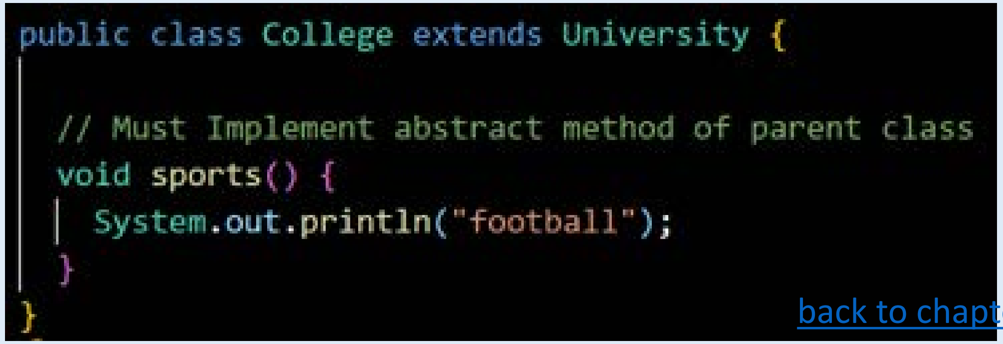
# Q. What is abstract class In Java? How to implement it?

- Abstract classes contain **both abstract and concrete methods** and abstract class object creation is not possible.
- Abstract methods of abstract class must be implemented by child class.





```java
abstract class University {

  // Concrete (non-abstract) method
  void courses() {
    System.out.println("P, C, M");
  }

  // Abstract method: at-least 1 method
  abstract void sports();

  public static void main(String[] args) {
    // Error: Object creation of
    // abstract class not possible
    University university = new University();
  }
}
```

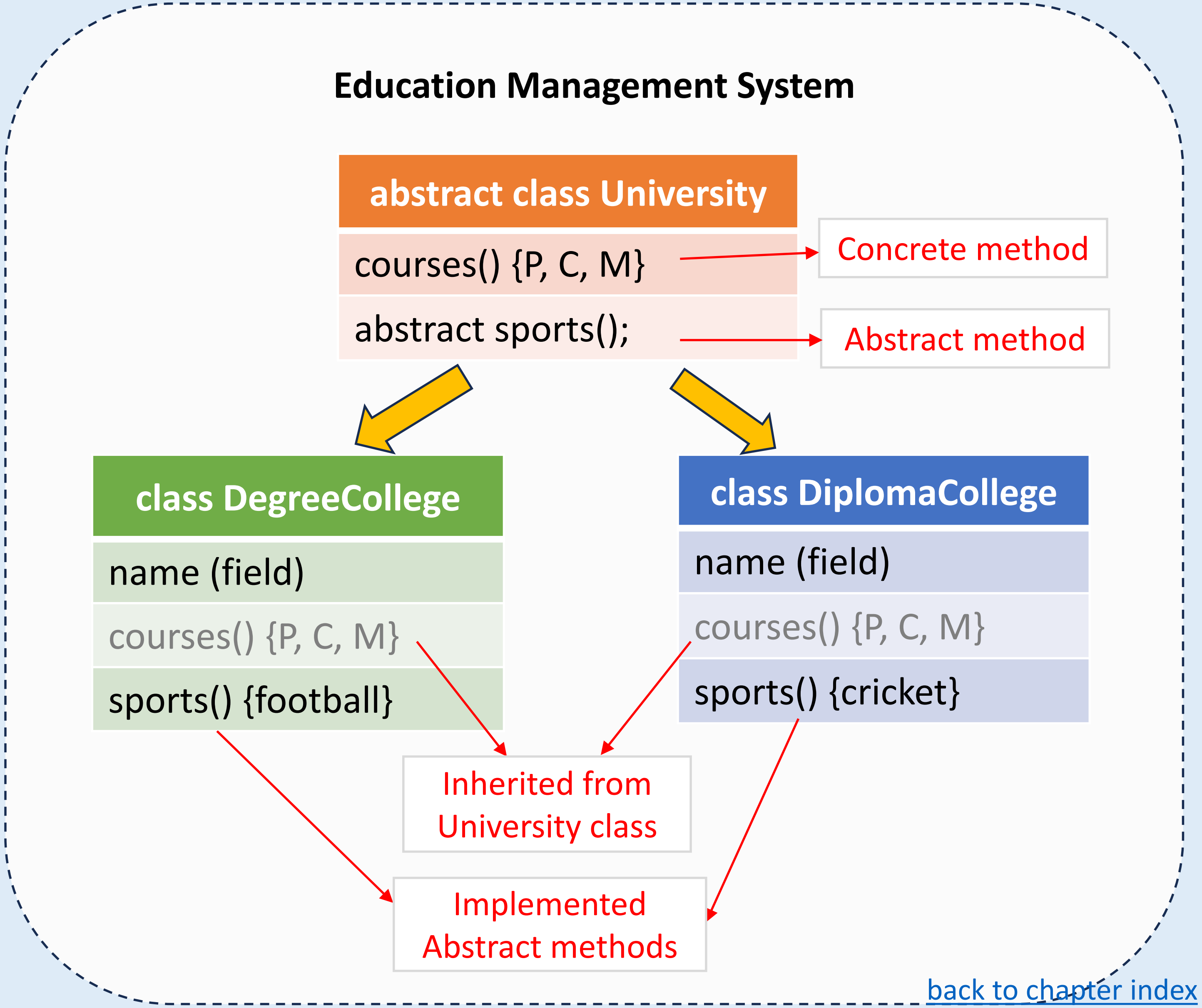


```java
public class College extends University {

  // Must Implement abstract method of parent class
  void sports() {
    System.out.println("football");
  }
}
```

back to chapter index

# Q. When to use abstract class in real applications? V. IMP.

- Abstract class is a good choice when you are sure some methods are concrete/defined and must be implemented in the same way in all derived classes. And some methods are abstract and can be implemented differently by implementing classes.



Education Management System

abstract class University

courses() {P, C, M}

abstract sports();

Concrete method

Abstract method

class DegreeCollege

name (field)

courses() {P, C, M}

sports() {football}

class DiplomaCollege

name (field)

courses() {P, C, M}

sports() {cricket}

Inherited from University class

Implemented Abstract methods

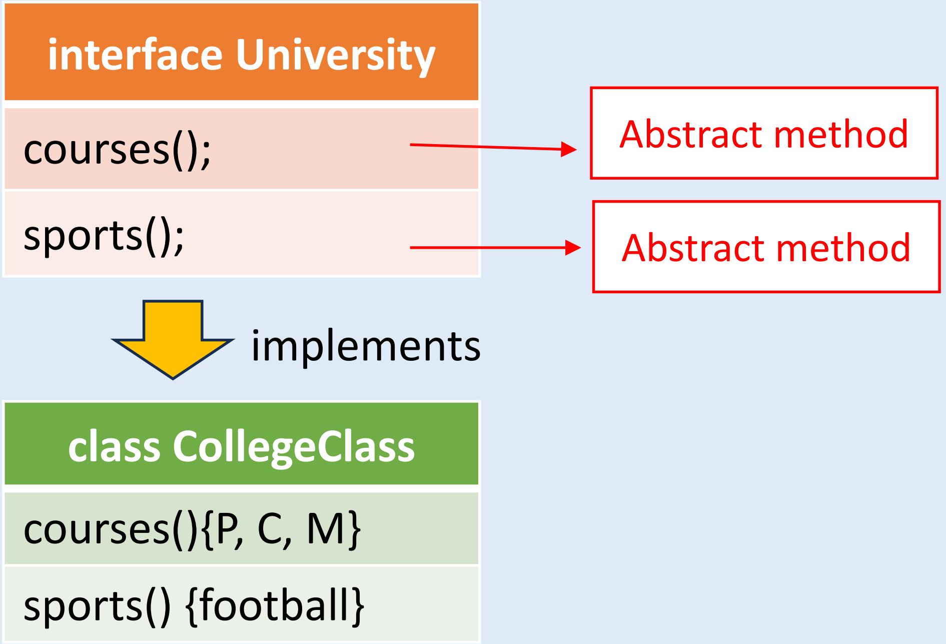
back to chapter index

# Q. What are interfaces in Java? How to implement it?

- Interfaces contain abstract methods which must be implemented in the implementing class.

- Two points about Interfaces:

1. Interfaces define a contract(rules) for implementing classes.
2. Multiple Inheritance via Interfaces is possible.



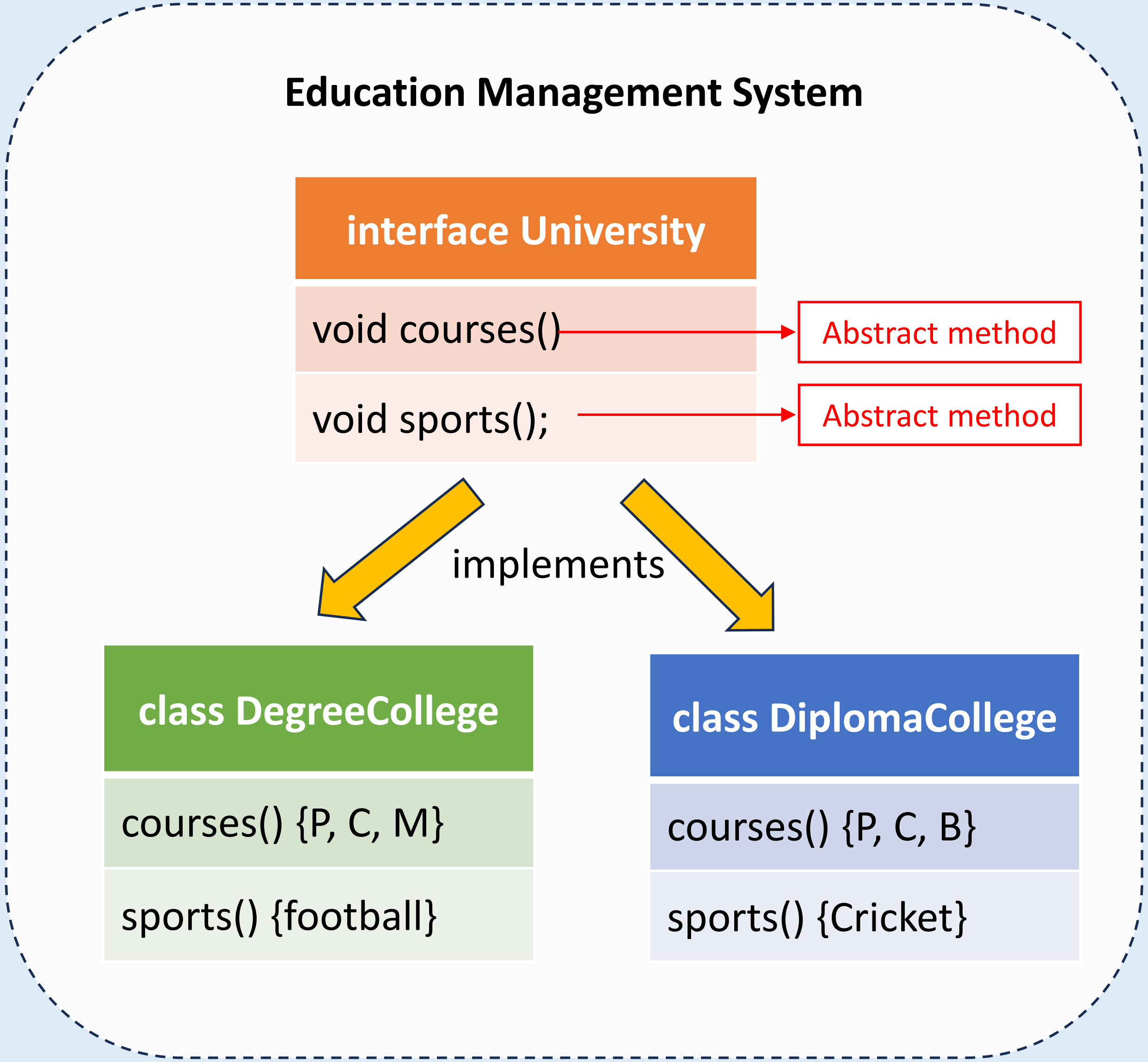
```java
public interface University {

  void courses(); // abstract method

  void sports(); // abstract method
}
```

```java
public class College implements University {

  // Must implement abstract methods
  public void courses() {
    System.out.println("P, C, M");
  }

  public void sports() {
    System.out.println("football");
  }
}
```

back to chapter index

# Q. When to use interfaces in real applications? V. IMP.

- An interface can be used to define contracts which the implementing class must implement. This will make classes more modular and structured.



back to chapter index

# Q. What are the differences between an Abstract class & an Interface (atleast 4)? V. IMP.

## Interface

```
public interface University {

  void courses(); // abstract method

  void sports(); // abstract method
}
```

```
public class College implements University {

  // Must implement abstract methods
  public void courses() {
    System.out.println("P, C, M");
  }

  public void sports() {
    System.out.println("football");
  }
}
```

## Abstract Class

```
abstract class University {
  void courses() { // Concrete method
    System.out.println(“P, C, M");
  }

  abstract void sports(); // Abstract method
}
```

```
public class College extends University {

  // Must Implement abstract method
  void sports() {
    System.out.println("football");
  }

}
```

back to chapter index

# Q. What are the differences between an Abstract class & an Interface (atleast 4)? V. IMP.

| Interfaces | Abstract Classes |
|---|---|
| Cannot be instantiated directly with new keyword. | |
| 1. Defined by using interface keyword. | Defined by using abstract keyword |
| 2. Supports multiple inheritance (can implement multiple interfaces). | Supports single inheritance (extends only one class) |
| 3. Methods are implicitly abstract | Use abstract keyword to make methods abstract |
| 4. Cannot have constructors. | Can have constructors. |
| 5. Can have default methods. | Cannot have default methods |

back to chapter index

# Q. What are default methods? When to use default methods?

- **Definition**: Default methods are methods in interfaces that have a **default implementation**.
- They are declared using the default keyword.
- **Purpose**: Default methods in Java interfaces introduced in Java 8 provide a way to add concrete method implementations to interfaces, avoiding the need to convert interfaces into abstract classes for such purposes.

```java
public interface Vehicle {
  // Abstract method
  void start();

  // Default method
  default void stop() {
    System.out.println("Stop...");
  }
}
```

```java
public class Car implements Vehicle {

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

  public static void main(String[] args) {
    Car myCar = new Car();
    myCar.start(); // Output: Start...
    myCar.stop(); // Output: Stop...
  }
}
```

back to chapter index

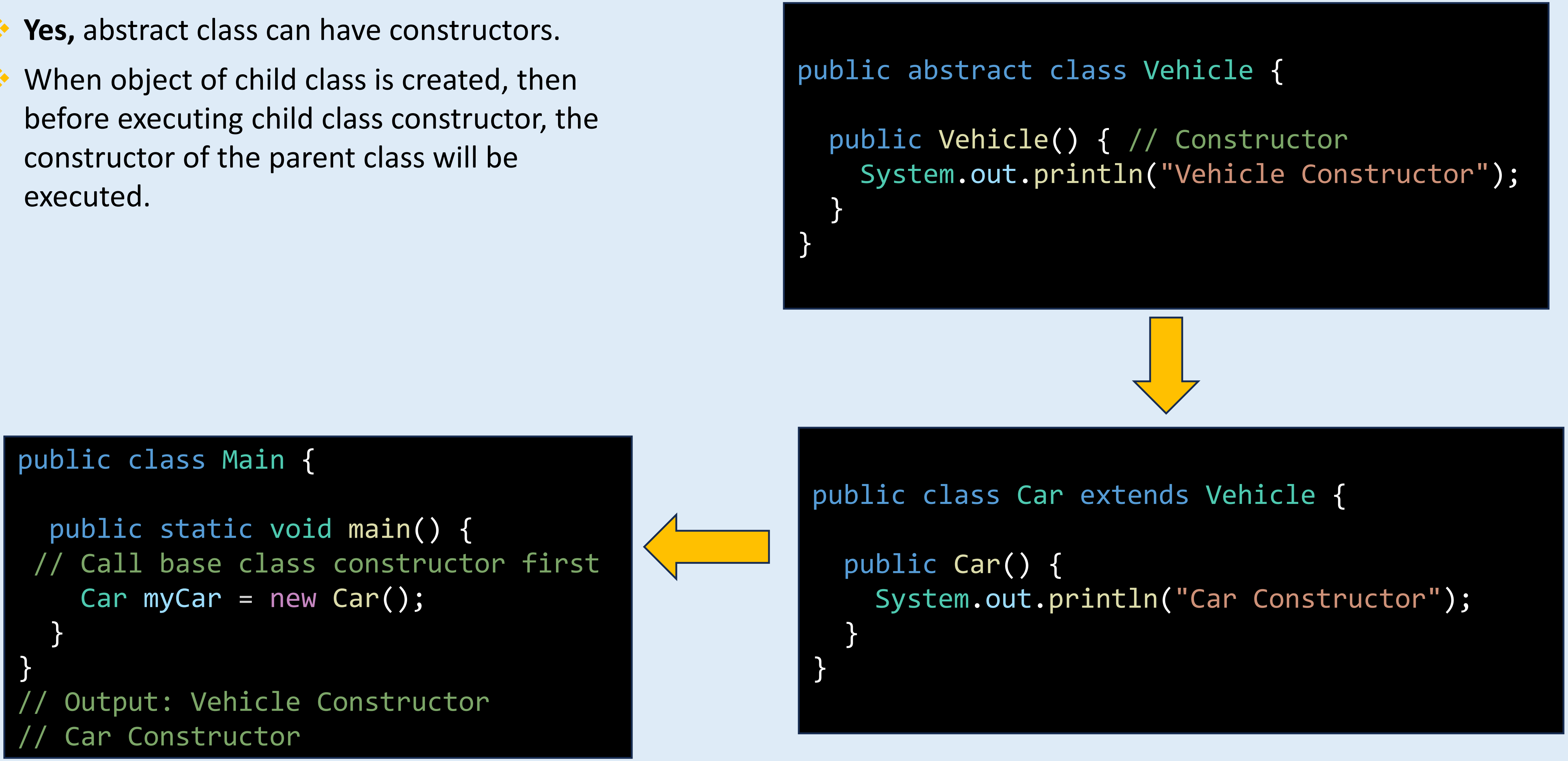
# Q. Can you create an instance of an Abstract class or an Interface?

- **No,** creating instance of abstract class and interface is not possible.

back to chapter index

# Q. Do abstract class can have Constructors? What is the use of that constructor?

- **Yes,** abstract class can have constructors.
- When object of child class is created, then before executing child class constructor, the constructor of the parent class will be executed.



```java
public abstract class Vehicle {

  public Vehicle() { // Constructor
    System.out.println("Vehicle Constructor");
  }
}
```

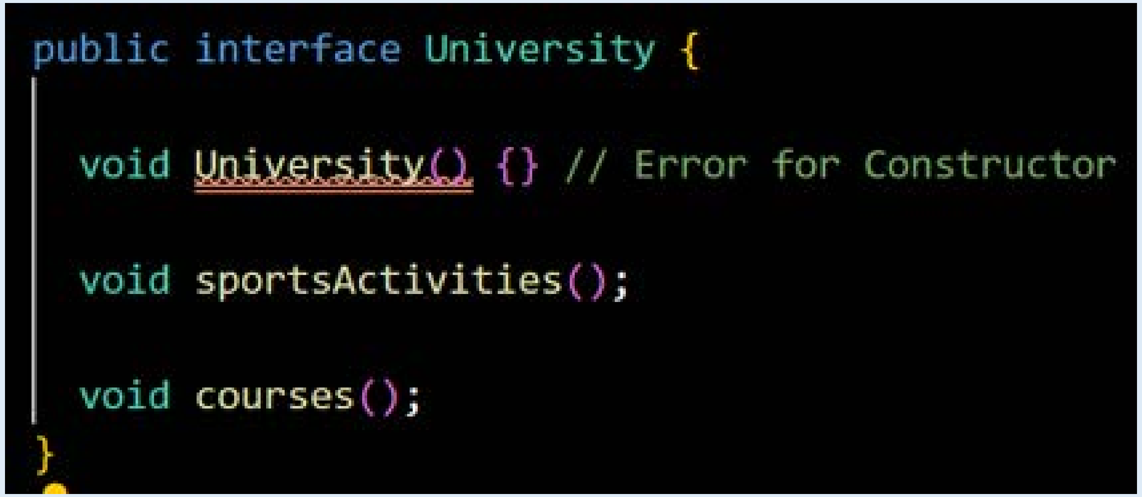
```java
public class Car extends Vehicle {

  public Car() {
    System.out.println("Car Constructor");
  }
}
```

```java
public class Main {

  public static void main() {
 // Call base class constructor first
    Car myCar = new Car();
  }
}
// Output: Vehicle Constructor
// Car Constructor
```

back to chapter index

# Q. Do Interface can have a constructor?

- **NO,** Interface can not have constructors.



```
public interface University {

   void University() {} // Error for Constructor

   void sportsActivities();

   void courses();
}
```

back to chapter index

# Q. When to use Interface and when Abstract class in real applications?

## When to use Interface?

1. An interface can be used to **define contracts** which the implementing class must implement. This will make classes more modular and structured.

2. Interface is a good choice when you know a method has to be there, but it can be **implemented differently** by implementing classes.

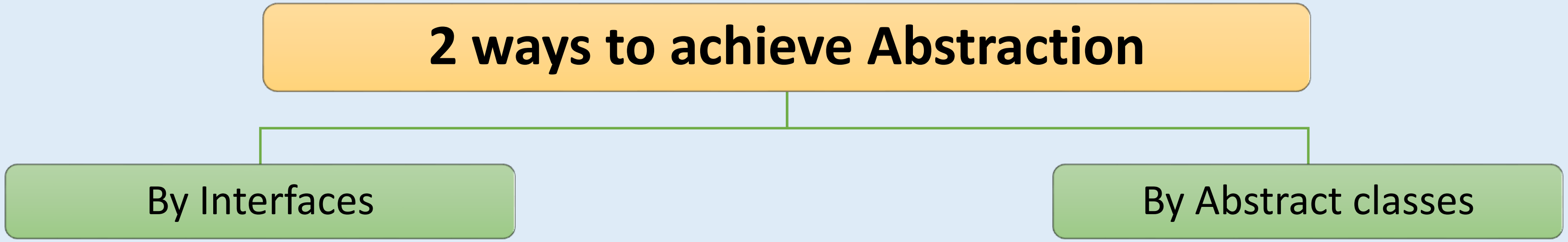
3. Interfaces do separation of concerns and because of that **unit testing is simpler**.

## When to use Abstract class?

1. Abstract class is a good choice when you are sure some methods are concrete/defined and must be implemented in the **same way** in all derived classes. And some methods are abstract and can be **implemented differently** by implementing classes.

back to chapter index

# Q. How to achieve abstraction? Difference between abstraction and abstract class?



```
public interface University {

  // interfaces provide complete
  // abstraction: methods showing features
  // but hiding implementation
  void sports();

  void courses();
}
```

```
abstract class University {

 // abstract class provide partial abstraction

 abstract void sports(); // Abstract method

  void courses() {// Concrete method
    System.out.println(“P, C, M");
  }
}
```

- Difference between abstraction and abstract class?
- **Abstraction is a broader concept** of hiding complex implementation details and showing only the essential features, whereas **abstract class is a practical way** to implement Abstraction.

back to chapter index

# 14. Constructors

Q. What is Constructor in Java? V. IMP.

Q. What are the types of constructors? What is Default constructor? V. IMP.

Q. What is Parameterized constructor? When to use it in real applications? V. IMP.

Q. What is constructor overloading? When to use it in real applications?

Q. What is constructor chaining?

Q. What is copy constructor?

Q. When to use copy constructor in real applications?

Q. Can a constructor have a return type?

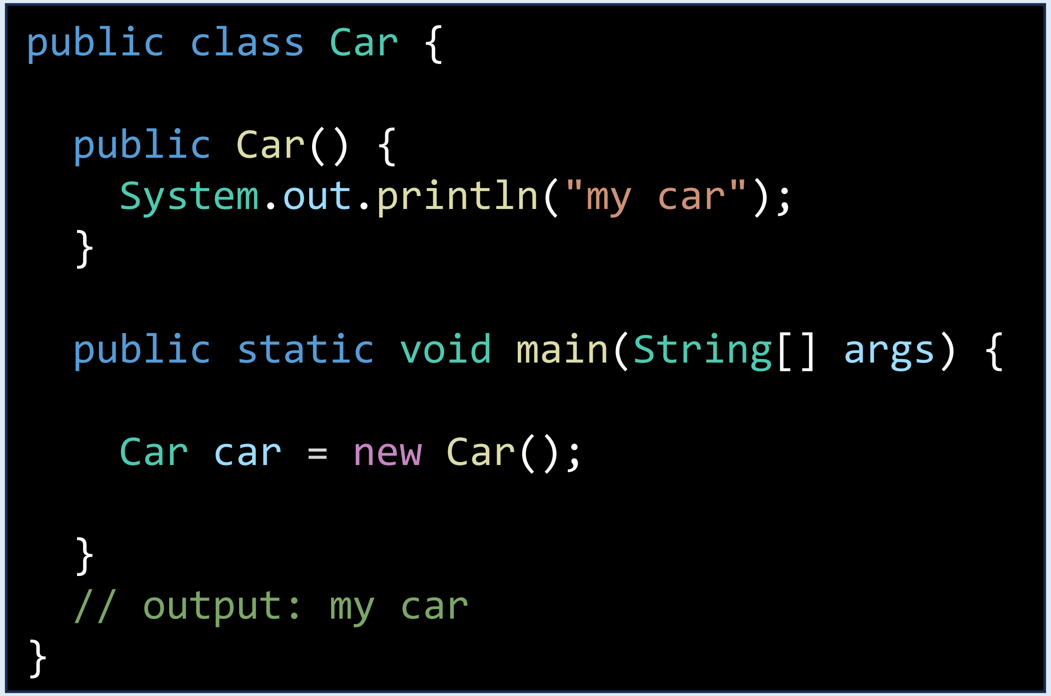
Q. What will happen if no constructor is defined inside the class?

Q. What is the role of super keyword?

Back to main index

# Q. What is Constructor in Java? V. IMP.

- A constructor is a special type of method that is called when the object of class is created.
- A constructor has the same name as the class and does not have a return type, not even void.

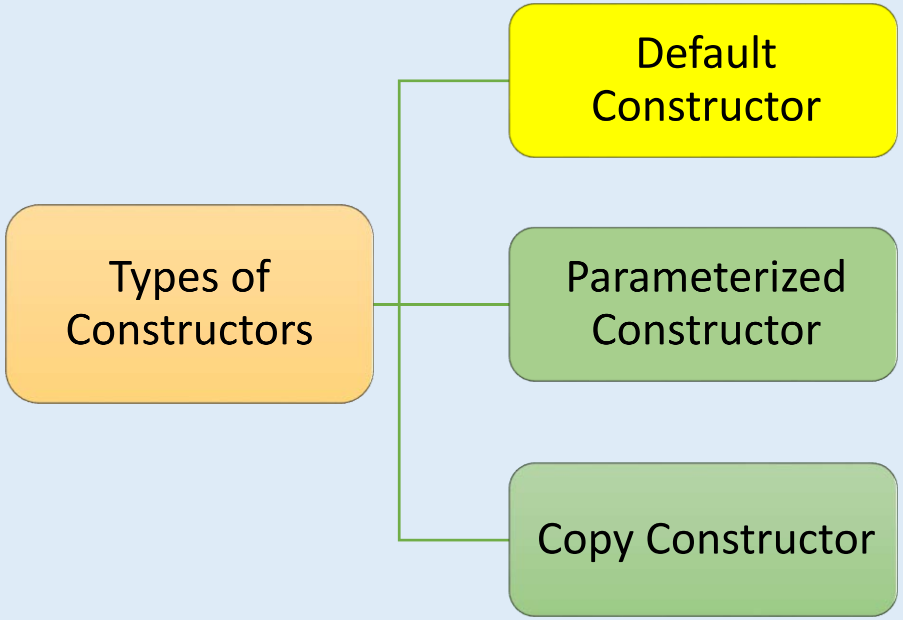


```java
public class Car {

  public Car() {
    System.out.println("my car");
  }

  public static void main(String[] args) {

    Car car = new Car();

  }
  // output: my car
}
```

back to chapter index

# Q. What are the types of constructors? What is Default constructor? V. IMP.

- A default constructor in Java is a constructor which is **automatically provided by Java** if no other constructors are explicitly defined.



```
public class Car {

 }
```

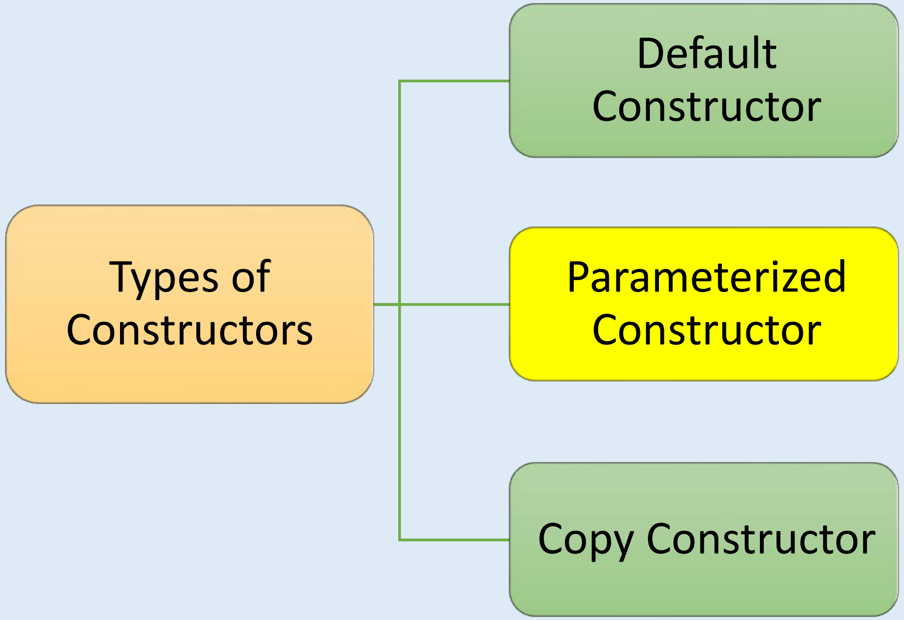
JVM (JIT Compiler)

```
public class Car {

    // Default Constructor
    public Car() { }

 }
```

back to chapter index

# Q. What is Parameterized constructor? When to use it in real applications? V. IMP.

- **Definition:** A parameterized constructor is a constructor **with parameters**.
- **Use:** It is used to initialize the specific field values of the class provided during object creation.



```java
public class Employee {

  private String name;

  // Parameterized constructor
  public Employee(String name) {
    this.name = name; // field initialization
  }

  public String getName() {
    return name;
  }

  public static void main(String[] args) {

    Employee emp = new Employee("Happy");

    System.out.println(emp.getName());
  }
}
```

back to chapter index

# Q. What is constructor overloading? When to use it in real applications?

- **Definition:** Constructor overloading is the concept of having **multiple constructors** within the same class, each with a different parameter list.
- **Use:** Constructor overloading **enhances the usability of classes** by providing multiple ways to initialize objects based on varying requirements and input parameters.

```
public class Rectangle {
  private int width;
  private int height;

  public Rectangle(int width, int height) {
    this.width = width;
    this.height = height;
  }

  public Rectangle(int side) {
    this.width = side;
    this.height = side;
  }

  public static void main(String[] args) {

    Rectangle rec = new Rectangle(10, 20);
    System.out.println(rec.width * rec.height); // 200

    Rectangle squ = new Rectangle(5);
    System.out.println(squ.width * squ.height); // 25
  }
}
```

back to chapter index

# Q. What is constructor chaining?

- Constructor chaining is the process of one constructor calling another constructor from the same class.

```java
public class Vehicle {

  private String brand;

  public Vehicle(String brand) {
    this.brand = brand;
  }

  public Vehicle() {
    // Constructor chaining (Calls the other constructor)
    this("Honda");
  }

  public String getBrand() {
    return brand;
  }

  public static void main(String[] args) {
    Vehicle vehicle = new Vehicle();
    System.out.println(vehicle.getBrand()); // Honda
  }
}
```

back to chapter index

# Q. What is copy constructor?

- A copy constructor is used to create a new object as a copy of an existing object.
- A copy constructor typically accepts an instance of the same class as a parameter.

```java
public class Student {

  private String name;

  // Parameterized Constructor
  public Student(String name) {
    this.name = name;
  }

  // Copy constructor
  public Student(Student other) {
     // Copy name from the other object
     this.name = other.name;
  }

  // Getter method for name
  public String getName() {
    return name;
  }
```

```java
public static void main(String[] args) {
    Student stu1 = new Student("Happy");

    // Create object stu2 as a copy of stu1 object
    Student stu2 = new Student(stu1);

    System.out.println(stu1.getName()); //Happy
    System.out.println(stu2.getName()); //Happy
  }
}
```

back to chapter index

# Q. When to use copy constructor in real applications?

- Copy constructors are used for deep copying complex objects. Ex: for implementing clone behaviors.

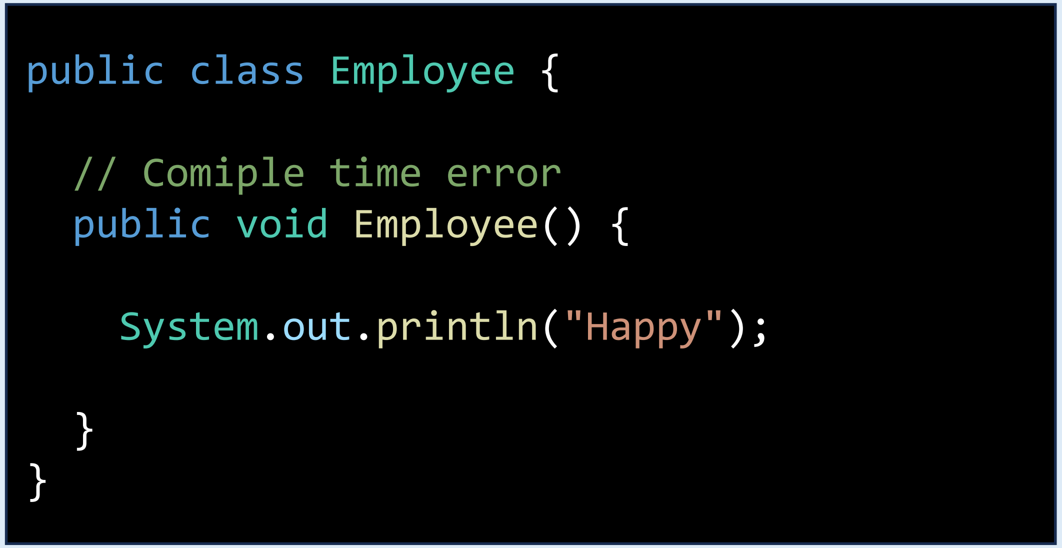
```java
public class Student {

  private String name;

  // Parameterized Constructor
  public Student(String name) {
    this.name = name;
  }

  // Copy constructor
  public Student(Student other) {
     // Copy name from the other object
     this.name = other.name;
  }

  // Getter method for name
  public String getName() {
    return name;
  }
```

```java
public static void main(String[] args) {
    Student stu1 = new Student("Happy");

    // Create object stu2 as a copy of stu1 object
    Student stu2 = new Student(stu1);

    System.out.println(stu1.getName()); //Happy
    System.out.println(stu2.getName()); //Happy
  }
}
```

back to chapter index

# Q. Can a constructor have a return type?

- No, constructors do not have a return type, not even void.

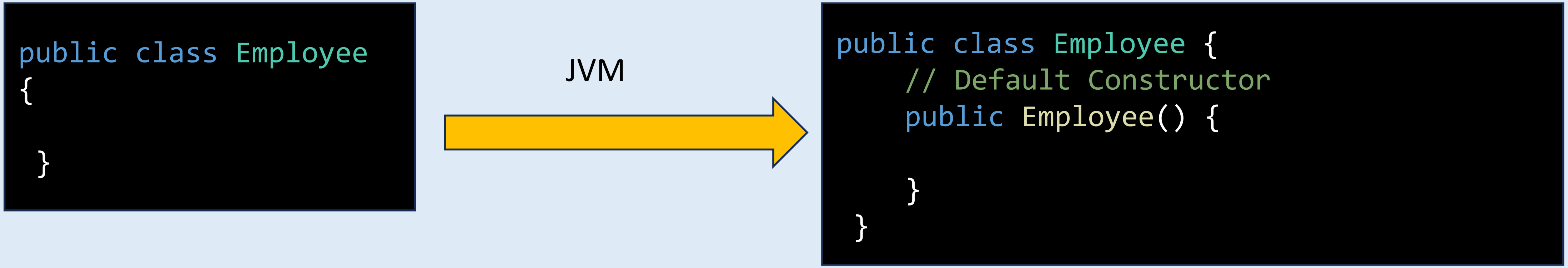


```java
public class Employee {

  // Comiple time error
  public void Employee() {

    System.out.println("Happy");

  }
}
```

back to chapter index

# Q. What will happen if no constructor is defined inside the class?

- If no constructor is explicitly defined inside a class, the Java compiler automatically provides a default constructor for that class.



```
public class Employee
{

 }
```

JVM

```
public class Employee {
    // Default Constructor
    public Employee() {

    }
 }
```

back to chapter index

# Q. What is the role of super keyword?

- Super() keyword is used to call the constructor of superclass from subclass constructor, but it is optional to write.

```
public class Superclass {

  public Superclass() {
    System.out.println("from superclass");
  }
}
```

```
public class Subclass extends Superclass {

  public Subclass() {
    super(); // (optional - automatically placed)
    System.out.println("from subclass");
  }

  public static void main(String[] args) {
    // Creating an object of Subclass
    Subclass subclass = new Subclass();
  }
}
// Output: from superclass
// from subclass
```

back to chapter index

# 15. Exception Handling - Basics

Q. What is Exception Handling? How to implement it in Java? **V. IMP.**

Q. What is the role of finally in exception handling? **V. IMP.**

Q. When to use finally in real applications?

Q. Can we have multiple catch blocks and when should we use multiple catch blocks?

Q. What is catch-all block? Is it a good practice in real applications?

Q. Can we execute all catch blocks at one time?

Back to main index

# Q. What is Exception Handling? How to implement it in Java? V. IMP.

- Exception handling in Object-Oriented Programming is used to **MANAGE ERRORS.**

- How to implement error handling:

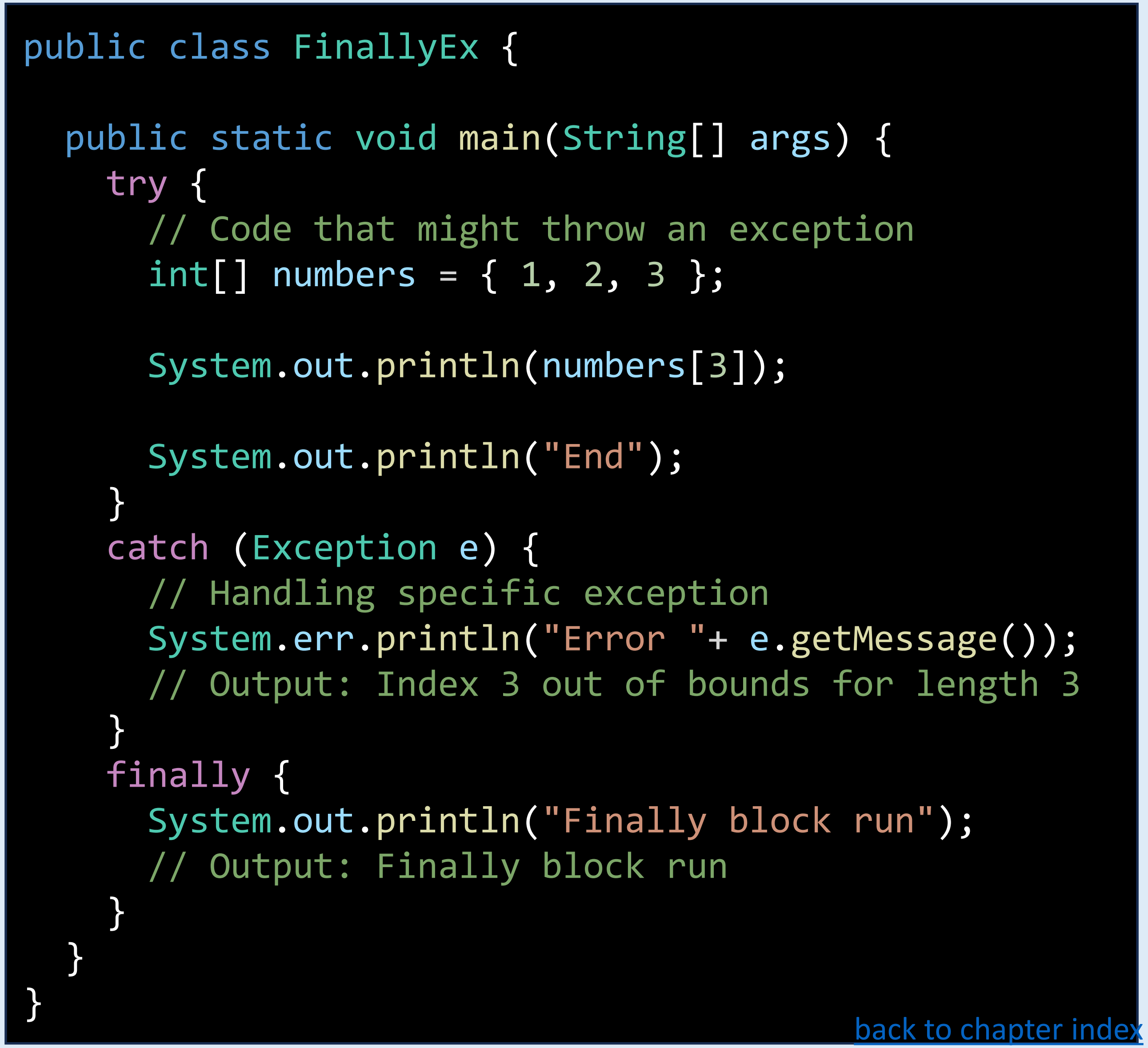
| | |
|---|---|
| **Try** | A try block is a block of code inside which any error can occur. |
| **Catch** | When any error occur in TRY block then it is passed to catch block to handle it. |

```java
public class ExceptionEx {

  public static void main(String[] args) {
    try {
      // Code that might throw an exception
      int[] numbers = { 1, 2, 3 };

      System.out.println(numbers[3]);

      System.out.println("End");
    }
    catch (Exception e) {
      // Handling specific exception
      System.err.println("Error "+ e.getMessage());
      // Output: Index 3 out of bounds for length 3
    }
  }
}
```

back to chapter index

# Q. What is the role of finally in exception handling? V. IMP.

- The finally block is used to execute a given set of statements, whether an exception occur or not.



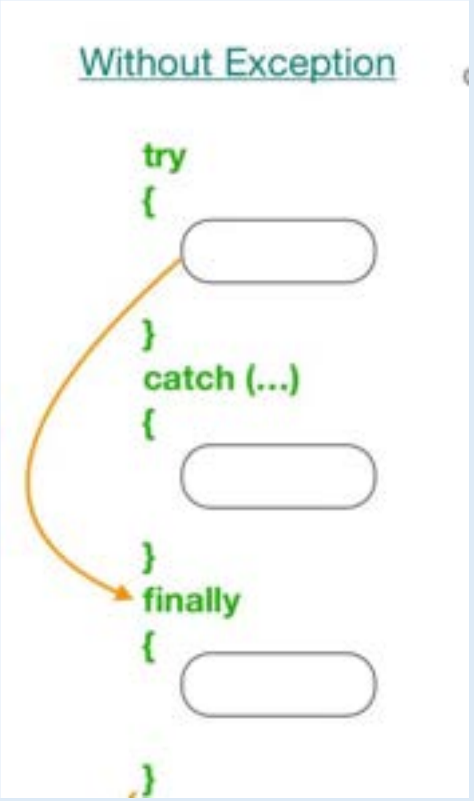
```
public class FinallyEx {

  public static void main(String[] args) {
    try {
      // Code that might throw an exception
      int[] numbers = { 1, 2, 3 };

      System.out.println(numbers[3]);

      System.out.println("End");
    }
    catch (Exception e) {
      // Handling specific exception
      System.err.println("Error "+ e.getMessage());
      // Output: Index 3 out of bounds for length 3
    }
    finally {
      System.out.println("Finally block run");
      // Output: Finally block run
    }
  }
}
```

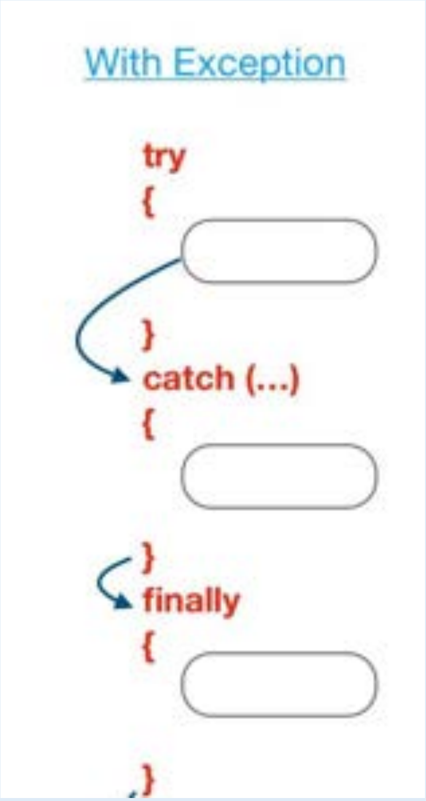
back to chapter index

# Q. When to use finally in real applications?

- Finally block is mostly used for **cleaning up resources**. For example:

1. Closing DB connections
2. Closing IO resources
3. Logging





back to chapter index

# Q. Can we have multiple catch blocks and when should we use multiple catch blocks?

- Multiple catch blocks are used to:

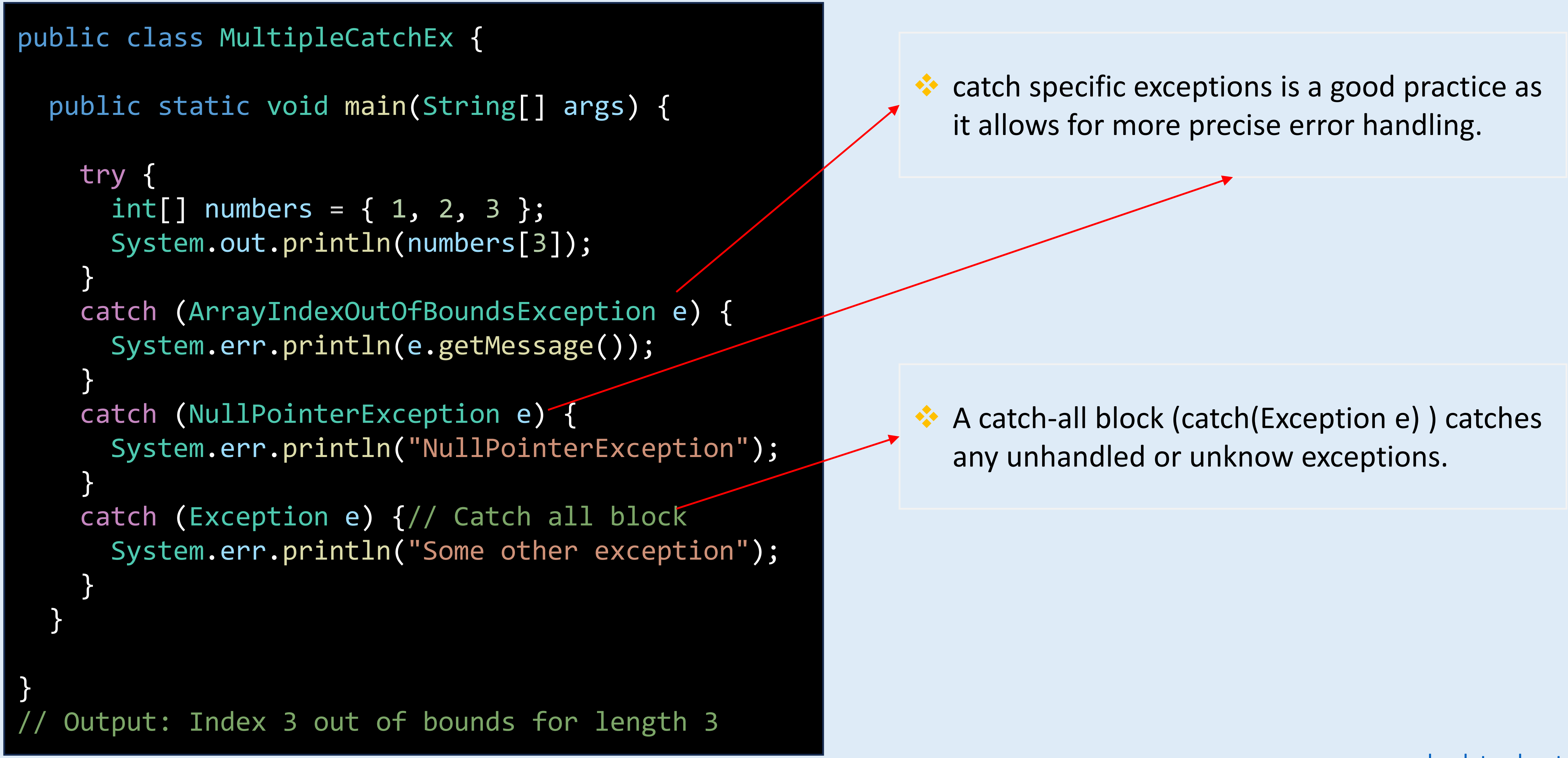
1. Handling Different Types of Exceptions

3. Avoiding Catch-All Blocks

```
public class MultipleCatchEx {

  public static void main(String[] args) {

    try {
      int[] numbers = { 1, 2, 3 };
      System.out.println(numbers[3]);
    }
    catch (ArrayIndexOutOfBoundsException e) {
      System.err.println(e.getMessage());
    }
    catch (NullPointerException e) {
      System.err.println("NullPointerException");
    }
  }

}
// Output: Index 3 out of bounds for length 3
```

back to chapter index

# Q. What is catch-all block? Is it a good practice in real applications?



```
public class MultipleCatchEx {

  public static void main(String[] args) {

    try {
      int[] numbers = { 1, 2, 3 };
      System.out.println(numbers[3]);
    }
    catch (ArrayIndexOutOfBoundsException e) {
      System.err.println(e.getMessage());
    }
    catch (NullPointerException e) {
      System.err.println("NullPointerException");
    }
    catch (Exception e) {// Catch all block
      System.err.println("Some other exception");
    }
  }

}
// Output: Index 3 out of bounds for length 3
```

- catch specific exceptions is a good practice as it allows for more precise error handling.

- A catch-all block (catch(Exception e) ) catches any unhandled or unknow exceptions.

back to chapter index

# Q. Can we execute all catch blocks at one time?

- No, in Java, only one catch block will be executed for a given try block execution, even if multiple catch blocks are present.

```java
public class MultipleCatchEx {

  public static void main(String[] args) {
    try {
      int[] numbers = { 1, 2, 3 };
      System.out.println(numbers[3]);
    }
    catch (ArrayIndexOutOfBoundsException e) {
      System.err.println(e.getMessage());
    }
    catch (NullPointerException e) {
      System.err.println("NullPointerException");
    }
    catch (Exception e) {
      System.err.println("Some other exception");
    }
  }
}
// Output: Index 3 out of bounds for length 3
```

back to chapter index

# 16. Exception Handling - Advanced

Q. What is the role of throw keyword in exception handling? When to use it? **V. IMP.**

Q. What is the role of throws keyword in exception handling?

Q. What are the differences between throw and throws keywords? **V. IMP.**

Q. What are the types of exceptions in Java?

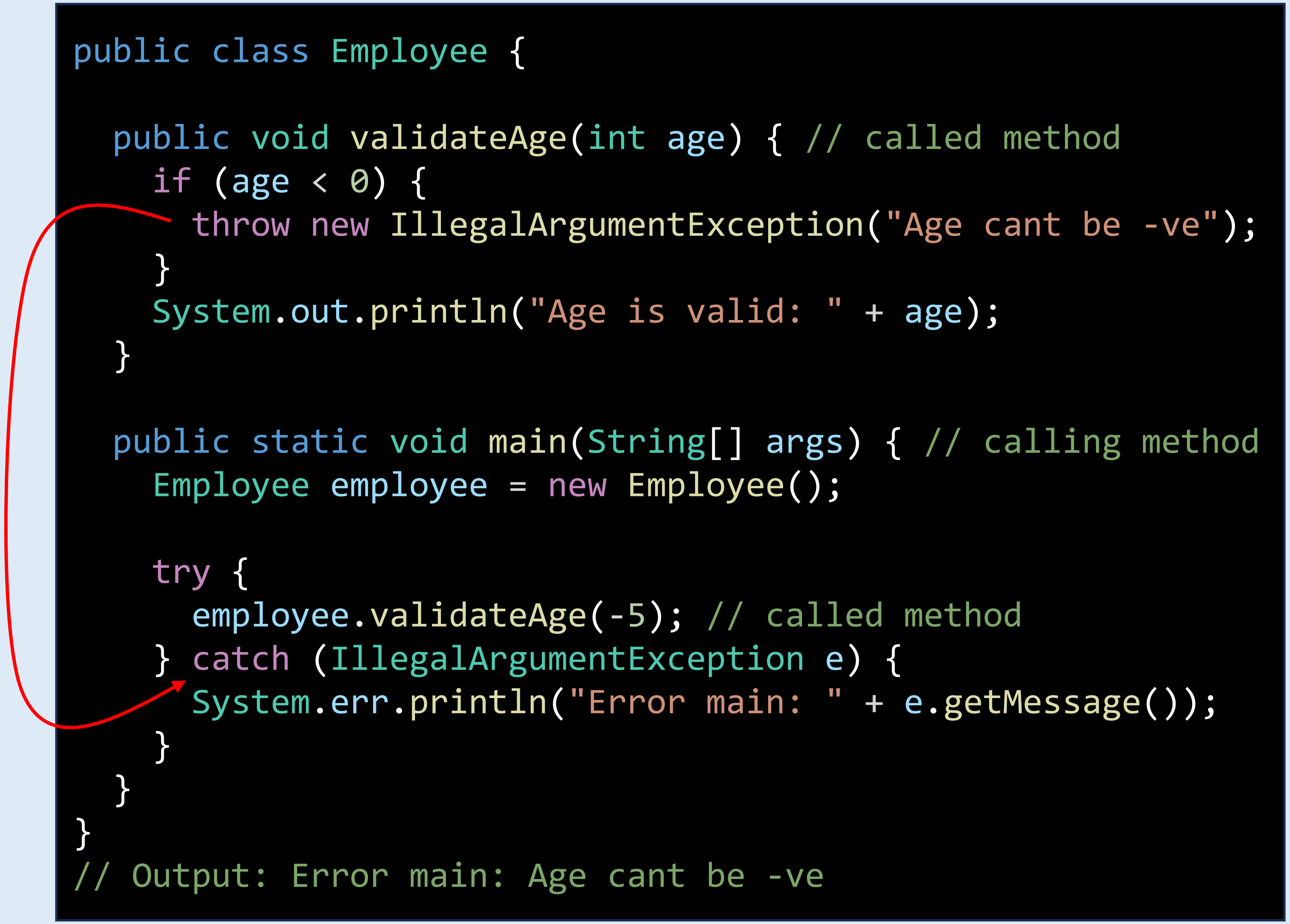
Q. What are checked and unchecked exceptions? What is the difference between them?

Back to main index

# Q. What is the role of throw keyword in exception handling? When to use it? V. IMP.

- The throw keyword to pass or propagate an exception from a lower-level function (nested function) to higher-level functions (outer function).

- This allows for centralized handling of exceptions.

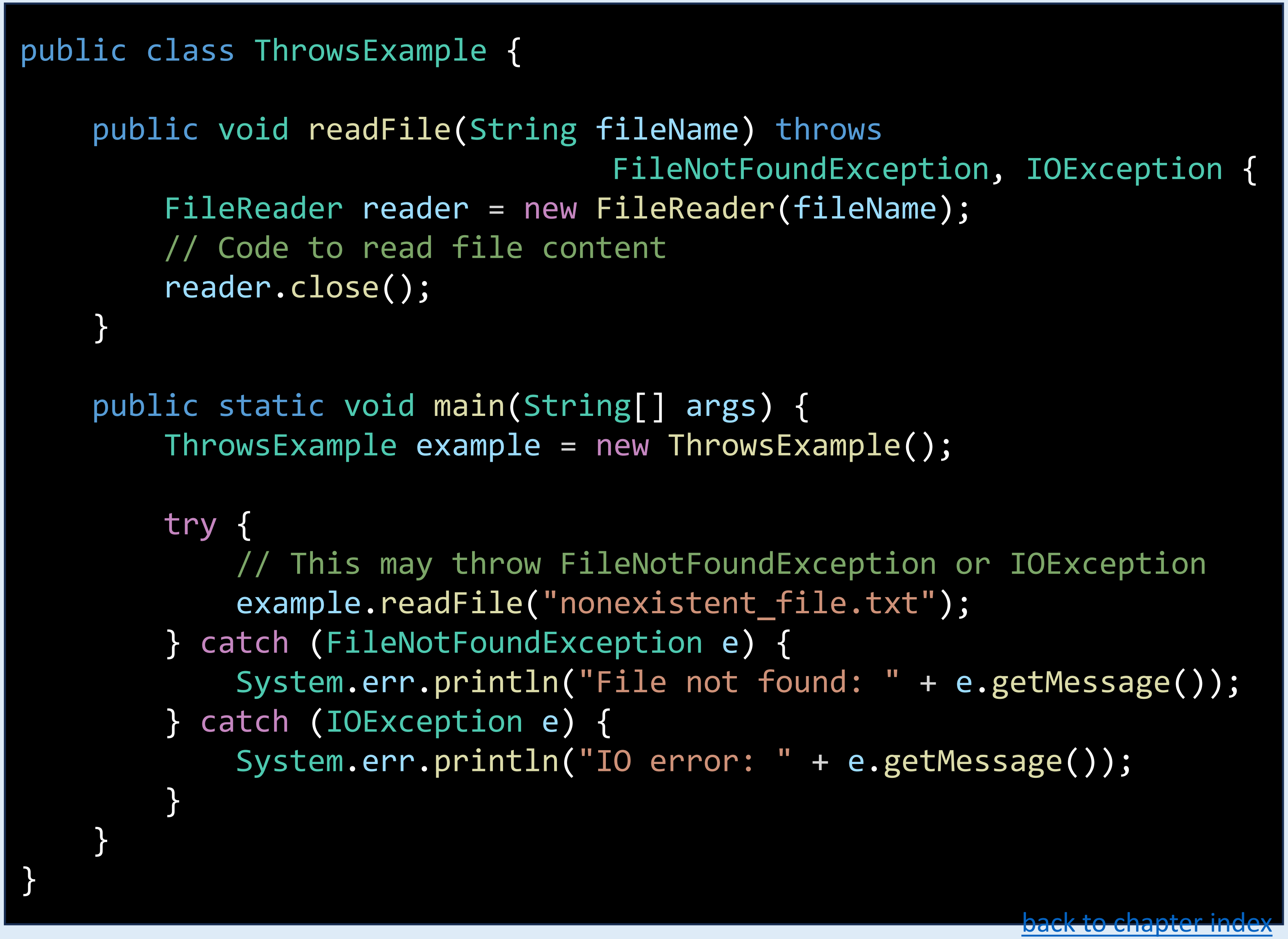


```
public class Employee {

  public void validateAge(int age) { // called method
    if (age < 0) {
      throw new IllegalArgumentException("Age cant be -ve");
    }
    System.out.println("Age is valid: " + age);
  }

  public static void main(String[] args) { // calling method
    Employee employee = new Employee();

    try {
      employee.validateAge(-5); // called method
    } catch (IllegalArgumentException e) {
      System.err.println("Error main: " + e.getMessage());
    }
  }
}
// Output: Error main: Age cant be -ve
```

back to chapter index

# Q. What is the role of throws keyword in exception handling?

- throws keyword is used in method declarations to specify that the method may throw certain types of exceptions during its execution.



```java
public class ThrowsExample {

    public void readFile(String fileName) throws
                                        FileNotFoundException, IOException {
        FileReader reader = new FileReader(fileName);
        // Code to read file content
        reader.close();
    }

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

        try {
            // This may throw FileNotFoundException or IOException
            example.readFile("nonexistent_file.txt");
        } catch (FileNotFoundException e) {
            System.err.println("File not found: " + e.getMessage());
        } catch (IOException e) {
            System.err.println("IO error: " + e.getMessage());
        }
    }
}
```

back to chapter index

# Q. What are the differences between throw and throws keywords? V. IMP.

```java
public class Employee {

  public void validateAge(int age) {
    if (age < 0) {
      throw new IllegalArgumentException(“Age");
    }
    System.out.println("Age is valid: " + age);
  }

  public static void main(String[] args)
{    Employee employee = new Employee();

    try {
      employee.validateAge(-5);
    } catch (IllegalArgumentException e) {
      System.err.println(e.getMessage());
    }
  }
}
// Output: Age
```

```java
public class ThrowsExample {

    public void readFile(String fileName) throws
                 FileNotFoundException, IOException {
        FileReader reader = new FileReader(fileName);
        // Code to read file content
        reader.close();
    }

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

        try {
            example.readFile("nonexistent_file.txt");
        } catch (FileNotFoundException e) {
            System.err.println("File not found”);
        } catch (IOException e) {
            System.err.println("IO error");
        }
    }
}
```

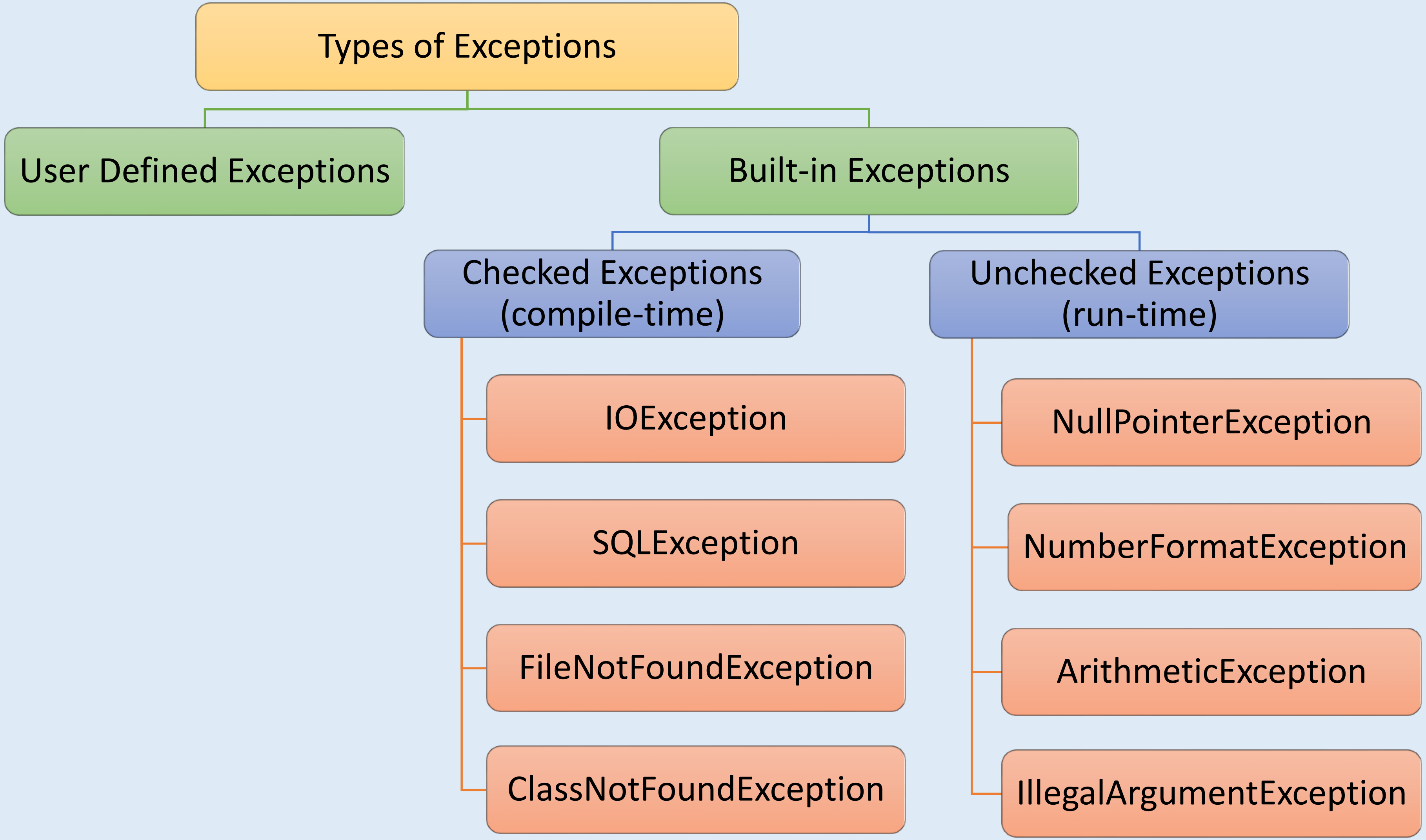
back to chapter index

# Q. What are the differences between throw and throws keywords? V. IMP.

| throw | throws |
|---|---|
| Both are used to throw exception. | |
| 1. Used to explicitly throw an exception | Used to declare potential exceptions |
| 2. Used within a method to raise an exception | Used in method declaration(signature) . |
| 3. Raise an exception at a specific point in the code | Indicates which exceptions a method may raise |
| 4. Syntax: throw new SomeException(); | Syntax: public void methodName() throws SomeException; |

back to chapter index

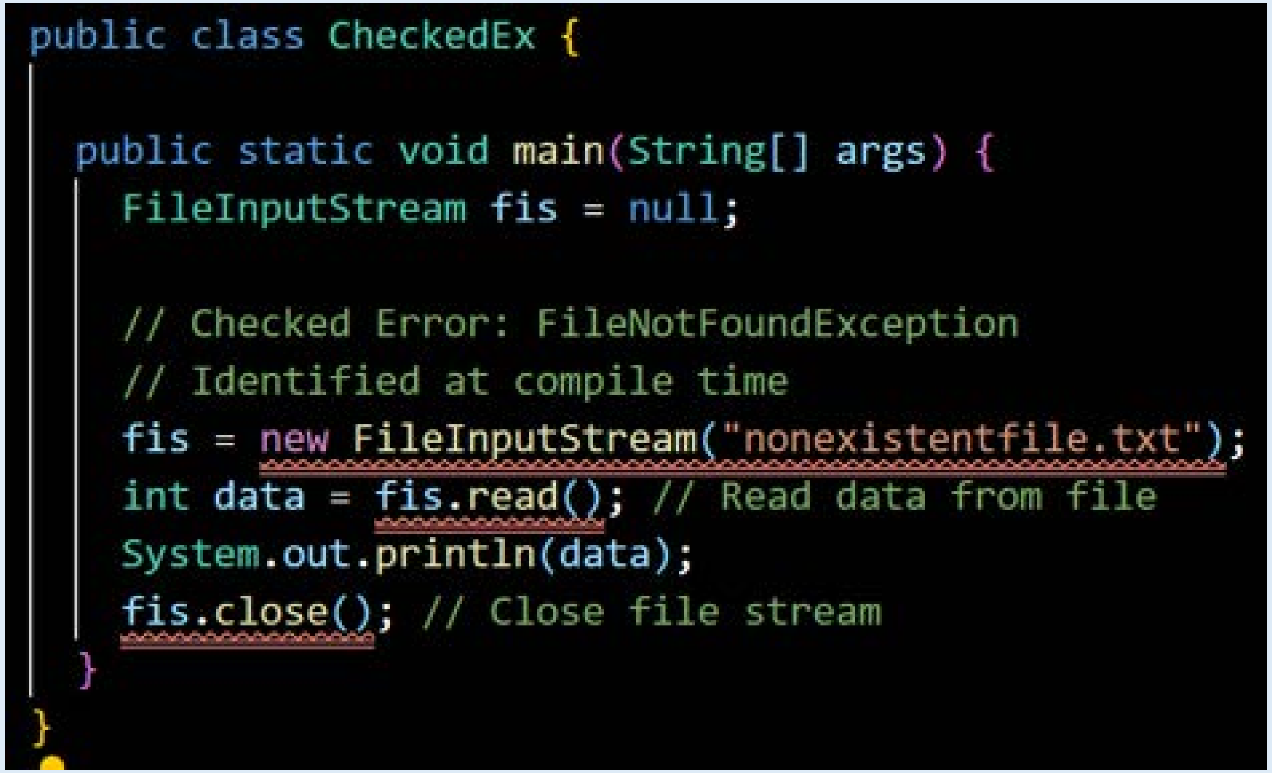
# Q. What are the types of exceptions in Java?



back to chapter index

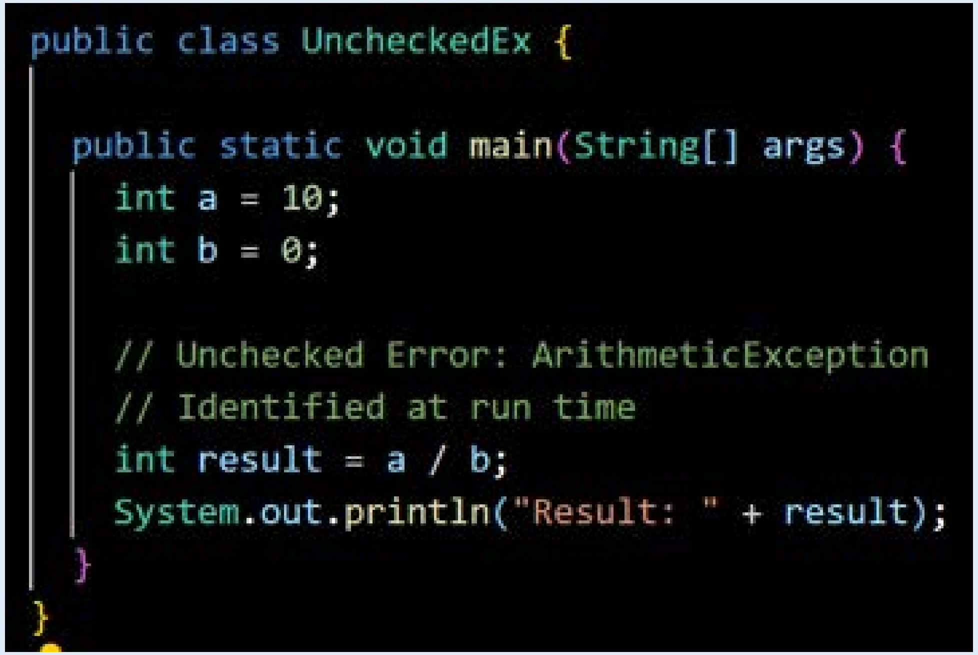
# Q. What are checked and unchecked exceptions? What is the difference between them?

- Checked exceptions are exceptions that are checked by the compiler at compile-time.
- Checked exceptions will not compile without handling them either using try-catch blocks or throws statement.



```
public class CheckedEx {

  public static void main(String[] args) {
    FileInputStream fis = null;

    // Checked Error: FileNotFoundException
    // Identified at compile time
    fis = new FileInputStream("nonexistentfile.txt");
    int data = fis.read(); // Read data from file
    System.out.println(data);
    fis.close(); // Close file stream
  }
}
```

- Unchecked exceptions, not checked by the compiler at compile-time.
- Unchecked exception can compile without try-catch or throws statement.



```
public class UncheckedEx {

  public static void main(String[] args) {
    int a = 10;
    int b = 0;

    // Unchecked Error: ArithmeticException
    // Identified at run time
    int result = a / b;
    System.out.println("Result: " + result);
  }
}
```

back to chapter index

# 17. Collections - Basics

Q. What are collections and what is their use in Java? **V. IMP.**

Q. What are the types of collections in Java? **V. IMP.**

Q. What are Iterable & Collection interfaces?

Q. What are List, Queue & Set collections? What is the difference between them? **V. IMP.**

Q. What is Arraylist? How to implement it and when to use it?

Q. What are the differences between Array and Arraylist? **V. IMP.**

Q. What is HashSet? What are the differences btw ArrayList(List) & HashSet(Set)? **V. IMP.**

Q. What is Map In Java? Which classes implements Map interface?

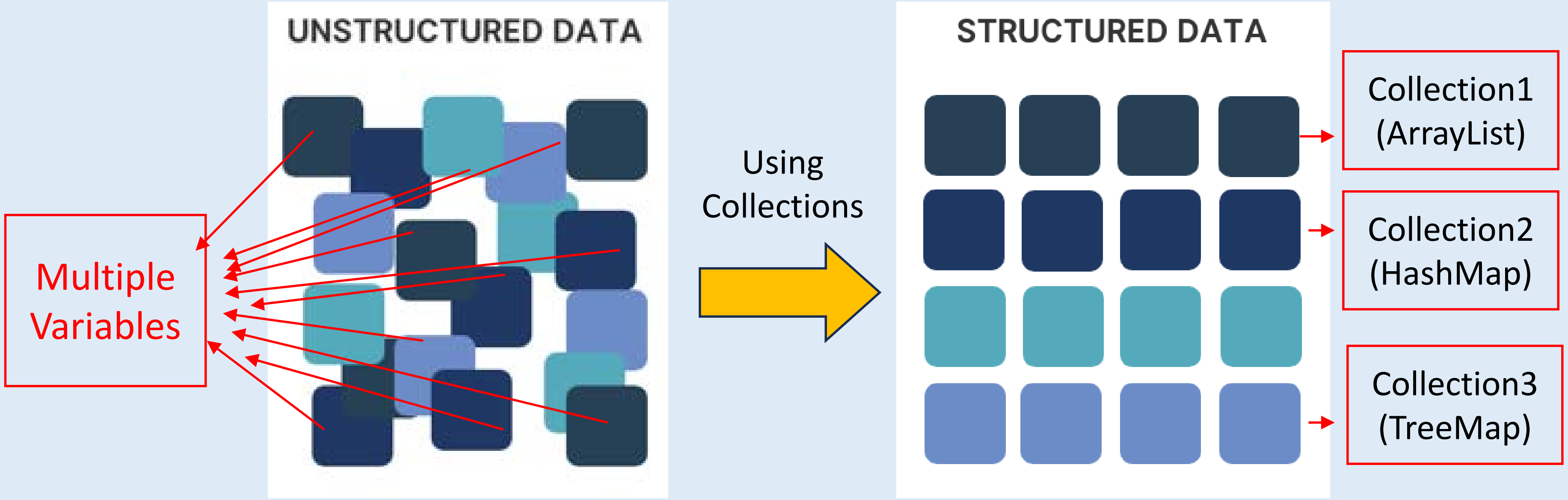
Q. What is HashMap In Java? How to implement it and when to use it?

Q. What are the differences btw HashSet(Set) and HashMap(Map)?

Back to main index

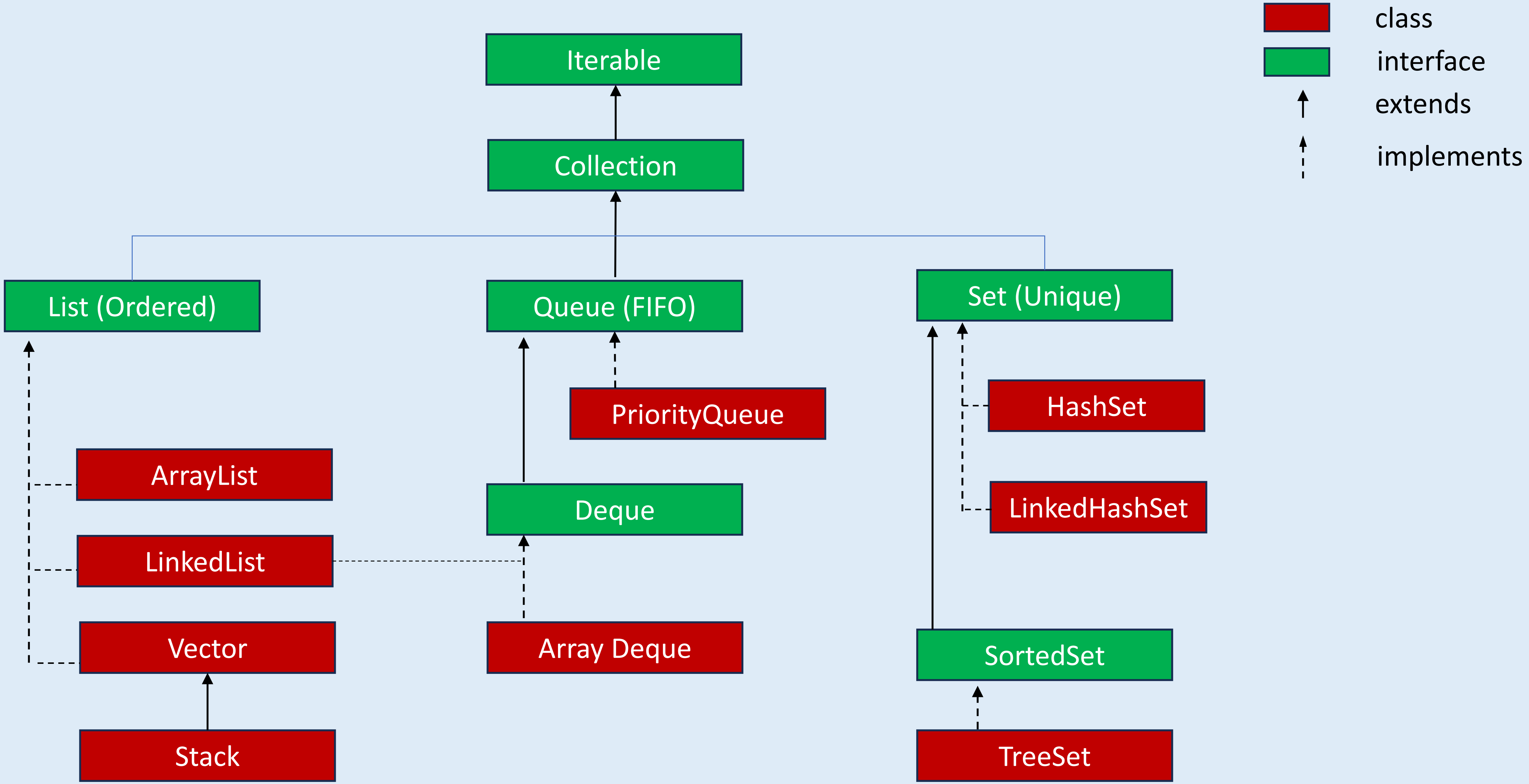
# Q. What are collections and what is their use in Java? V. IMP.

- Collections refer to a group of classes and interfaces in the java.util package that are designed to **store, manipulate, and manage groups of objects**.



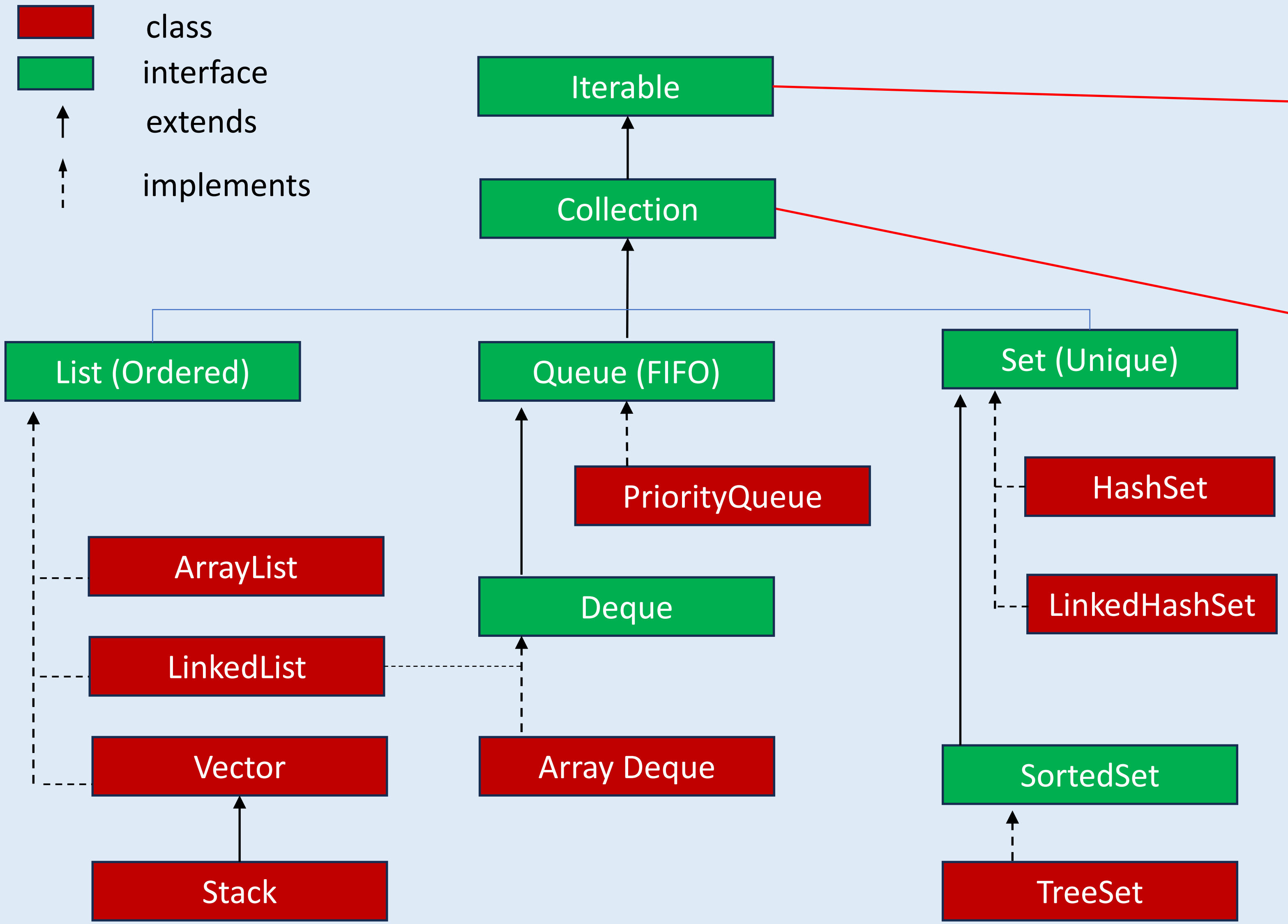
back to chapter index

# Q. What are the types of collections in Java? V. IMP.



back to chapter index

# Q. What are Iterable & Collection interfaces?

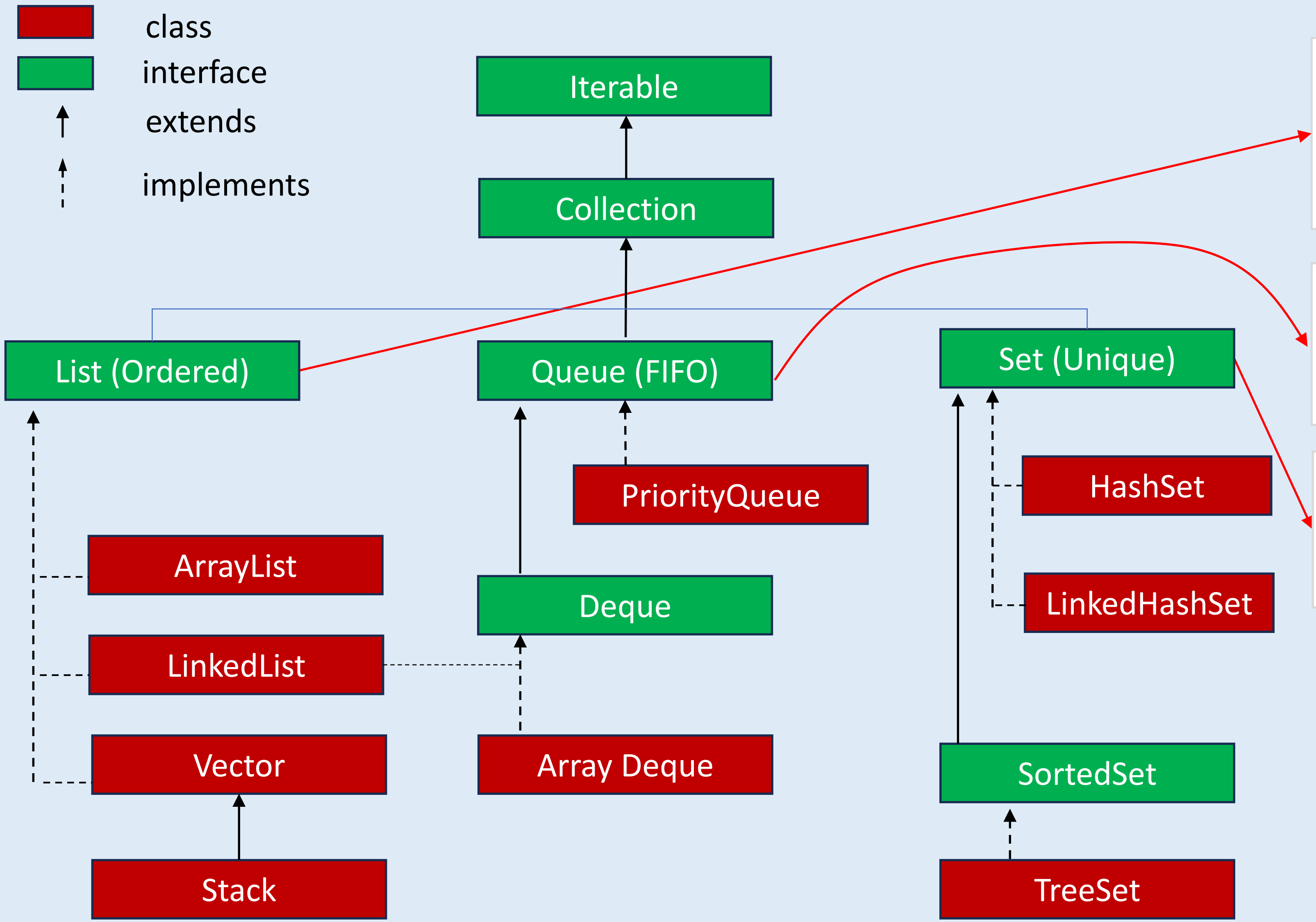


- Iterable is an interface that provides a way to iterate over a collection of elements.

- The Collection interface provide common operations like **adding, removing, and accessing elements** of collection.

back to chapter index

# Q. What are List, Queue & Set collections? What is the difference between them? V. IMP.



🟥 class

🟩 interface

↑ extends

⇡ implements

Iterable

Collection

List (Ordered)

Queue (FIFO)

Set (Unique)

PriorityQueue

ArrayList

LinkedList

Vector

Stack

Deque

Array Deque

HashSet

LinkedHashSet

SortedSet

TreeSet

- **Ordered** collection allowing **duplicate** elements and access collection based on the **indices** (get(index) method).

- Use Queue when you need to process elements in a specific order (**FIFO order**) (e.g., task scheduling, event handling).

- **Unordered** collection that does not allow duplicates.

back to chapter index

# Q. What is Arraylist? How to implement it and when to use it?

❖ ArrayList is a class that implements the List interface and uses a dynamic array to store elements.

❖ Use ArrayList when:

1. You need a resizable list of elements.
2. You want to access elements by their index.
3. You frequently add/ remove elements from the list.

```java
// ArrayList Example
import java.util.ArrayList;

public class ArrayListEx {
  public static void main(String[] args) {
    // Creating an ArrayList of integers
    ArrayList<Integer> nums = new ArrayList<>();

    // Adding elements to ArrayList dynamically
    nums.add(10);
    nums.add(20);
    nums.add(30);

    // Accessing and printing ArrayList elements
    System.out.println(nums.get(2)); // Output: 30

    // Removing an element from the ArrayList
    nums.remove(2); // Remove element at index 3

    // Printing ArrayList
    System.out.println(nums); // Output: [10, 20]
  }
}
```

back to chapter index

# Q. What are the differences between Array and Arraylist? V. IMP.

```java
// Array Example
public class ArrayEx {
  public static void main(String[] args) {
    // Declaring and initializing
    // an array of integers of fix size 5
    int[] numbers = new int[5];

    // Assigning values to array elements
    numbers[0] = 10;
    numbers[1] = 20;
    numbers[2] = 30;

    // Declaring and initializing an
    // array in single line
    // int[] numbers = {10, 20, 30};

    // Accessing and printing array elements
    System.out.println(numbers[2]);
    // Output: 30
  }
}
```

```java
// ArrayList Example
import java.util.ArrayList;

public class ArrayListEx {
  public static void main(String[] args) {
    // Creating an ArrayList of integers
    ArrayList<Integer> nums = new ArrayList<>();

    // Adding elements to ArrayList dynamically
    nums.add(10);
    nums.add(20);
    nums.add(30);

    // Accessing and printing ArrayList elements
    System.out.println(nums.get(2)); // Output: 30

    // Removing an element from the ArrayList
    nums.remove(2); // Remove element at index 3
    // Printing ArrayList
    System.out.println(nums); // Output: [10, 20]
  }
}
```

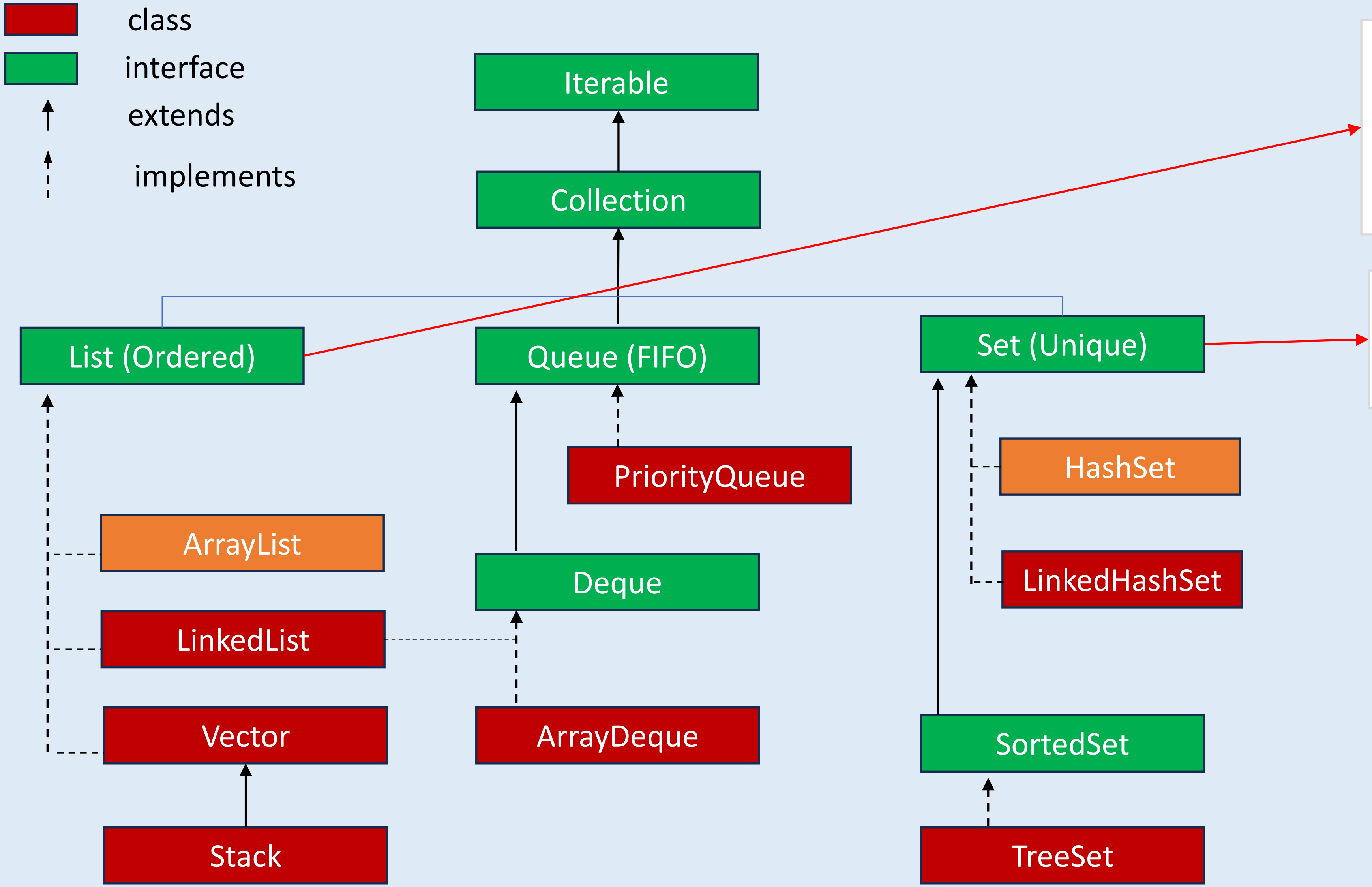
back to chapter index

# Q. What are the differences between Array and Arraylist? V. IMP.

| Array | ArrayList |
|---|---|
| 1. Fixed size (determined at creation) | Dynamic size (can grow or shrink during runtime) |
| 2. Cannot resize once created | Automatically resizes when elements are added or removed |
| 3. Not part of Java Collections Framework | Part of Java Collections Framework (java.util.ArrayList) |
| 4. Declared using square brackets ([]) | Declared and initialized using ArrayList class |
| 5. No built-in methods for adding or removing elements | Provides built-in methods (add(), remove(), get(), size(), etc.) for dynamic operations. |

back to chapter index

# Q. What is HashSet? What are the differences btw ArrayList(List) & HashSet(Set)? V. IMP.



🟥 class

🟩 interface

↑ extends

⇡ implements

- **Ordered** collection allowing **duplicate** elements and access collection based on the **indices** (get(index) method).

- **Unordered** collection that does not allow duplicates.

back to chapter index

# Q. What is HashSet? What are the differences btw ArrayList(List) & HashSet(Set)? V. IMP.

```java
// ArrayList Example
import java.util.ArrayList;

public class ArrayListEx {
  public static void main(String[] args) {
    // Creating an ArrayList of integers
    ArrayList<Integer> nums = new ArrayList<>();

    // Adding elements to ArrayList dynamically
    nums.add(10);
    nums.add(20);
    nums.add(30);

    // Accessing and printing ArrayList elements
    System.out.println(nums.get(2)); // Output: 30

    // Removing an element from the ArrayList
    nums.remove(2); // Remove element at index 3
    // Printing ArrayList
    System.out.println(nums); // Output: [10, 20]
  }
}
```

```java
// HashSet Example
import java.util.HashSet;

public class HashSetEx {
  public static void main(String[] args) {
    // Creating a HashSet of integers
    HashSet<Integer> numsSet = new HashSet<>();

    // Adding elements to the HashSet
    numsSet.add(10);
    numsSet.add(20);
    numsSet.add(30);
    numsSet.add(20); // duplicate not allowed

    System.out.println(numsSet); // [20, 10, 30]

    // Access a specific element not supported

    numsSet.remove(30); // Removing an element

    System.out.println(numsSet); // [20, 10]
  }
}
```

back to chapter index

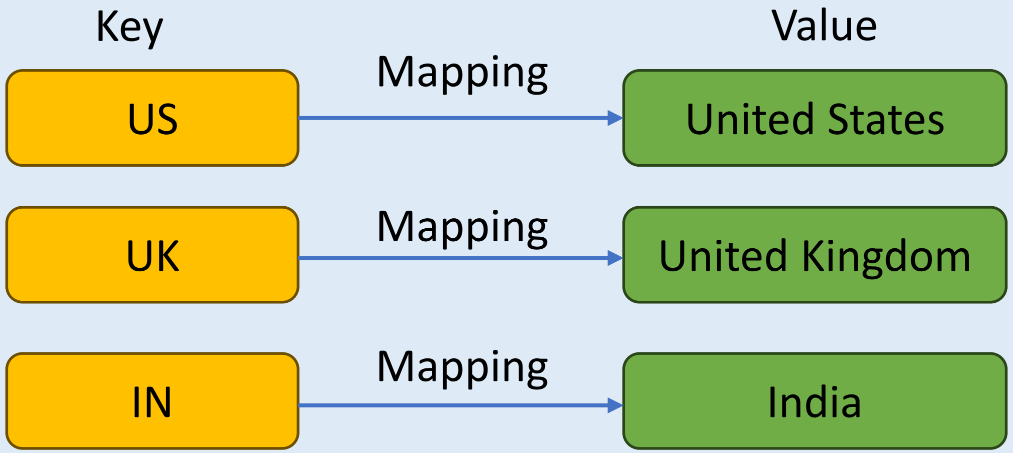
# Q. What is HashSet? What are the differences btw ArrayList(List) & HashSet(Set)? V. IMP.

| ArrayList | HashSet |
|---|---|
| 1. Implements List interface | Implements Set interface |
| 2. Maintains insertion order | Does not maintain insertion order |
| 3. Allows duplicate elements | Does not allow duplicate elements (stores unique) |
| 4. Supports indexed access (get(index)) | Does not support direct indexed access |
| 5. Ordered iteration based on insertion sequence | Unordered iteration; iteration order not guaranteed |

back to chapter index

# Q. What is Map In Java? Which classes implements Map interface? V. IMP.

- Map is an interface that represents a collection of **key-value pairs** where each key is unique.



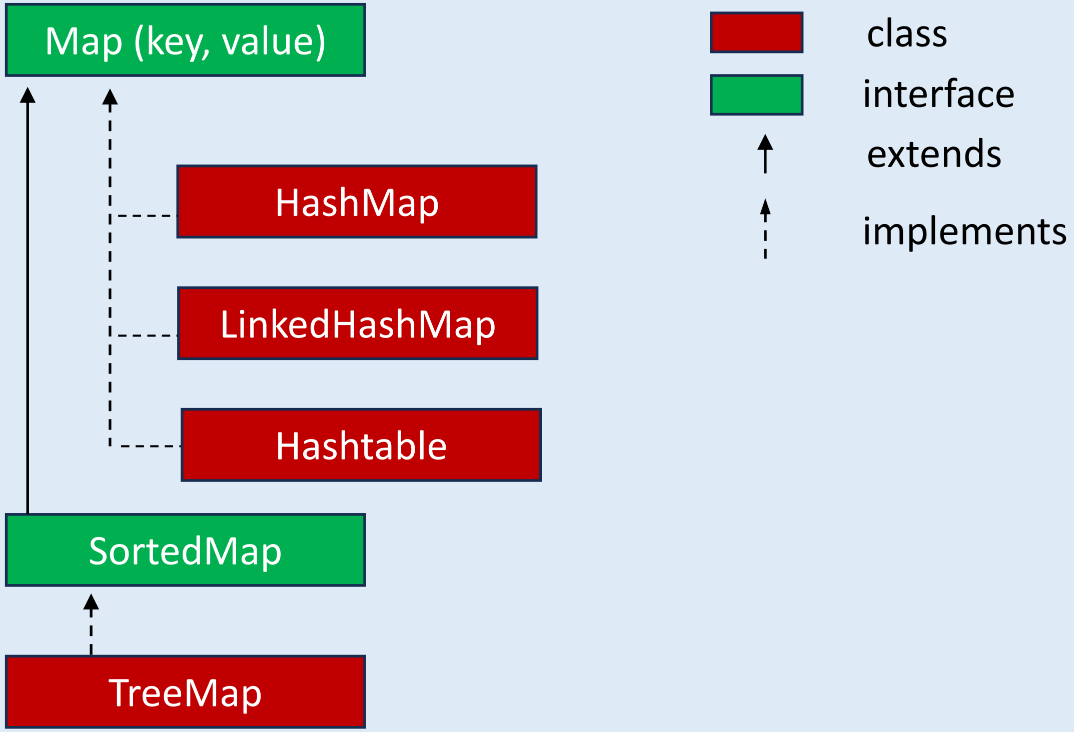
Key

Value

US → Mapping → United States

UK → Mapping → United Kingdom

IN → Mapping → India



Map (key, value)

HashMap

LinkedHashMap

Hashtable

SortedMap

TreeMap

🟥 class

🟩 interface

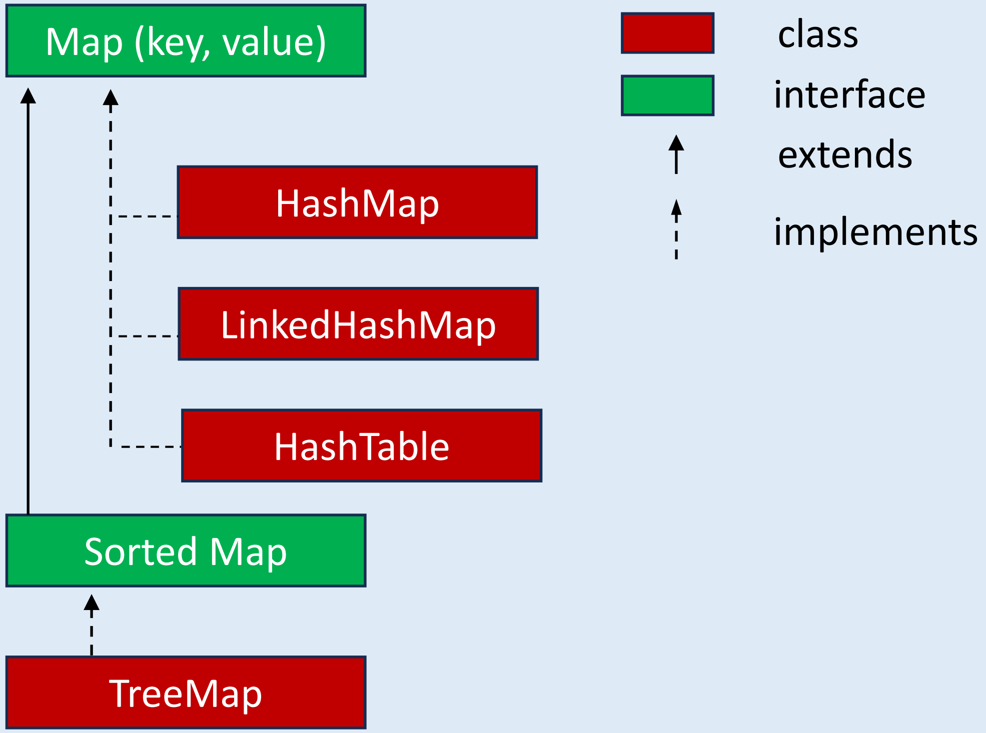
↑ extends

⇡ implements

back to chapter index

# Q. What is HashMap In Java? How to implement it and when to use it?

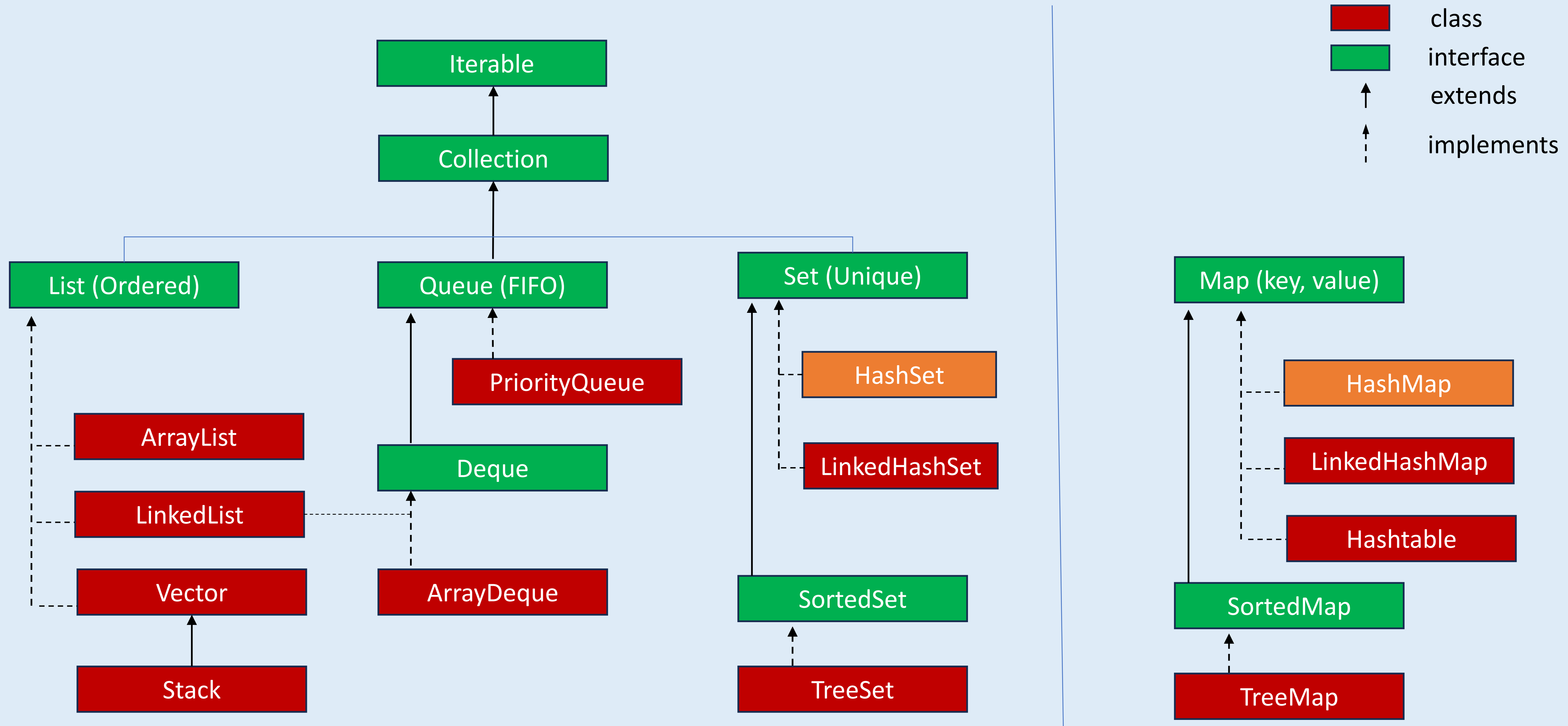
- HashMap is a class that implements the Map interface and represents a collection of **key-value pairs**.



```
// HashMap Example
import java.util.HashMap;

public class HashMapEx {

  public static void main(String[] args) {

    // Creating a HashMap to store key-value pairs
    HashMap<String, Integer> marks = new HashMap<>();

    // Adding key-value pairs to the HashMap
    marks.put("Happy", 33);
    marks.put("Anurag", 34);
    marks.put("Rawat", 35);

    // Accessing and printing HashMap elements
    // (value associated with key "Anurag")
    System.out.println(marks.get("Anurag"));
    // Output: 34

    // Removing an element from the HashMap
    marks.remove("Rawat");
  }
}
```

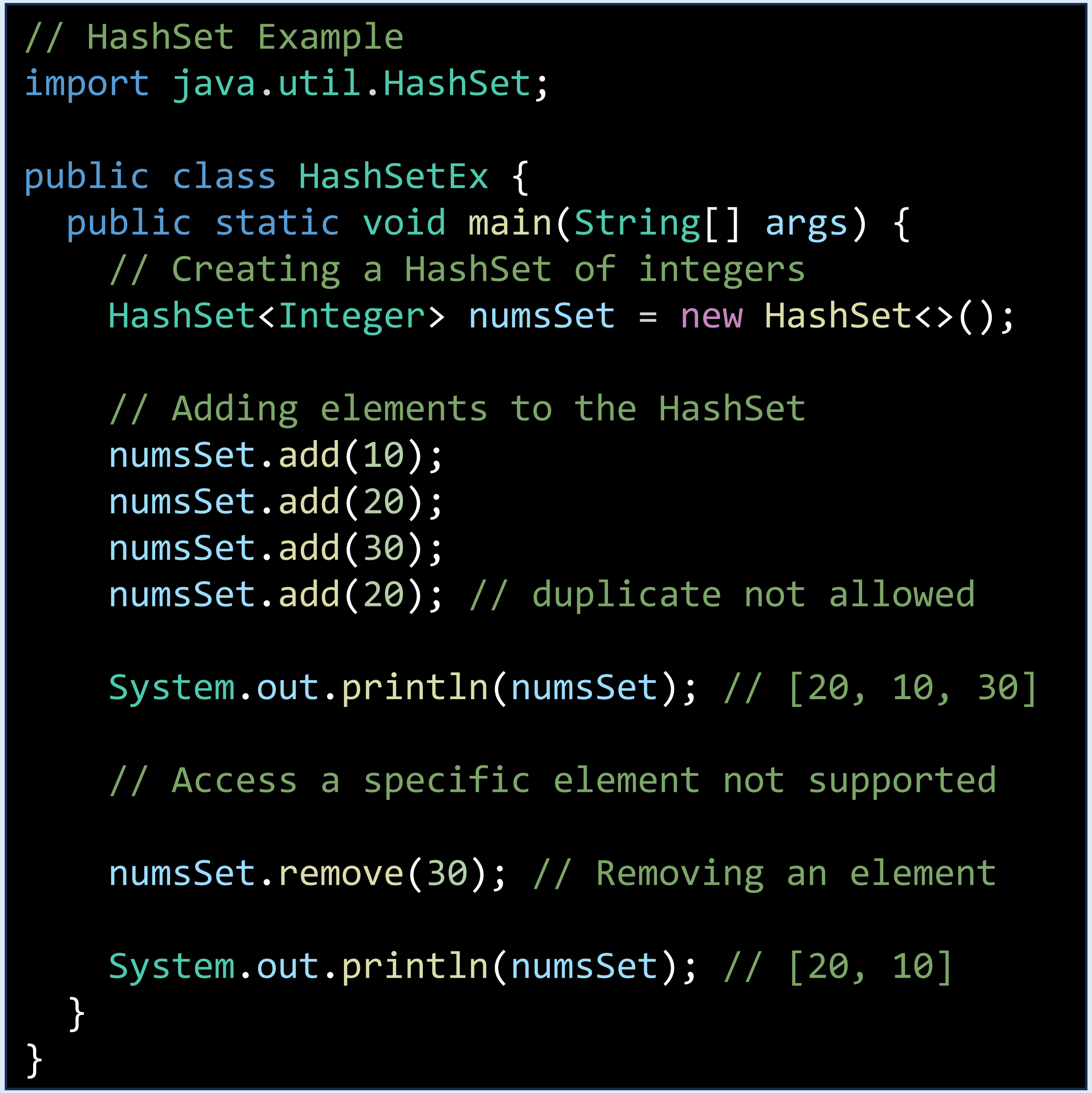
back to chapter index

# Q. What are the differences btw HashSet(Set) and HashMap(Map)?

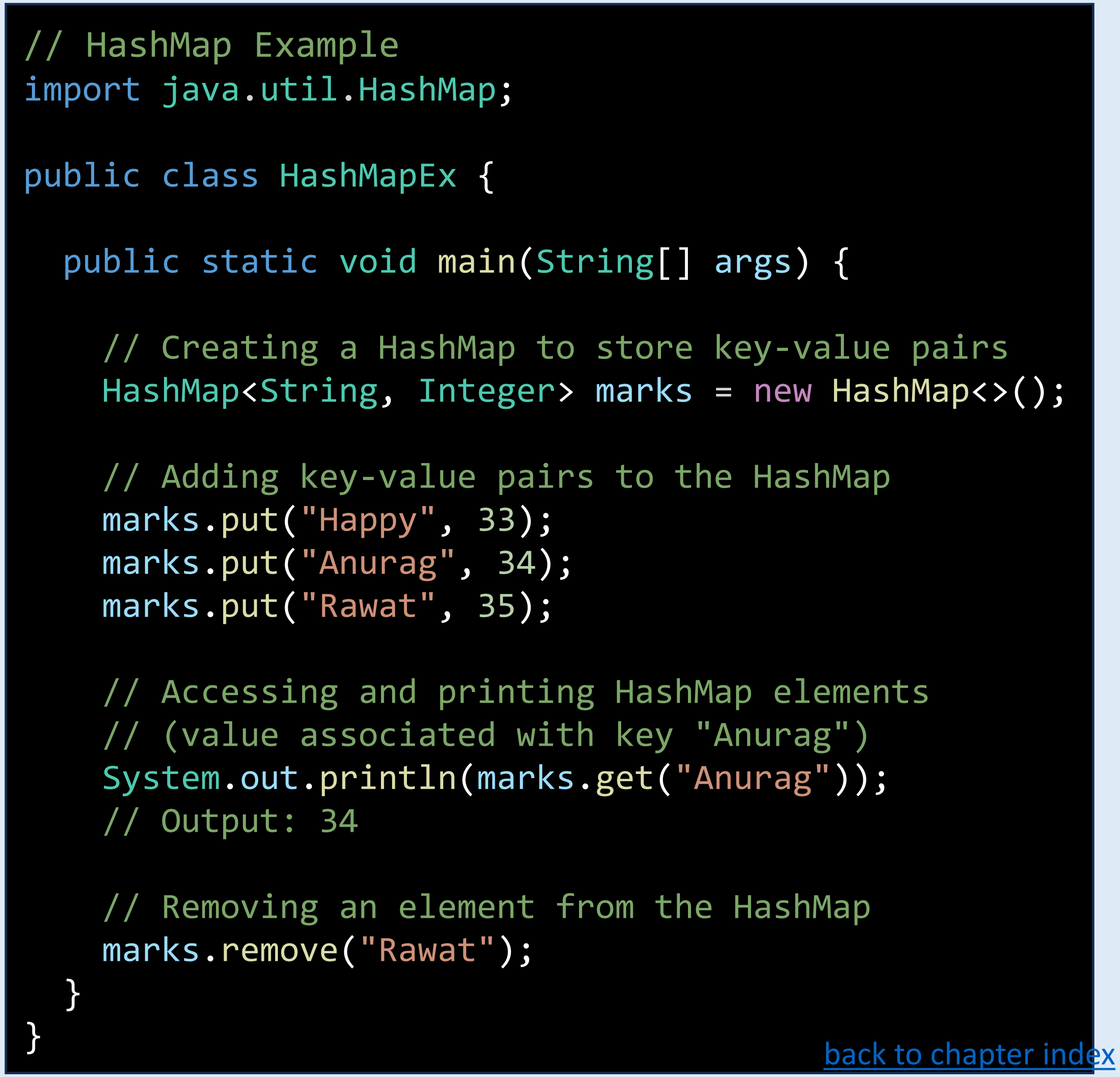


back to chapter index

# Q. What are the differences btw HashSet(Set) and HashMap(Map)?



```java
// HashSet Example
import java.util.HashSet;

public class HashSetEx {
  public static void main(String[] args) {
    // Creating a HashSet of integers
    HashSet<Integer> numsSet = new HashSet<>();

    // Adding elements to the HashSet
    numsSet.add(10);
    numsSet.add(20);
    numsSet.add(30);
    numsSet.add(20); // duplicate not allowed

    System.out.println(numsSet); // [20, 10, 30]

    // Access a specific element not supported

    numsSet.remove(30); // Removing an element

    System.out.println(numsSet); // [20, 10]
  }
}
```



```java
// HashMap Example
import java.util.HashMap;

public class HashMapEx {

  public static void main(String[] args) {

    // Creating a HashMap to store key-value pairs
    HashMap<String, Integer> marks = new HashMap<>();

    // Adding key-value pairs to the HashMap
    marks.put("Happy", 33);
    marks.put("Anurag", 34);
    marks.put("Rawat", 35);

    // Accessing and printing HashMap elements
    // (value associated with key "Anurag")
    System.out.println(marks.get("Anurag"));
    // Output: 34

    // Removing an element from the HashMap
    marks.remove("Rawat");
  }
}
```

back to chapter index

# Q. What are the differences btw HashSet(Set) and HashMap(Map)?

<table>
<tr><th>HashSet</th><th>HashMap</th></tr>
<tr><td colspan="2">Does not allow duplicates</td></tr>
<tr><td colspan="2">Does not maintain insertion order</td></tr>
<tr><td colspan="2">Uses hashing mechanism to store elements</td></tr>
<tr><td>1. HashSet is used to store a collection of individual elements.</td><td>HashMap is used to store key-value pairs.</td></tr>
<tr><td>2. HashSet Implements Set interface</td><td>HashMap Implements Map interface</td></tr>
</table>

back to chapter index

# 18. Collections - Advanced

Q. What is LinkedList in Java? What are Singly and Doubly linked list? **V. IMP.**

Q. How to implement LinkedList? Difference between Arraylist & LinkedList?

Q. What is the difference between Arraylist & LinkedList? When to use what?

Q. What are Collections? What is the difference between Collection and Collections? **V. IMP.**

Q. What is TreeSet in Java? What is the difference between HashSet and TreeSet?

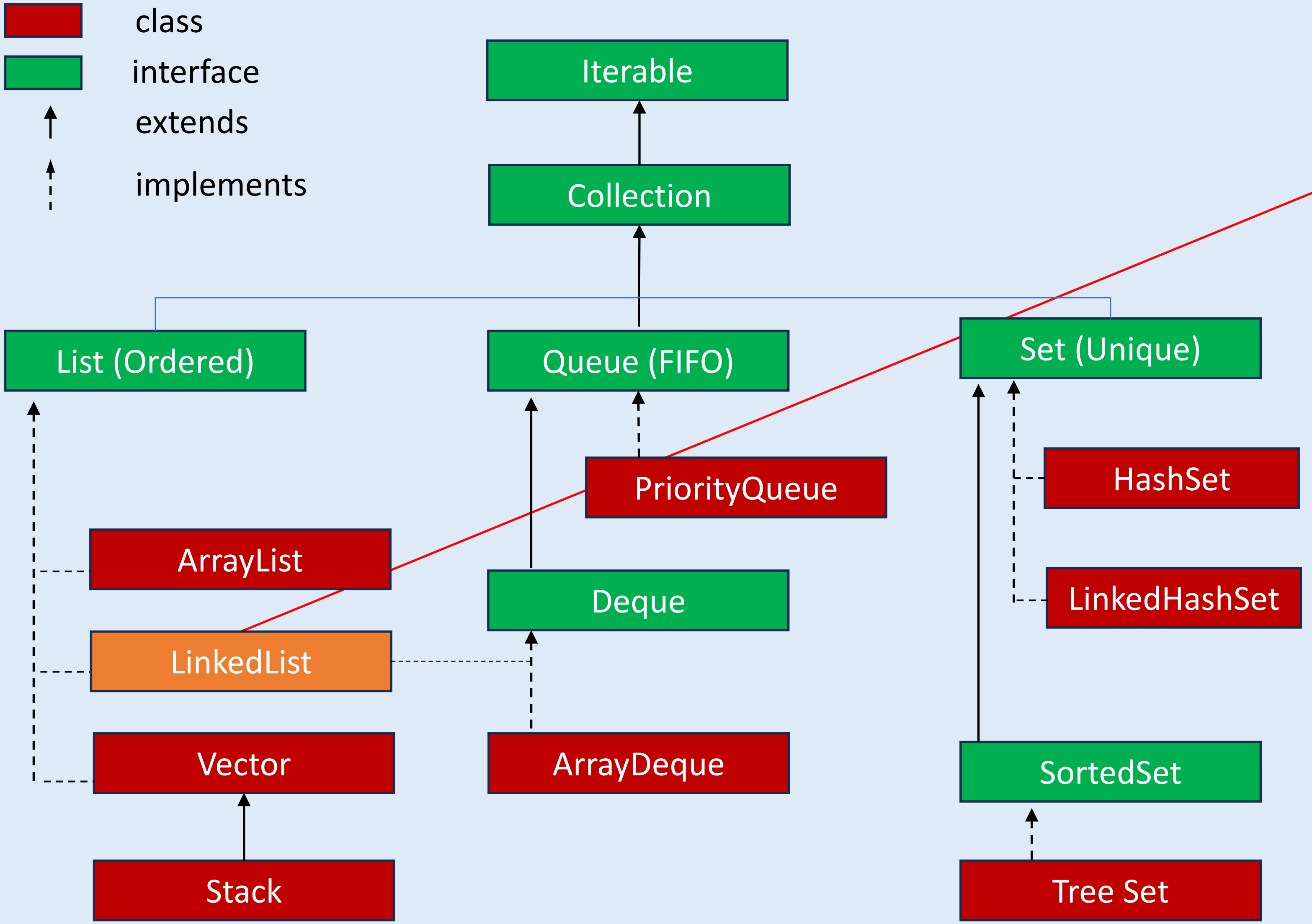
Q. What is the difference between HashSet and TreeSet?

Q. What is the difference between HashMap and HashTable? **V. IMP.**

Q. What are the advantages of using collections?

Back to main index

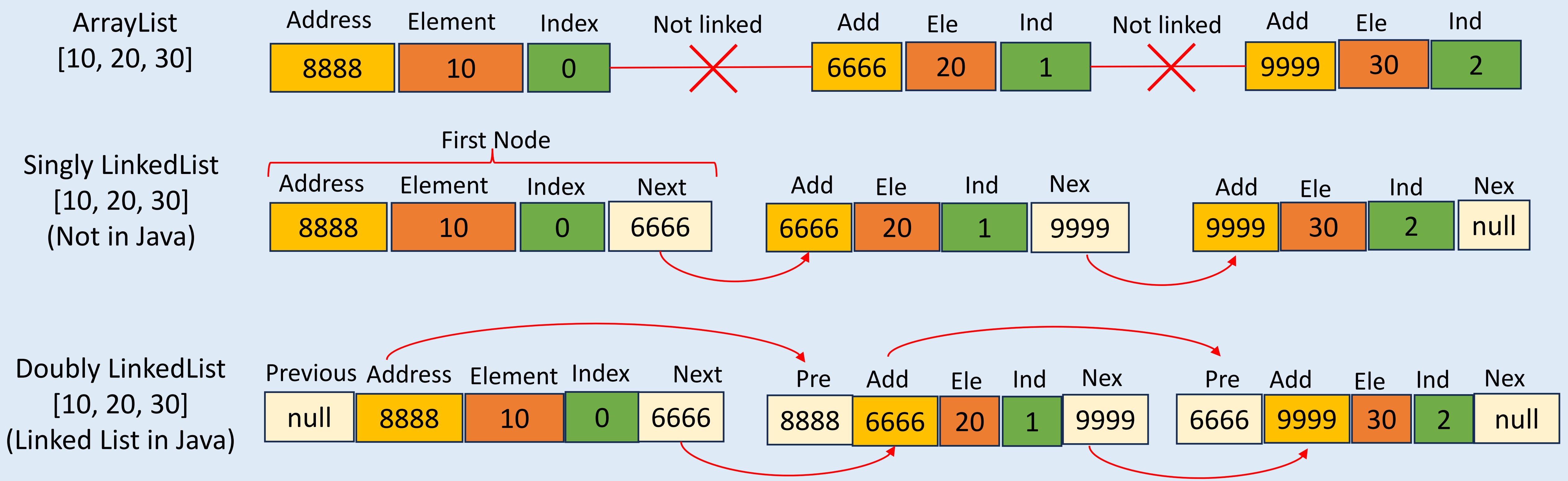
# Q. What is LinkedList in Java? What are Singly and Doubly linked list? V. IMP.



- LinkedList in Java is a doubly linked list implementation of the List interface. It consists of nodes where each node contains a reference to the next and previous node.

back to chapter index

# Q. What is LinkedList in Java? What are Singly and Doubly linked list? V. IMP.



- An ArrayList is a contiguous block of memory that directly contains the data elements.
- A singly linked list consists of list of nodes, each containing a data element and a reference to the next node.
- A doubly linked list consists of list of nodes, each containing a data element & a reference to the next & previous node.

back to chapter index

# Q. How to implement LinkedList? Difference between Arraylist & LinkedList?

```java
// ArrayList Example
import java.util.ArrayList;

public class ArrayListEx {

  public static void main(String[] args) {
    // Creating an ArrayList of integers
    ArrayList<Integer> nums = new ArrayList<>();

    // Adding elements to ArrayList dynamically
    nums.add(10);
    nums.add(20);
    nums.add(30);

    // Accessing and printing ArrayList elements
    System.out.println(nums.get(2)); // 30

    // Removing an element from the ArrayList
    nums.remove(2); // Remove element at index 2
  }
}
```

```java
// LinkedList Example
import java.util.LinkedList;

public class LinkedListEx {

  public static void main(String[] args) {
    // Creating a LinkedList of integers
    LinkedList<Integer> nums = new LinkedList<>();

    // Adding elements to LinkedList dynamically
    nums.add(10);
    nums.add(20);
    nums.add(30);

    // Accessing and printing LinkedList elements
    System.out.println(nums.get(2)); // 30

    // Removing an element from the LinkedList
    nums.remove(2); // Remove element at index 2
  }
}
```

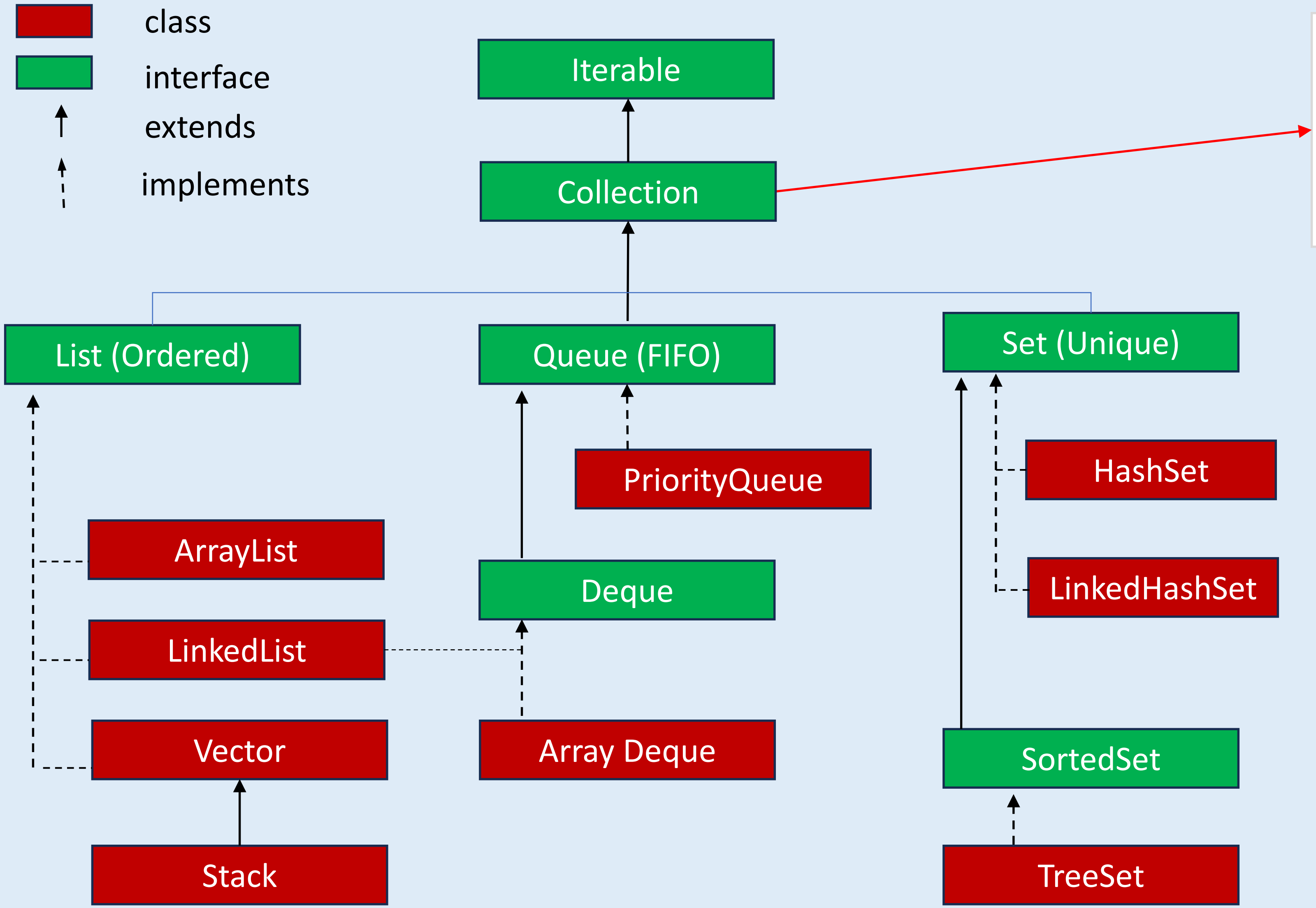
back to chapter index

# Q. What is the difference between Arraylist & LinkedList? When to use what?

| ArrayList | LinkedList |
|---|---|
| 1. Resizable array | Doubly linked list with previous and next node references. |
| 2. Efficient using index-based retrieval 🏆 | Inefficient for random access |
| 3. Slower for insertions/removals in the middle (shifts elements) | Faster (O(1)) for insertions/removals in the middle (adjusts links) 🏆 |
| 4. More memory efficient for storing large lists. 🏆 | Less memory efficient due to node overhead |
| 5. Faster iteration due to contiguous memory layout 🏆 | Slower iteration due to node traversal |

back to chapter index

# Q. What are Collections? What is the difference between Collection and Collections?



- The Collection interface provide common operations like **adding, removing, and accessing elements** of collection.

back to chapter index

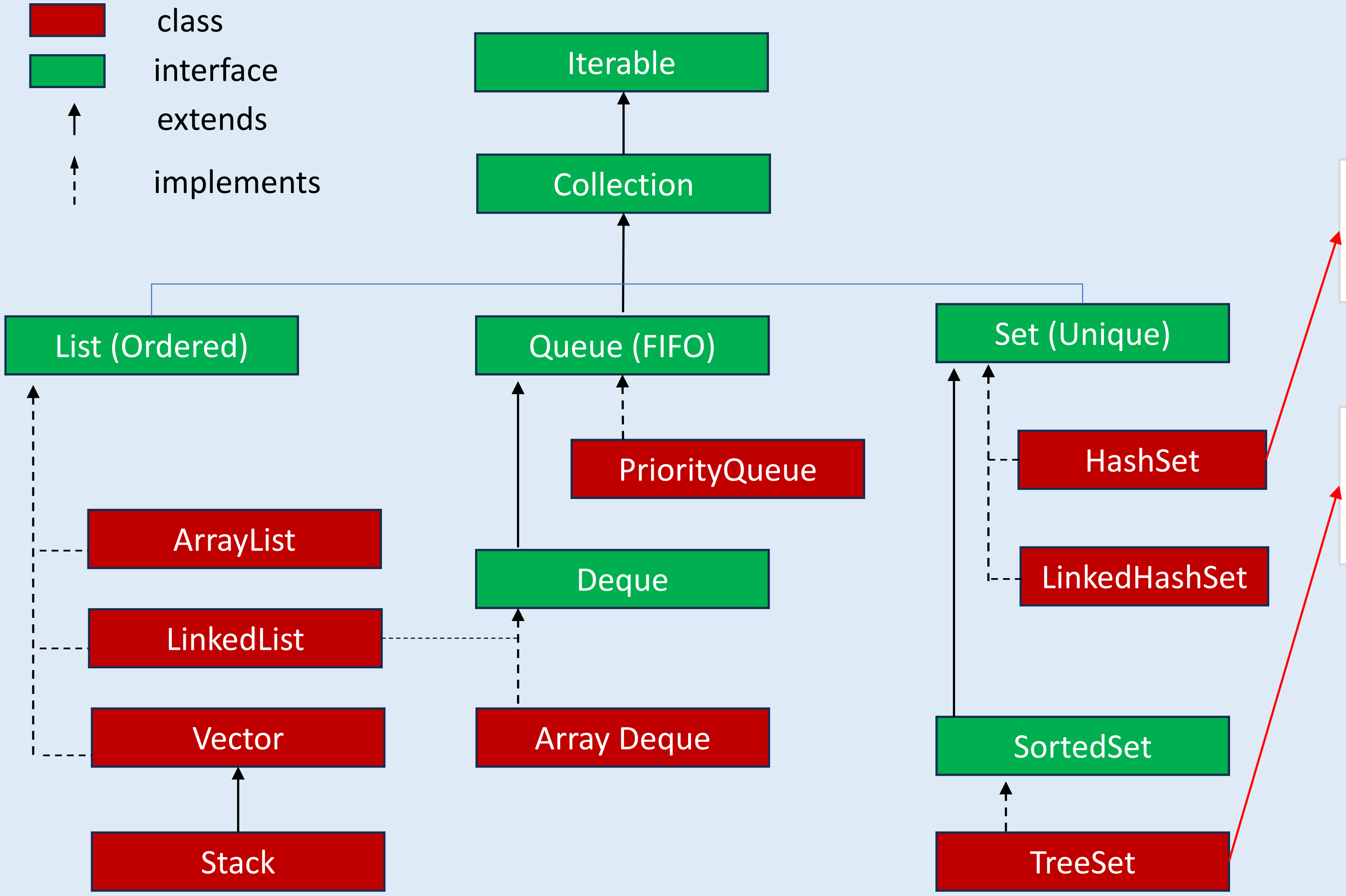
# Q. What are Collections? What is the difference between Collection and Collections?

- Collections is a utility class in Java that contains static methods for working with collections. For example, Collections.sort(objectArrayList)

```java
// Collections Example
import java.util.ArrayList;
import java.util.Collections;

public class CollectionsEx {
  public static void main(String[] args) {
    ArrayList<Integer> numbers = new ArrayList<>();
    numbers.add(5);
    numbers.add(2);
    numbers.add(8);

    // Sorts the list
    Collections.sort(numbers);
    System.out.println(numbers); // Output: [2, 5, 8]

    // Reversing the order of elements
    Collections.reverse(numbers);
    System.out.println(numbers); // Output: [8, 5, 2]

    // Search for an element in the list
    int index = Collections.binarySearch(numbers, 5);
    System.out.println(index); // 1
  }
}
```

back to chapter index

# Q. What is TreeSet in Java? What is the difference between HashSet and TreeSet?

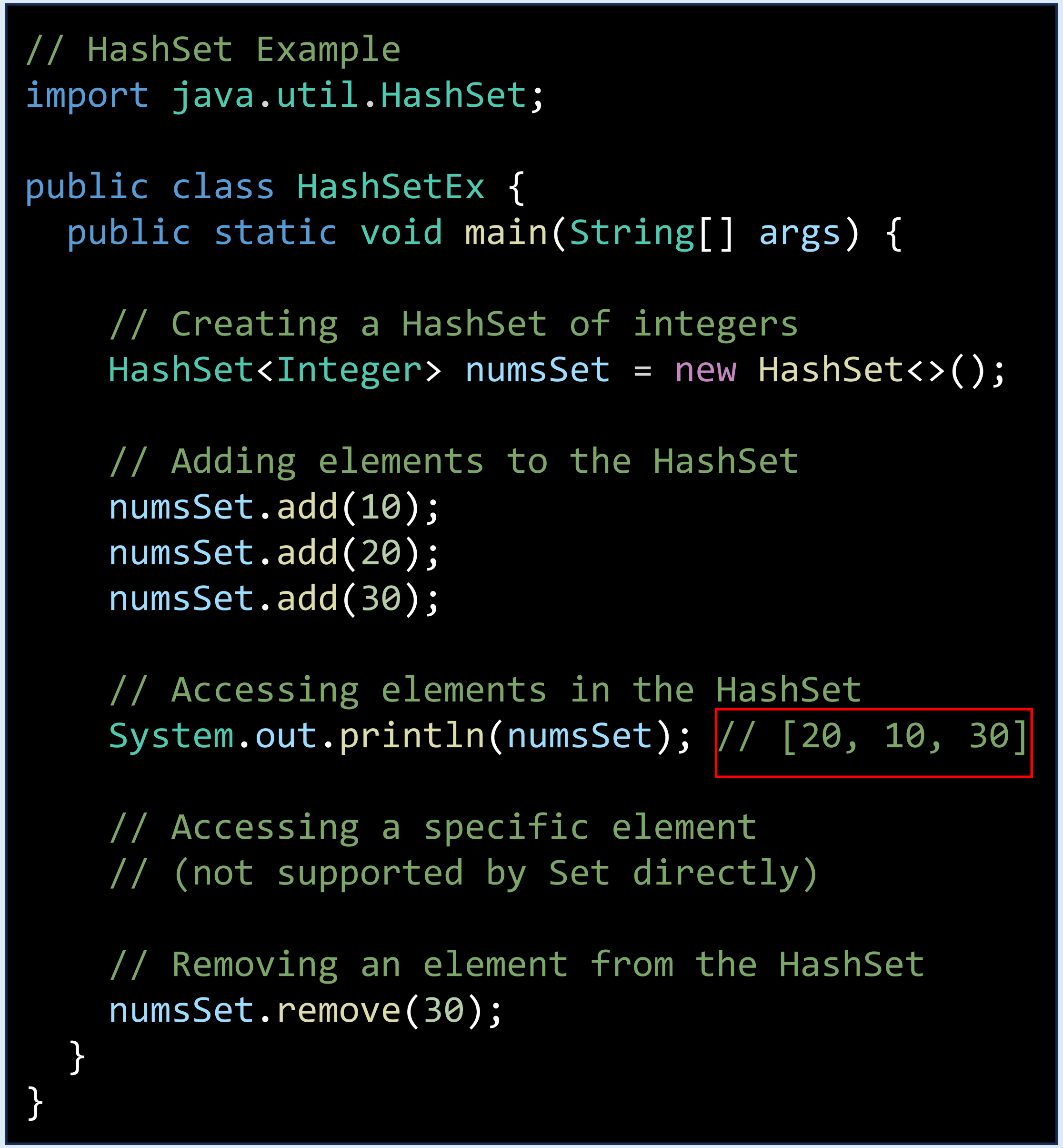


- HashSet is a collection that stores **unique elements**.

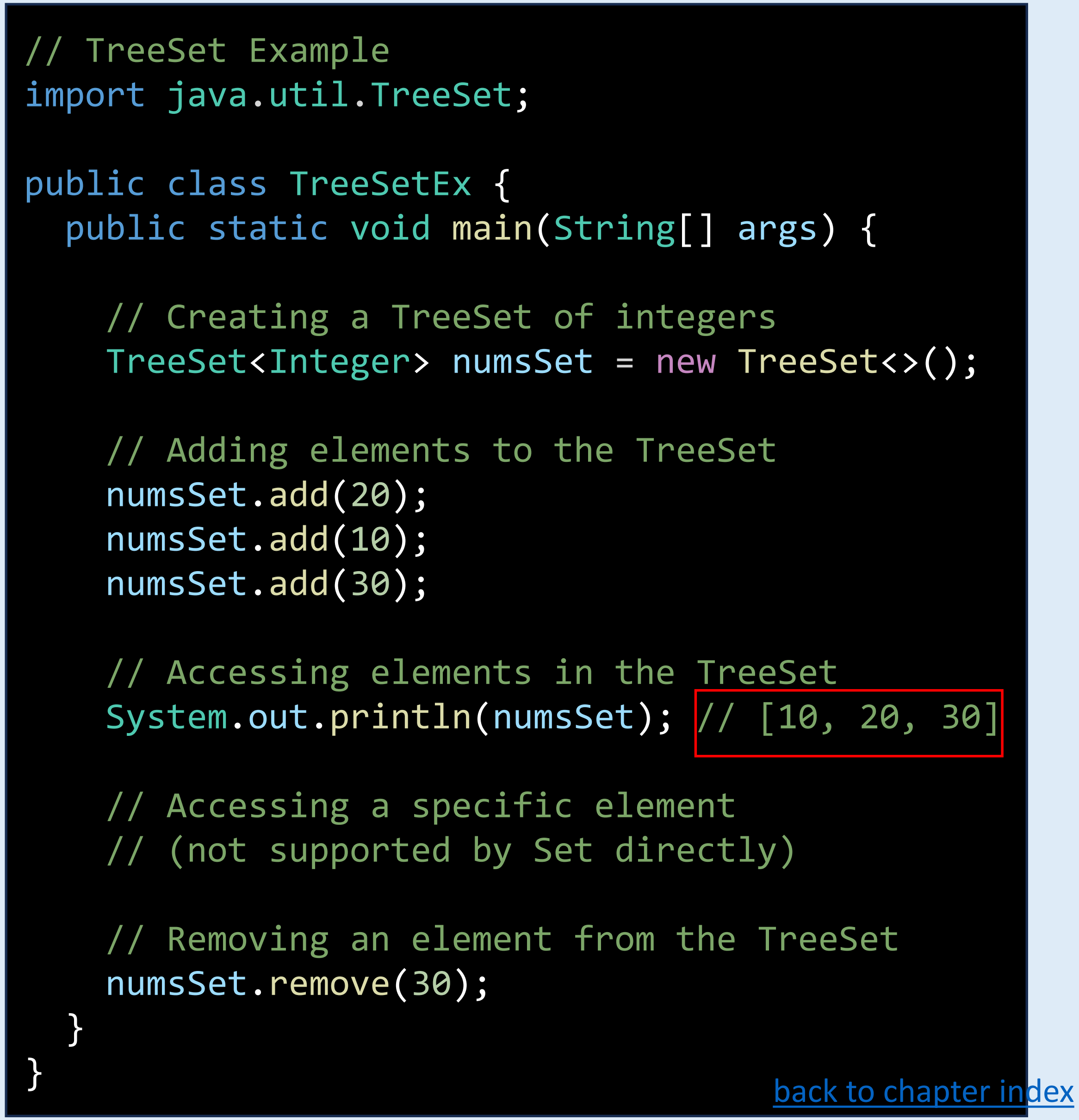
- TreeSet is a collection that stores **unique elements in sorted order**.

back to chapter index

# Q. What is TreeSet in Java? What is the difference between HashSet and TreeSet?



```java
// HashSet Example
import java.util.HashSet;

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

    // Creating a HashSet of integers
    HashSet<Integer> numsSet = new HashSet<>();

    // Adding elements to the HashSet
    numsSet.add(10);
    numsSet.add(20);
    numsSet.add(30);

    // Accessing elements in the HashSet
    System.out.println(numsSet); // [20, 10, 30]

    // Accessing a specific element
    // (not supported by Set directly)

    // Removing an element from the HashSet
    numsSet.remove(30);
  }
}
```



```java
// TreeSet Example
import java.util.TreeSet;

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

    // Creating a TreeSet of integers
    TreeSet<Integer> numsSet = new TreeSet<>();

    // Adding elements to the TreeSet
    numsSet.add(20);
    numsSet.add(10);
    numsSet.add(30);

    // Accessing elements in the TreeSet
    System.out.println(numsSet); // [10, 20, 30]

    // Accessing a specific element
    // (not supported by Set directly)

    // Removing an element from the TreeSet
    numsSet.remove(30);
  }
}
```

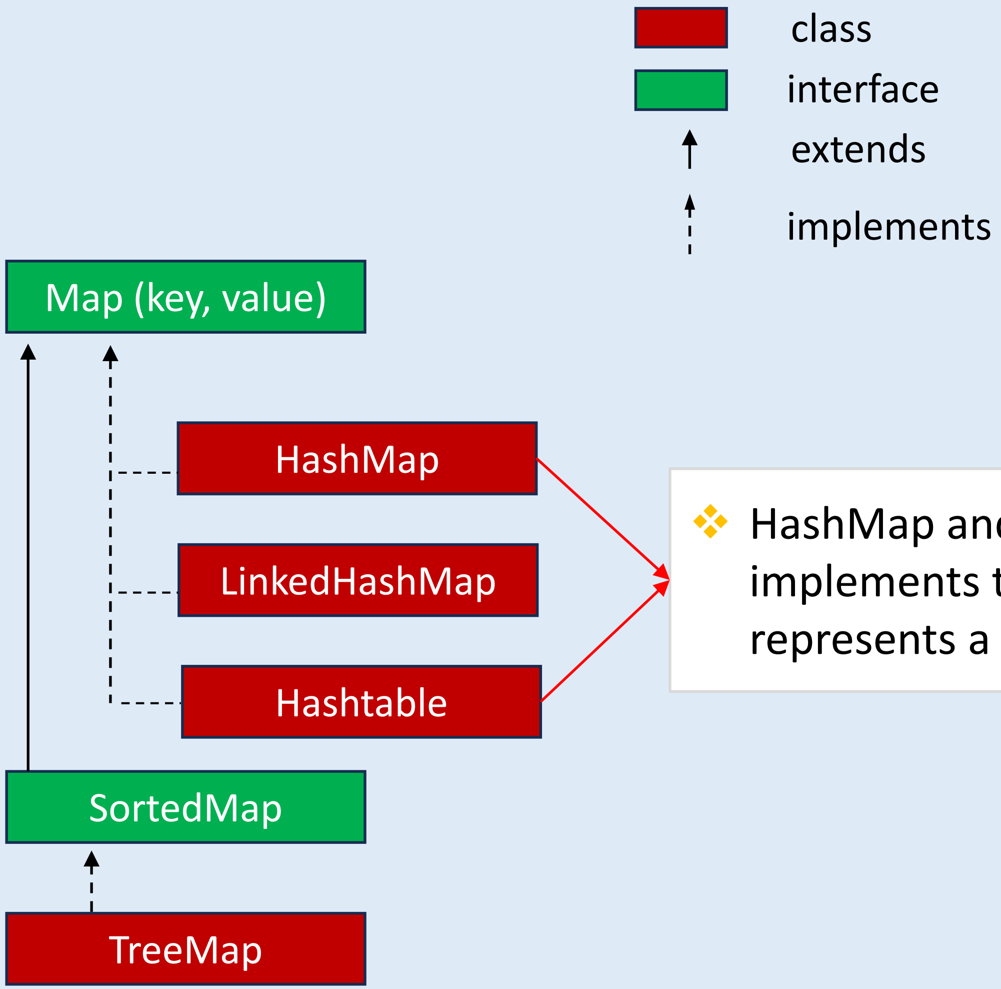
back to chapter index

# Q. What is the difference between HashSet and TreeSet?

| HashSet | TreeSet |
|---|---|
| Does not allow duplicate elements | |
| No direct support for accessing by index | |
| 1. Does not maintain any specific order | Maintains elements in sorted (natural) order |
| 2. Unpredictable order (based on hash codes) | Iterates in sorted (ascending) order |
| 3. Used when uniqueness of elements are required | Used when uniqueness of elements in sorted order is required |
| 4. Generally lower memory overhead | Potentially higher memory overhead |

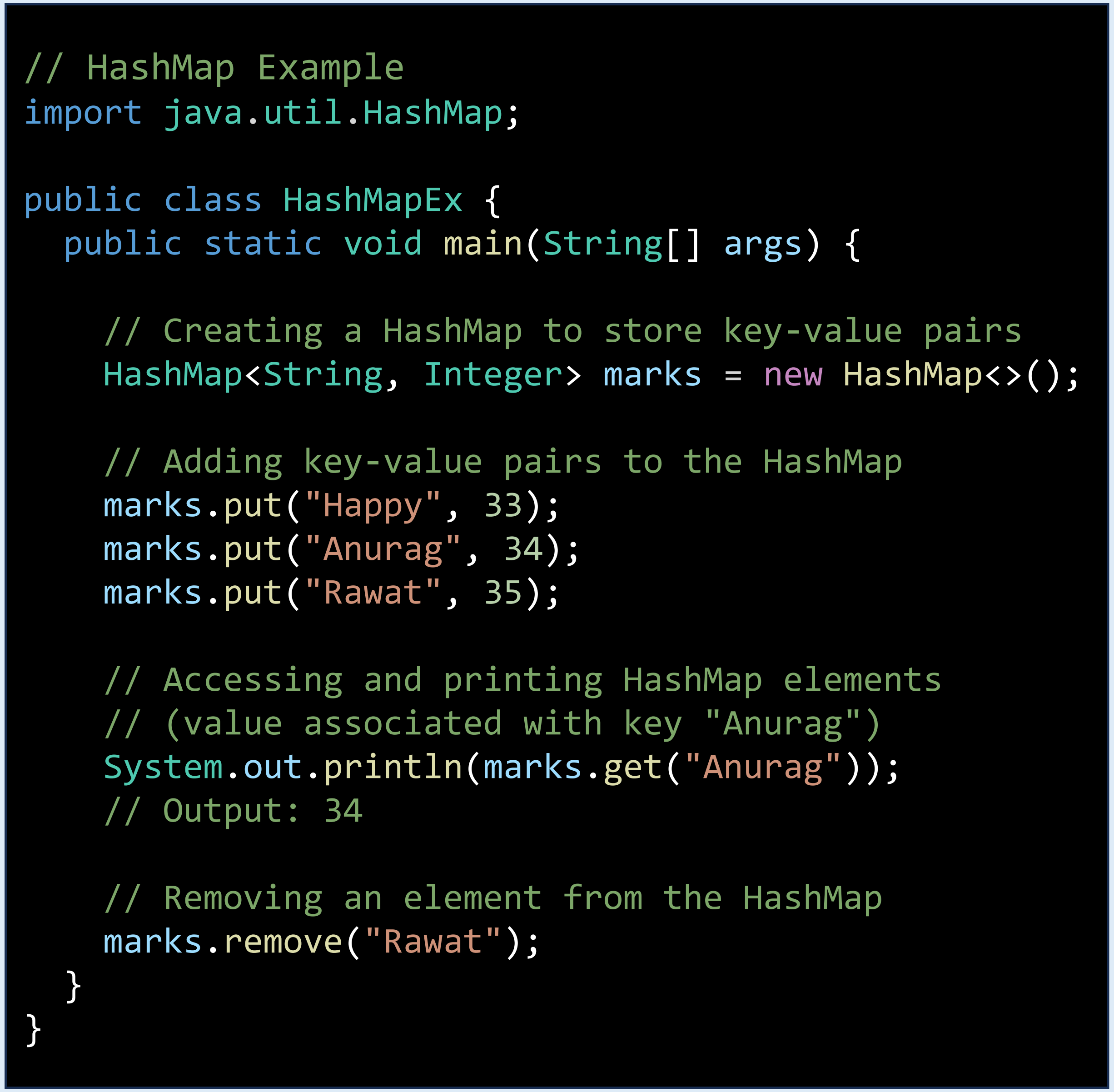
back to chapter index

# Q. What is the difference between HashMap and HashTable?

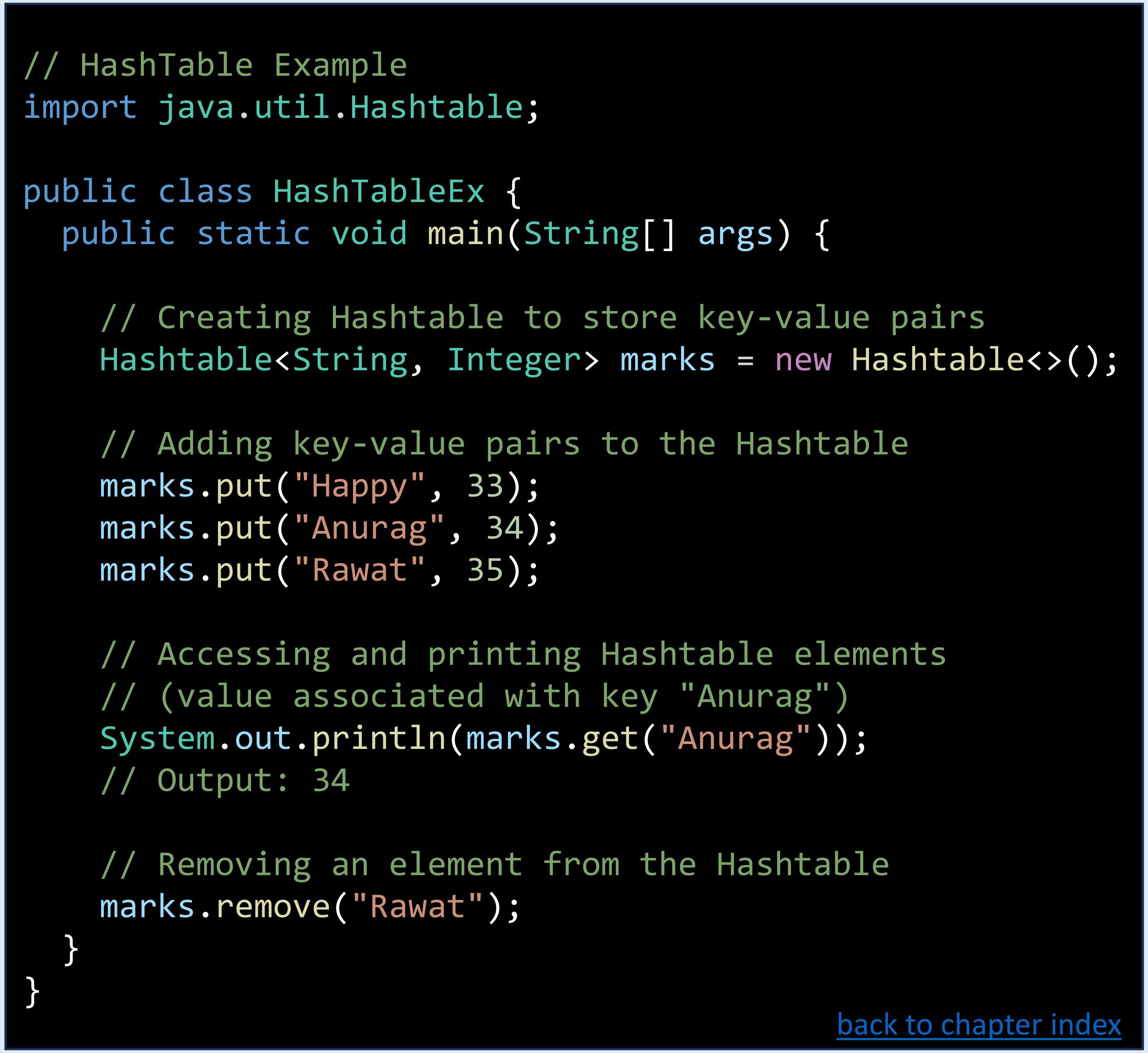


- HashMap and HashTable are classes that implements the Map interface and represents a collection of key-value pairs.

# Q. What is the difference between HashMap and HashTable?



```java
// HashMap Example
import java.util.HashMap;

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

    // Creating a HashMap to store key-value pairs
    HashMap<String, Integer> marks = new HashMap<>();

    // Adding key-value pairs to the HashMap
    marks.put("Happy", 33);
    marks.put("Anurag", 34);
    marks.put("Rawat", 35);

    // Accessing and printing HashMap elements
    // (value associated with key "Anurag")
    System.out.println(marks.get("Anurag"));
    // Output: 34

    // Removing an element from the HashMap
    marks.remove("Rawat");
  }
}
```



```java
// HashTable Example
import java.util.Hashtable;

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

    // Creating Hashtable to store key-value pairs
    Hashtable<String, Integer> marks = new Hashtable<>();

    // Adding key-value pairs to the Hashtable
    marks.put("Happy", 33);
    marks.put("Anurag", 34);
    marks.put("Rawat", 35);

    // Accessing and printing Hashtable elements
    // (value associated with key "Anurag")
    System.out.println(marks.get("Anurag"));
    // Output: 34

    // Removing an element from the Hashtable
    marks.remove("Rawat");
  }
}
```

back to chapter index

# Q. What is the difference between HashMap and HashTable?

| HashMap | HashTable |
|---|---|
| 1. Allows one null key and multiple null values | Does not allow null keys or values |
| 2. Not synchronized, not thread-safe | Synchronized, thread-safe |
| 3. Generally faster (no synchronization overhead) | Slower due to synchronization overhead |
| 4. Use in single-threaded applications | Use in multi-threaded applications |

back to chapter index

# Q. What are the advantages of using collections? V. IMP.

**1. Grouping of Elements:** Collections allow you to group multiple elements into a single entity.

**2. Dynamic Size:** Collections can grow or shrink dynamically based on the number of elements they contain, unlike traditional arrays whose size is fixed.

**3. Ready-to-Use Data Structures** Java collections framework provides a set of ready-to-use data structures (such as lists, sets, maps, queues, etc.)

**4. Efficient Manipulation:** Collections offer methods to add, remove, and modify elements efficiently, without needing to handle low-level details.

**5. Improved Code Quality** By using collections, developers can write cleaner, more modular, and reusable code, leading to better software design and maintenance.

back to chapter index

# 19. Multithreading - Overview

Q. What is Process and Thread? What is the difference between them?

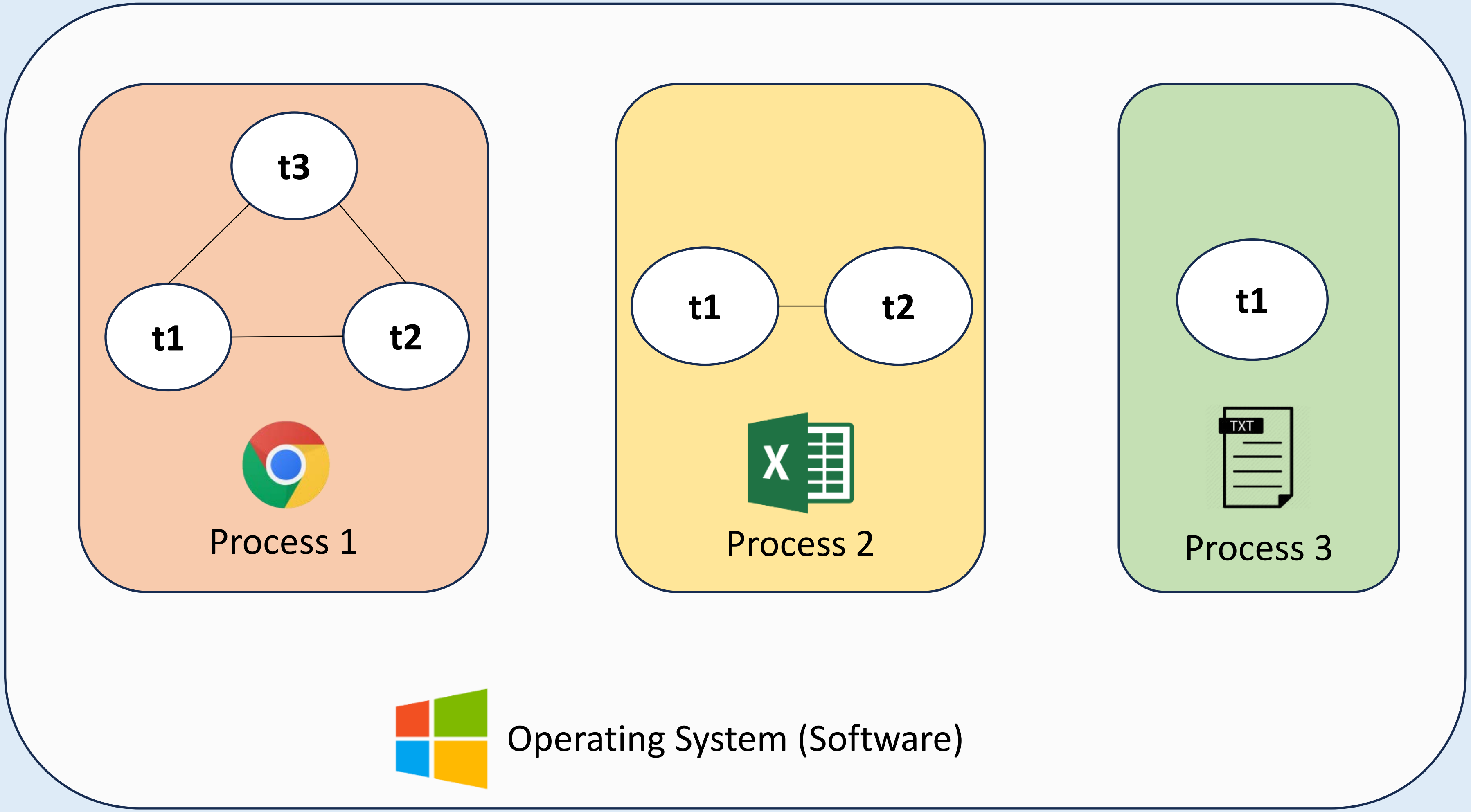
Q. What is the difference between Process and Thread?

Q. Explain Multithreading? What is the advantage of it? **V. IMP.**

Q. What is Main Thread & Daemon Thread in Java?

Back to main index

# Q. What is Process and Thread? What is the difference between them?



back to chapter index

# Q. What is the difference between Process and Thread?

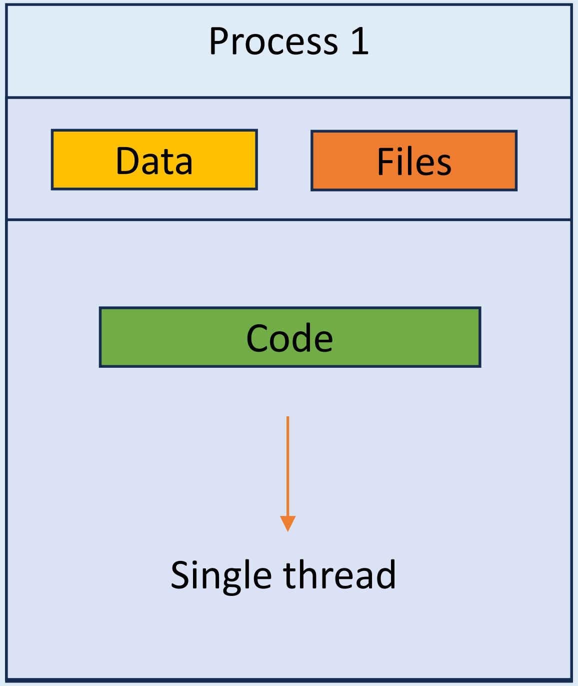
| Process | Thread |
|---|---|
| 1. A process is an instance of a program with its **own memory** space and system resources. | A thread is the smallest unit of process, that **shares memory** and resources with other threads within the same process. |
| 2. Processes are **independent** of each other. Changes in one process do not affect other processes. | Threads are **not independent**; they share resources such as memory, files, and I/O with other threads within the same process. |

back to chapter index

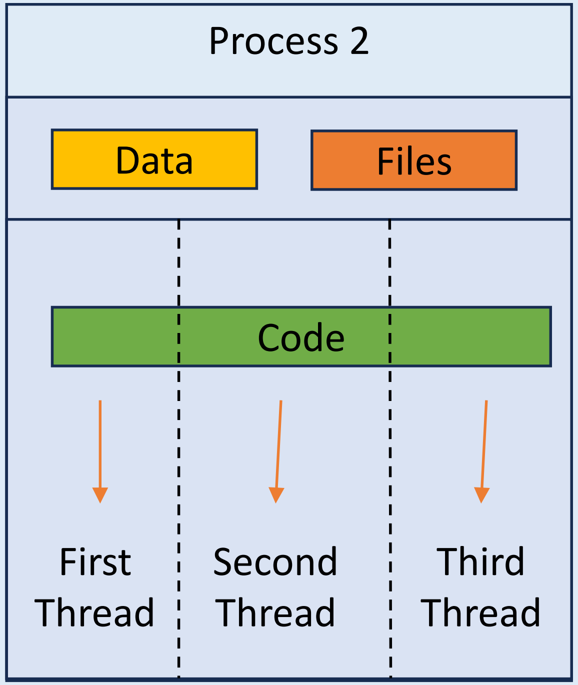
# Q. Explain Multithreading? What is the advantage of it? V. IMP.

- **Definition:** Multithreading refers to the ability to execute multiple threads of code **concurrently** within a single process.

- **Advantage:** Multithreading allows you to perform multiple tasks simultaneously, which can increase the performance.



Single process 1 with single thread

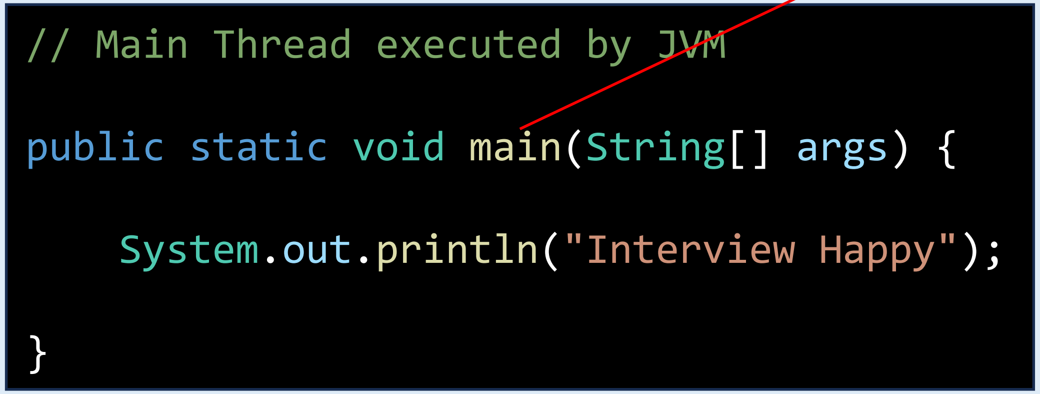


Single process 2 with multiple threads

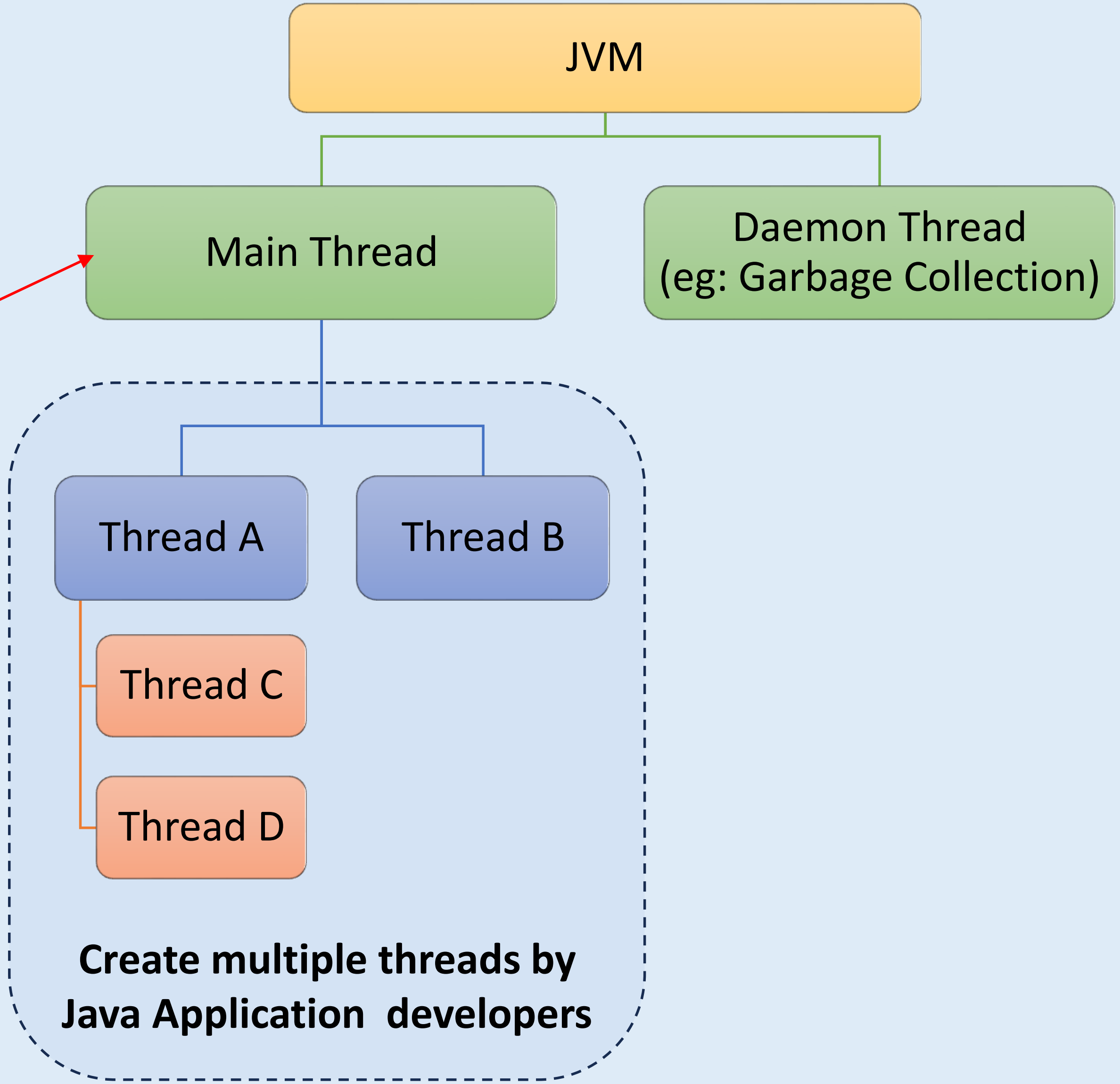
back to chapter index

# Q. What is Main Thread & Daemon Thread in Java?

- Main Thread is the primary thread created by the Java Virtual Machine (JVM) which executes the main() method.
- Daemon threads are background threads that provide supporting services. For example, Garbage Collection.



```
// Main Thread executed by JVM

public static void main(String[] args) {

    System.out.println("Interview Happy");

}
```



back to chapter index

# 20. Multithreading - Implementation

Q. In how many ways we can implement multithreading in Java? **V. IMP.**

Q. How to implement multithreading using Thread class?

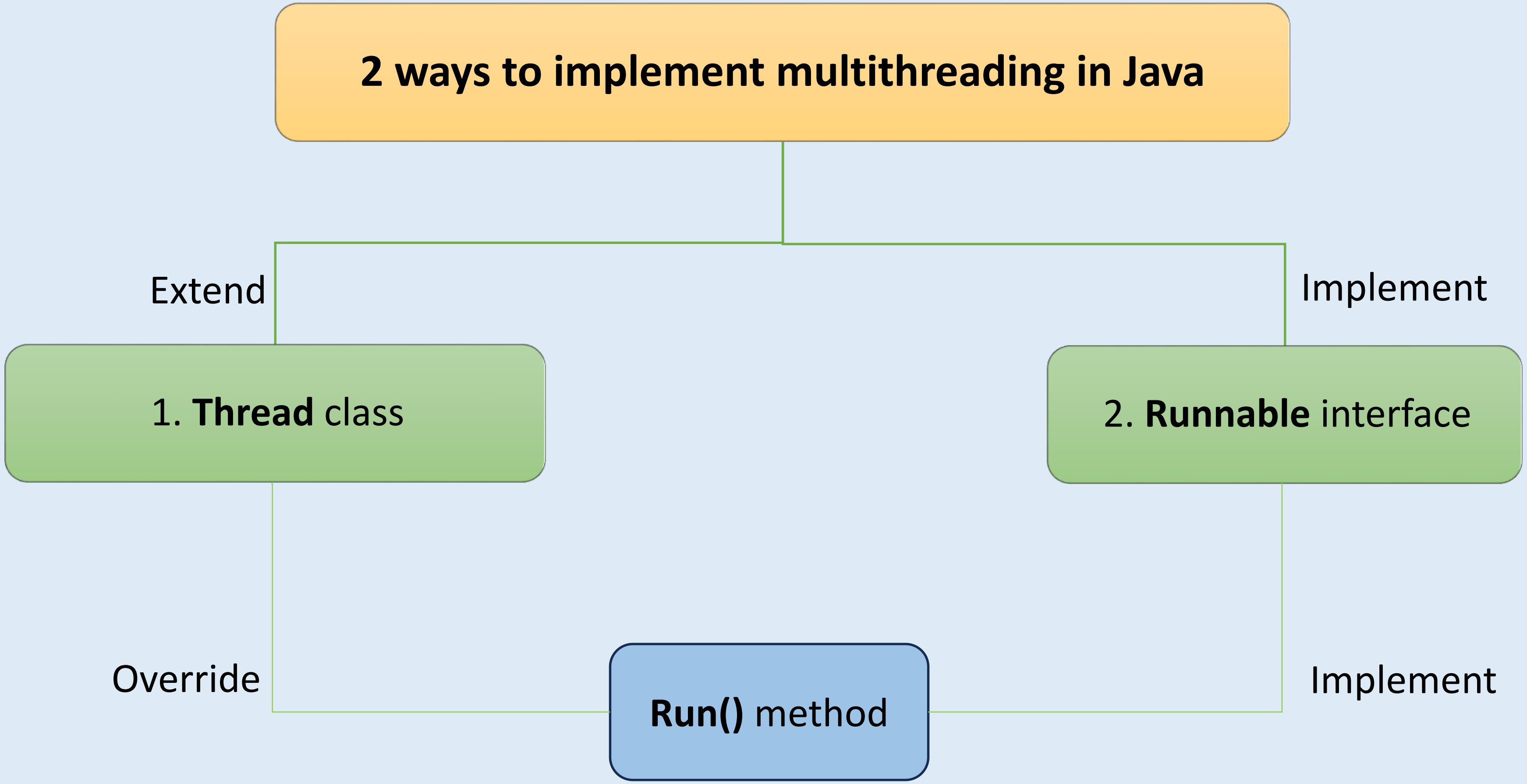
Q. How to implement multithreading using Runnable Interface? **V. IMP.**

Q. What is the differences between Thread class & Runnable interface? **V. IMP.**

Q. What are some important methods of Thread class?

Back to main index

# Q. In how many ways we can implement multithreading in Java? V. IMP.



2 ways to implement multithreading in Java

Extend

Implement

1. **Thread** class

2. **Runnable** interface

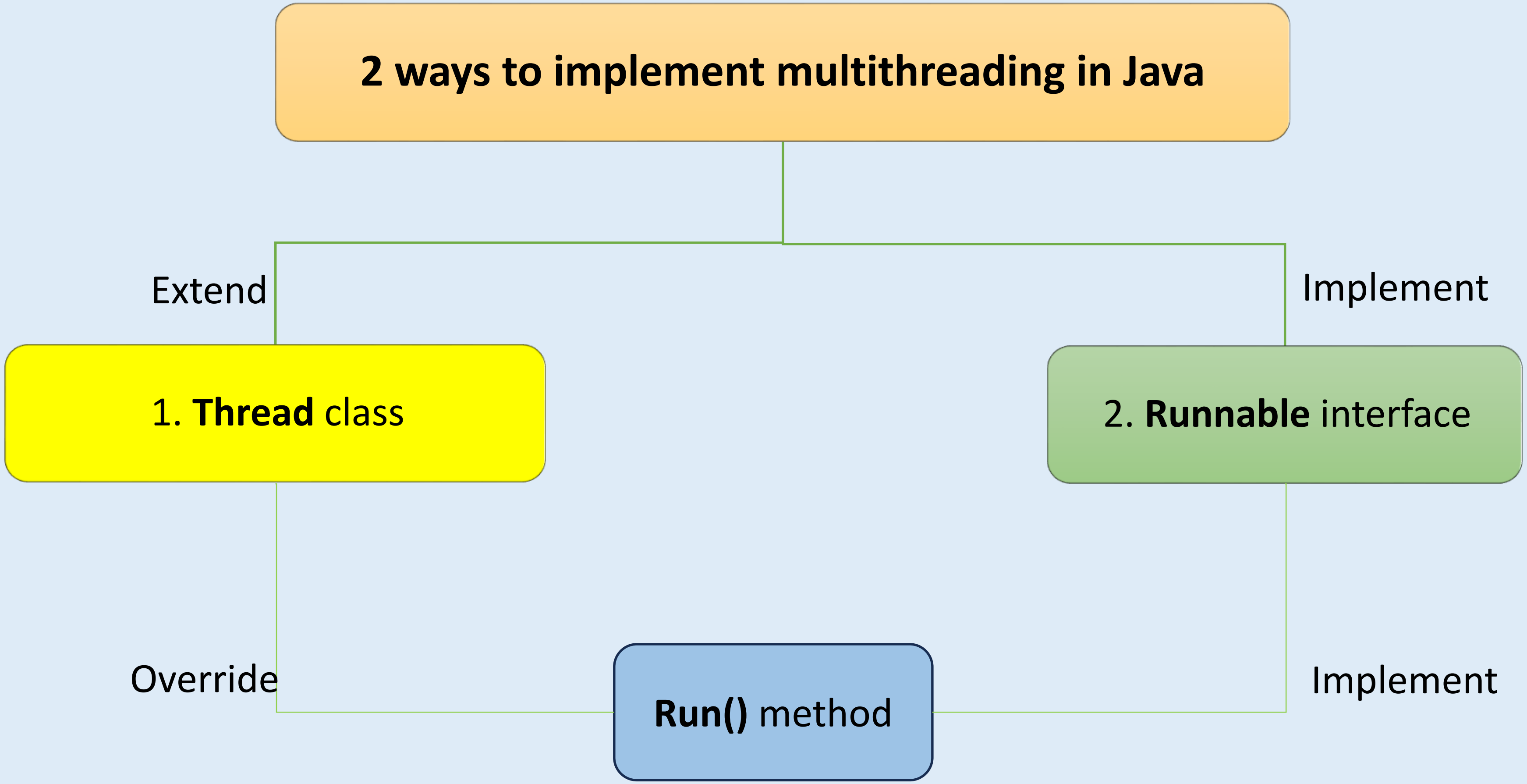
Override

**Run()** method

Implement

back to chapter index

# Q. How to implement multithreading using Thread class?



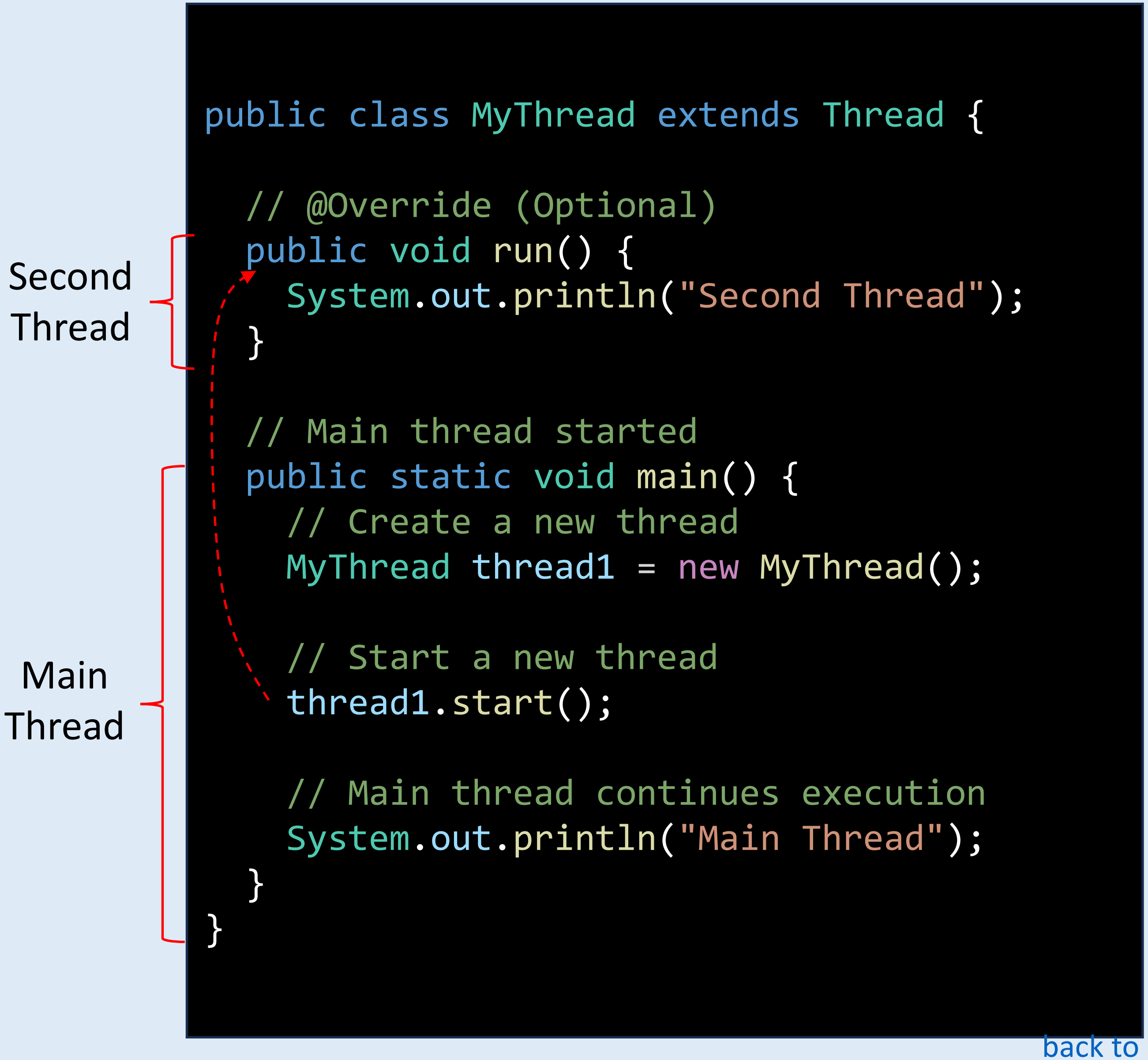
2 ways to implement multithreading in Java
Extend
Implement
1. **Thread** class
2. **Runnable** interface
Override
Implement
**Run()** method

back to chapter index

# Q. How to implement multithreading using Thread class?

❖ **Steps to implement multithreading using Thread class:**

1. Define a class that extends Thread class.
2. Override the run() method with the code you want to run concurrently.
3. Instantiate the subclass and call its start() method to initiate the thread's execution.
4. The run() method will execute concurrently in a separate thread.



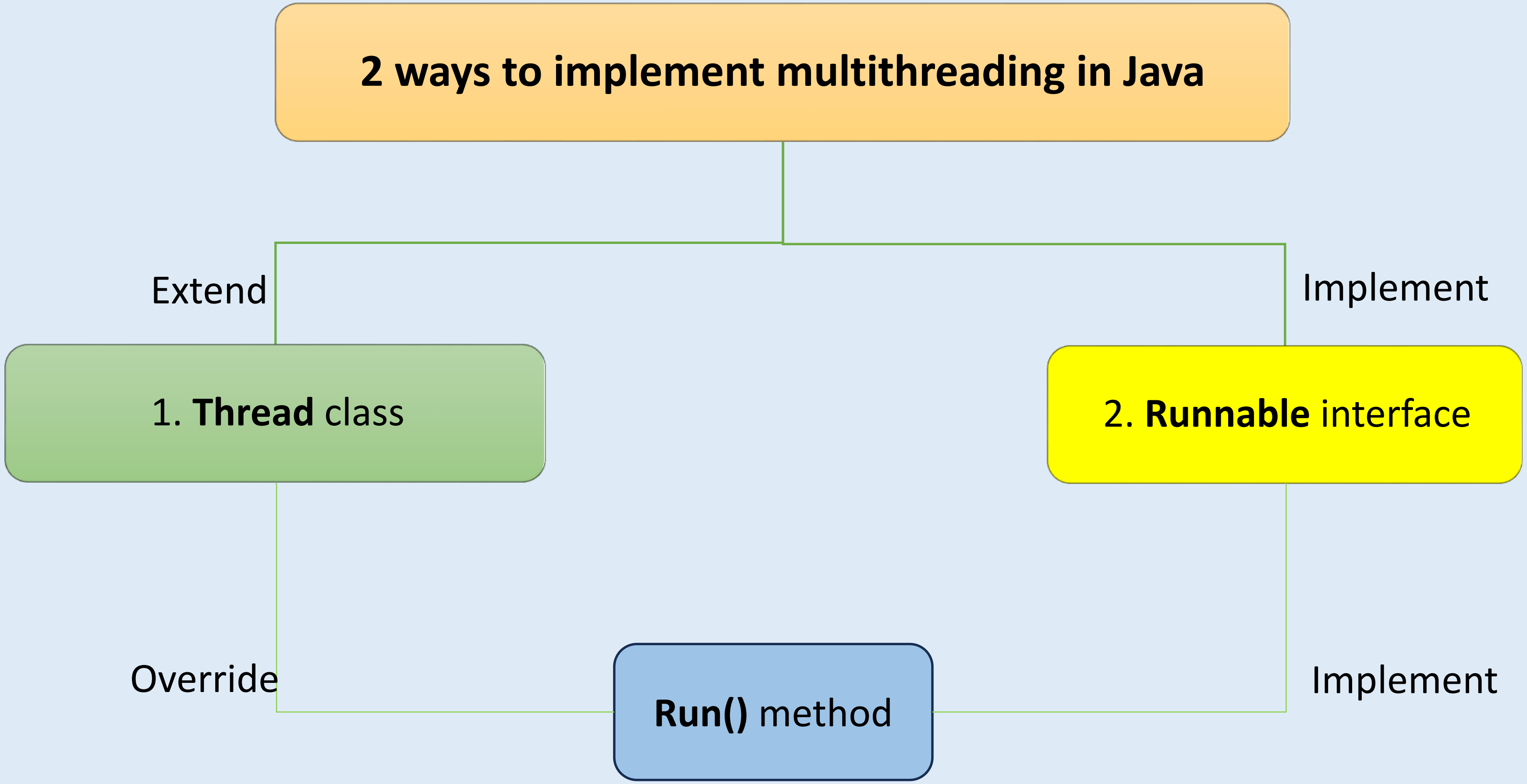
Second Thread

Main Thread

```
public class MyThread extends Thread {

  // @Override (Optional)
  public void run() {
    System.out.println("Second Thread");
  }

  // Main thread started
  public static void main() {
    // Create a new thread
    MyThread thread1 = new MyThread();

    // Start a new thread
    thread1.start();

    // Main thread continues execution
    System.out.println("Main Thread");
  }
}
```

back to chapter index

# Q. How to implement multithreading using Runnable Interface? V. IMP.


2 ways to implement multithreading in Java
Extend
Implement
1. **Thread** class
2. **Runnable** interface
Override
Implement
**Run()** method

back to chapter index

# Q. How to implement multithreading using Runnable Interface? V. IMP.

```java
public class MyThread extends Thread {

  // @Override (Optional)
  public void run() {
     System.out.println("Thread2");
  }

  // Main thread entry point
  public static void main() {

    //Create new thread
    MyThread thread1 = new MyThread();

    // Start a new thread
    thread1.start();

    // Main thread continues execution
    System.out.println("Main Thread");
  }
}
```

```java
public class MyRunnable implements Runnable {

  public void run() {
      System.out.println("Thread2");
  }

  // Main thread entry point
  public static void main(String[] args) {

      // Create a new instance of MyRunnable
      MyRunnable myRunnable = new MyRunnable();

      // Create a new thread passing myRunnable
        as the Runnable target
      Thread thread1 = new Thread(myRunnable);

      // Start the newly created thread
      thread1.start();
  }
}
```

back to chapter index

# Q. What is the differences between Thread class & Runnable interface? V. IMP.

```java
public class MyThread extends Thread {

  // @Override (Optional)
  public void run() {
     System.out.println("Thread2");
  }

  // Main thread entry point
  public static void main() {

    //Create new thread
    MyThread thread1 = new MyThread();

    // Start a new thread
    thread1.start();

    // Main thread continues execution
    System.out.println("Main Thread");
  }
}
```

```java
public class MyRunnable implements Runnable {

  public void run() {
      System.out.println("Thread2");
  }

  // Main thread entry point
  public static void main(String[] args) {

      // Create a new instance of MyRunnable
      MyRunnable myRunnable = new MyRunnable();

      // Create a new thread passing myRunnable
        as the Runnable target
      Thread thread1 = new Thread(myRunnable);

      // Start the newly created thread
      thread1.start();
  }
}
```

back to chapter index

# Q. What is the differences between Thread class & Runnable interface? V. IMP.

| Thread class | Runnable interface |
|---|---|
| Both are used for implementing multi-threading | |
| 1. Extends Thread class directly | Implements Runnable interface |
| 2. Your class can extend only one Thread class due to the limitation of multiple inheritance. | Your class can implement multiple interfaces along with Runnable interface. 🏆 |
| 3. Different Thread instances are required for different threads. | Same Runnable instance can be used for multiple threads 🏆 |
| 4. Less flexible | More flexible for managing tasks independently of threads |

back to chapter index

# Q. What are some important methods of Thread class?

```java
public class ThreadMethodsEx extends Thread {

  // @Override (Optional)
  public void run() { // Another thread started
concurrently
    String threadName =
Thread.currentThread().getName();
    System.out.println("ThreadMethodsEx: " +
threadName);
  }

  public static void main(String[] args) {
    ThreadMethodsEx thread = new
ThreadMethodsEx();

    // 1. Start the thread
    thread.start();

    // 2. Check thread status: true/ false
    if (thread.isAlive()) { // true
      System.out.println("Thread is running");
    }
```

```java
// 3. Get thread state: NEW/ RUNNABLE/ BLOCKED/
WAITING/ TERMINATED
    Thread.State state = thread.getState();
    System.out.println(state); // RUNNABLE

    // 4. Get thread priority: 1 to 10
    System.out.println(thread.getPriority()); // 5

    // 5. Set thread priority: MAX_PRIORITY = 10
    thread.setPriority(Thread.MAX_PRIORITY);

    // 6. Get thread priority
    System.out.println(thread.getPriority()); // 10

    try {
      // 7. Thread Sleeping(waiting/ pause)
      Thread.sleep(1000); // Sleep for milliseconds
    } catch (InterruptedException e) {
      System.out.println("Thread sleeping");
    }
  }
}
```

back to chapter index

# 21. Generics - Basics

Q. What is the role of Generics in Java? **V. IMP.**

Q. What is a Generic method? How to implement Generic method? **V. IMP.**

Q. What is a Generic class? How to implement Generic class? **V. IMP.**

Q. What are type parameters and type arguments? What is T in Generic class?

Q. When to use Generic method and when to use Generic class in Java?

Back to main index

# Q. What is the role of Generics in Java? V. IMP.

- Generics provide a way to define classes, interfaces, and methods that can work with **any data type**.

**Compare (class without generics)**

- areEqual(int num1, int num2)

- areEqual(String str1, String str2)

Different methods for handling different types

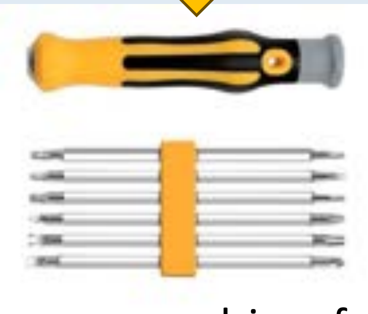
Different screwdrivers for handling different needles

**Generic version**

**Compare (class with generic methods)**

- areEqual(T arg1, T arg2)

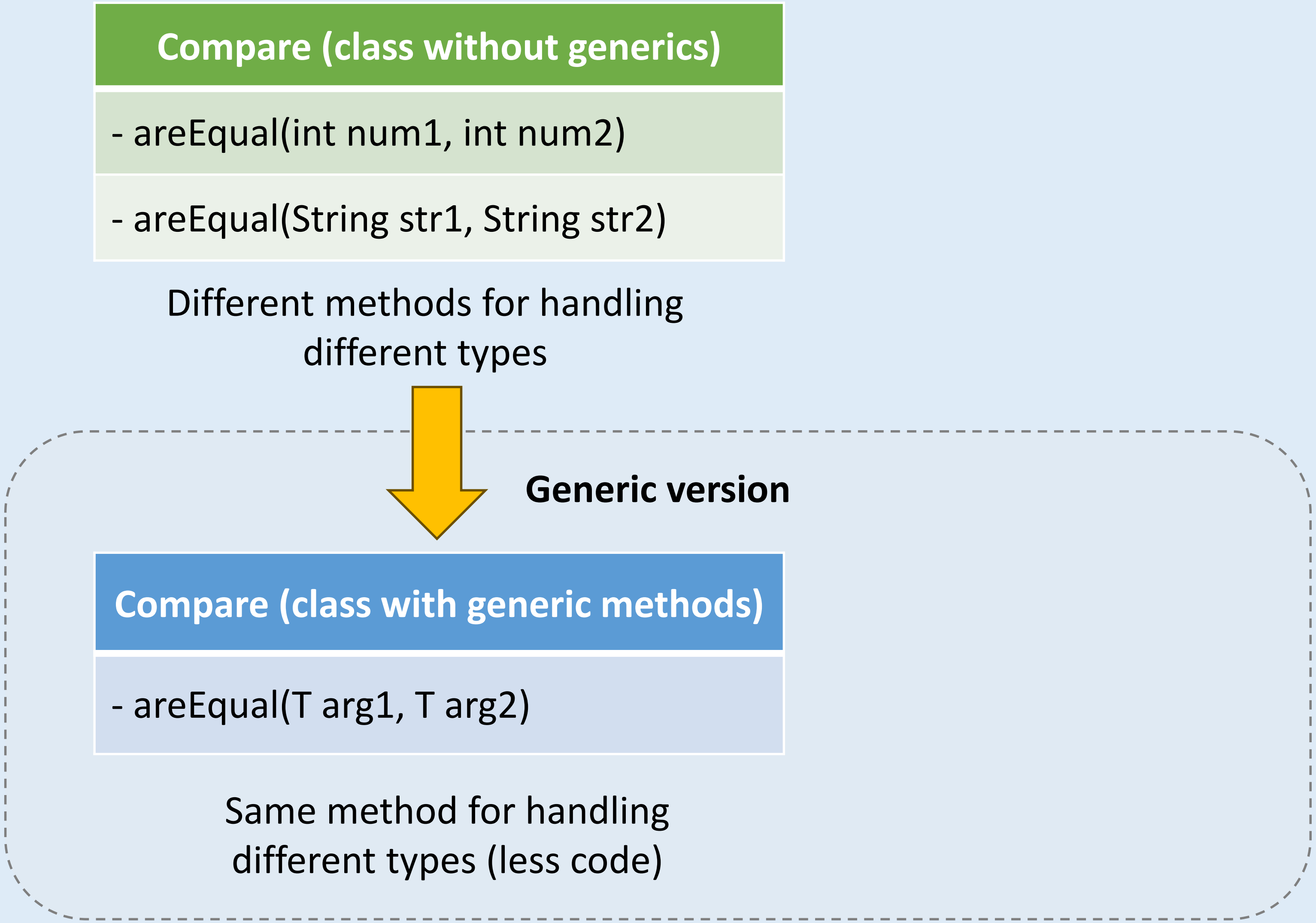
Same method for handling different types (less code)



Same screwdriver for handling different needles

back to chapter index

# Q. What is a Generic method? How to implement Generic method? V. IMP.



**Compare (class without generics)**

- areEqual(int num1, int num2)

- areEqual(String str1, String str2)

Different methods for handling different types

**Generic version**

**Compare (class with generic methods)**

- areEqual(T arg1, T arg2)

Same method for handling different types (less code)

back to chapter index

# Q. What is a Generic method? How to implement Generic method? V. IMP.

```
// Without Generics
public class Compare {
 public boolean areEqual(int value1, int value2) {
    return value1 == value2;
  }

  // Overloaded method to handle string type
 public boolean areEqual(String value1, String
value2) {
    return value1 == value2;
  }

  public static void main(String[] args) {
    Compare comp = new Compare();
    // Comparing integers
    boolean intResult = comp.areEqual(10, 10);
    System.out.println(intResult); // Output: true

    // Comparing strings will not work
    // because areEqual method is not type safe
    boolean strResult = comp.areEqual("ab", "ab");
    System.out.println(strResult); // Output: true
  }
}
```

```
// With Generics
public class GenericMethodEx {

  // Generic method to compare two values of type T
  public <T> boolean areEqual(T value1, T value2) {
    return value1 == value2;
  }

  public static void main(String[] args) {
    GenericMethodEx comp = new GenericMethodEx();

    // Comparing integers using generics
    boolean intResult = comp.areEqual(10, 10);
    System.out.println(intResult); // Output: true

    // Comparing strings using generics
    boolean strResult = comp.areEqual("ab", "ab");
    System.out.println(strResult); // Output: true
  }
}
```

back to chapter index

# Q. What is a Generic class? How to implement Generic class?

- To implement a generic class:

1. Use <T> after the class name.
2. Use the type parameter (T) with the methods inside the class.
3. Instantiate the generic class with specific types.

```java
public class GenericClassEx<T> {

  public boolean areEqual(T value1, T value2) {
    return value1 == value2;
  }

  public boolean notEqual(T value1, T value2) {
    return value1 != value2;
  }

  public static void main(String[] args) {
    GenericClassEx<Integer> intComp = new GenericClassEx<>();
    boolean intResult = intComp.areEqual(10, 10);
    System.out.println(intResult); // Output: true

    GenericClassEx<String> strComp = new GenericClassEx<>();
    boolean strResult = strComp.areEqual("ab", "ab");
    System.out.println(strResult); // Output: true

    boolean strNotEqualResult = strComp.notEqual("ab", "xy");
    System.out.println(strNotEqualResult); // Output: true
  }
}
```

back to chapter index

# Q. What are type parameters and type arguments? What is T in Generic class?

- T is the type parameter.
- Type parameter(T) is a placeholder for type which allows class to be generic. Meaning T can hold any specified type.
- T identifier is commonly used for Type parameter, but you can use any X, Y, Z etc.

- Type arguments are the actual types when creating instances of the generic classes.

```java
public class GenericClassEx<T> {

  public boolean areEqual(T value1, T value2) {
    return value1 == value2;
  }

  public boolean notEqual(T value1, T value2) {
    return value1 != value2;
  }

  public static void main(String[] args) {
    GenericClassEx<Integer> intComp = new GenericClassEx<>();
    boolean intResult = intComp.areEqual(10, 10);
    System.out.println(intResult); // Output: true

    GenericClassEx<String> strComp = new GenericClassEx<>();
    boolean strResult = strComp.areEqual("ab", "ab");
    System.out.println(strResult); // Output: true

    boolean strNotEqualResult = strComp.notEqual("ab", "xy");
    System.out.println(strNotEqualResult); // Output: true
  }
}
```

back to chapter index

# Q. When to use Generic method and when to use Generic class in Java?

## When to use Generic method?

- If you only need the generic type within a single method or a few methods only.

## When to use Generic class?

- If you only need the generic type within all the methods of that class.

back to chapter index

# 22. Generics - Advanced

Q. What are bounded type parameters in generics?

Q. Can primitive types be used as type arguments in generics?

Q. How to use Generics with Collections?

Q. What is Type Safety? How generics provide type safety?

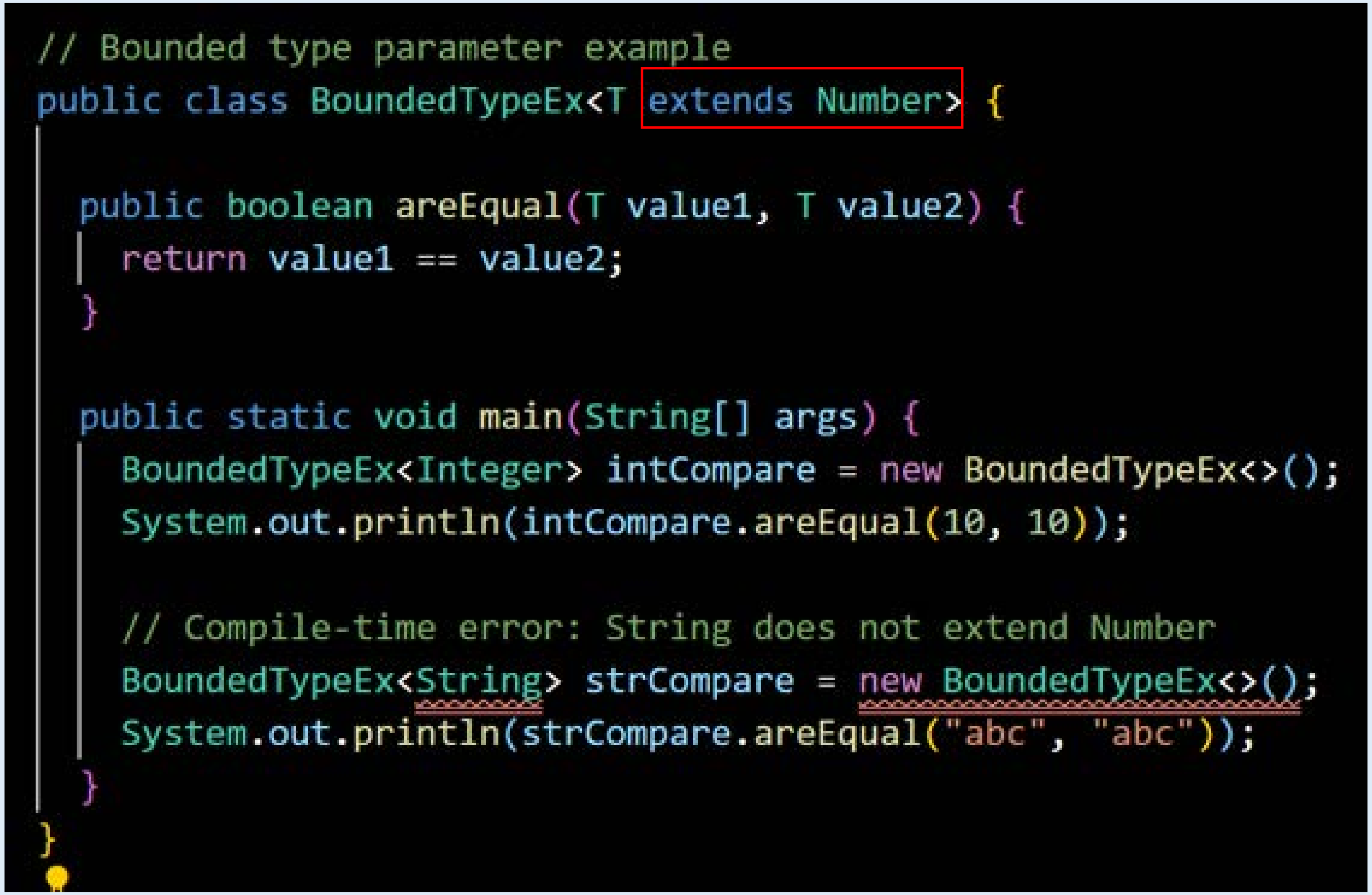
Q. What is Type casting? How generics eliminates type casting?

Q. What are the advantages of using Generics? **V. IMP.**

Back to main index

# Q. What are bounded type parameters in generics?

- Bounded type parameters in generics allow you to restrict the types that can be used as arguments for a type parameter in a generic class.
- For example, T extends Number means when the instance of BoundedTypeEx is created, then only integer can be the type.
- Bounded type parameter provide type safety and compile time error checking's, but it restricts the type flexibility.



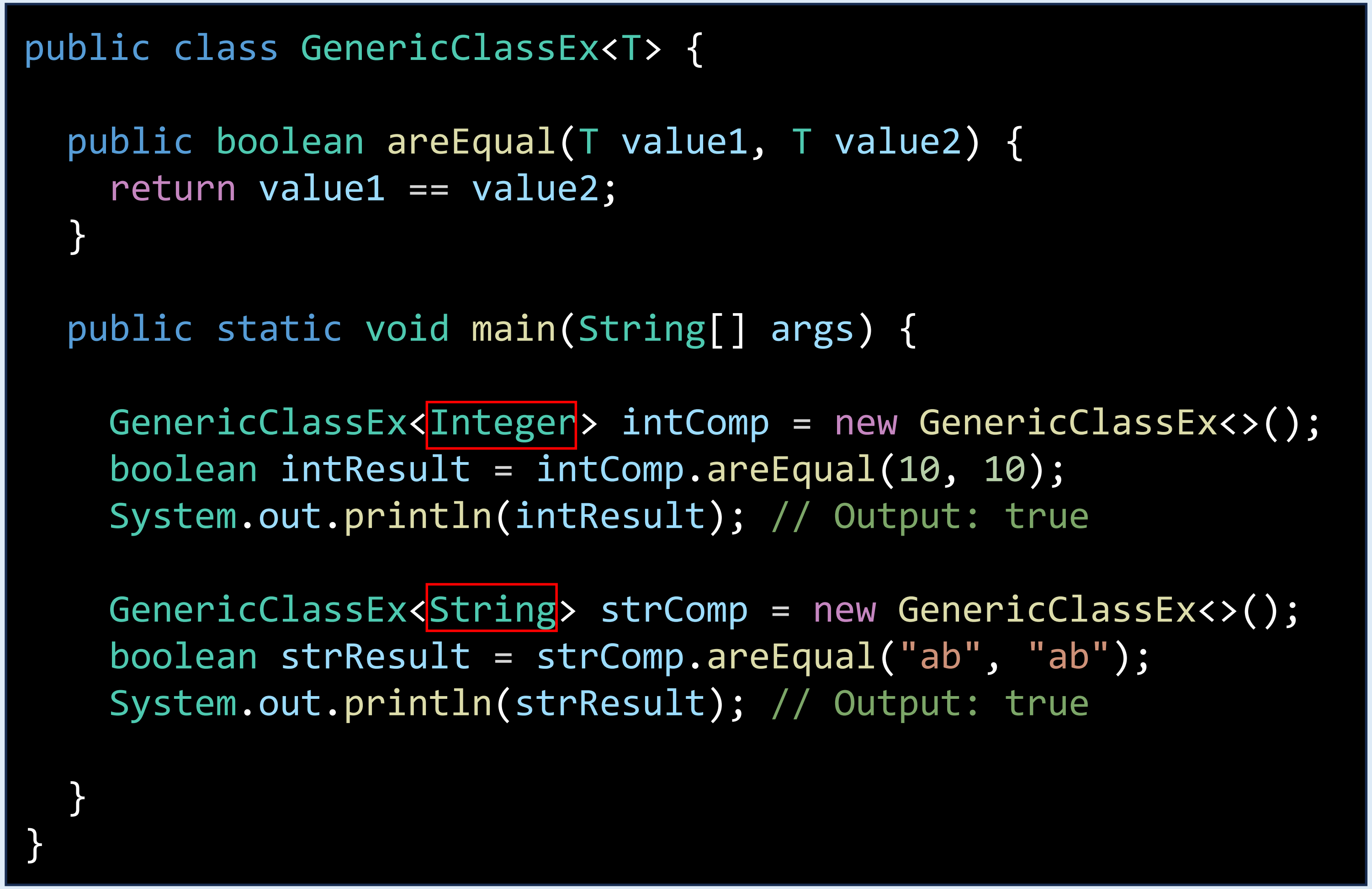
```
// Bounded type parameter example
public class BoundedTypeEx<T extends Number> {

    public boolean areEqual(T value1, T value2) {
        return value1 == value2;
    }

    public static void main(String[] args) {
        BoundedTypeEx<Integer> intCompare = new BoundedTypeEx<>();
        System.out.println(intCompare.areEqual(10, 10));

        // Compile-time error: String does not extend Number
        BoundedTypeEx<String> strCompare = new BoundedTypeEx<>();
        System.out.println(strCompare.areEqual("abc", "abc"));
    }
}
```

back to chapter index

# Q. Can primitive types be used as type arguments in generics?

- No, primitive types (int, double, char, etc.) cannot be used directly as type arguments in generics in Java.

- Generics in Java are designed to work with reference types (classes and interfaces) rather than primitive types.

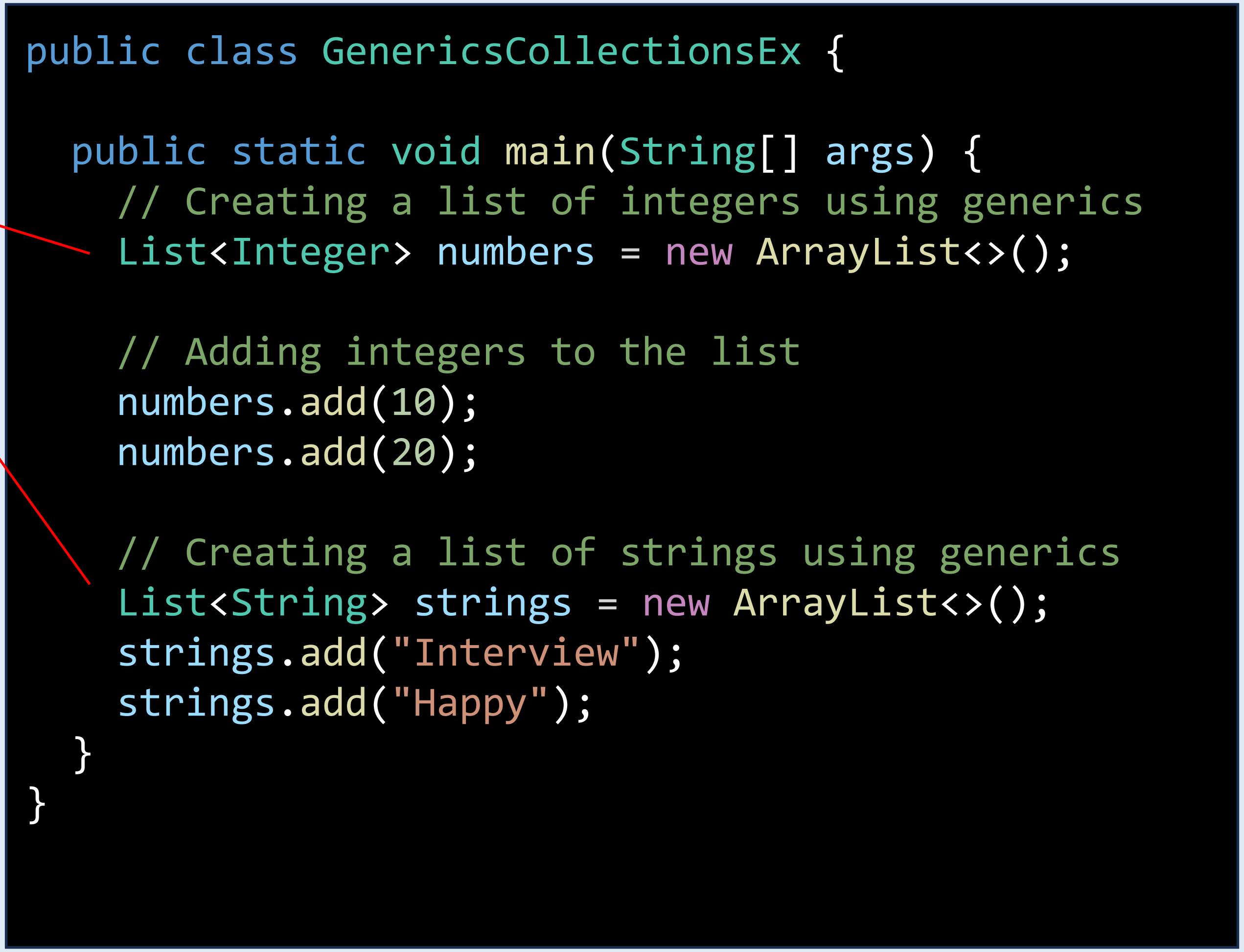


```java
public class GenericClassEx<T> {

  public boolean areEqual(T value1, T value2) {
    return value1 == value2;
  }

  public static void main(String[] args) {

    GenericClassEx<Integer> intComp = new GenericClassEx<>();
    boolean intResult = intComp.areEqual(10, 10);
    System.out.println(intResult); // Output: true

    GenericClassEx<String> strComp = new GenericClassEx<>();
    boolean strResult = strComp.areEqual("ab", "ab");
    System.out.println(strResult); // Output: true

  }
}
```

back to chapter index

# Q. How to use Generics with Collections?

- Java in-built collection classes or interfaces are generic classes or interfaces which can accept the type parameter(<T>) and handle any type at run time



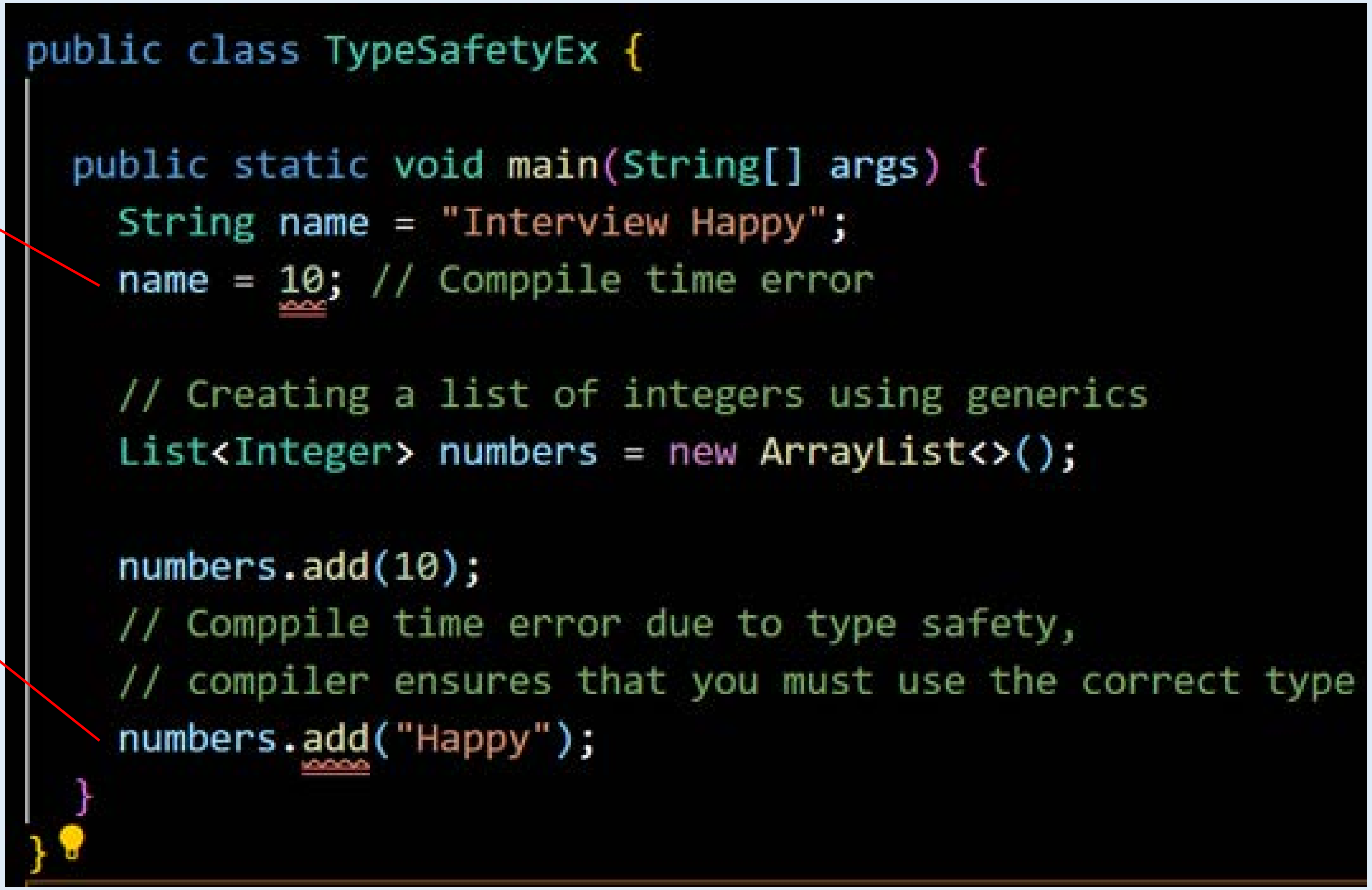
```java
public class GenericsCollectionsEx {

  public static void main(String[] args) {
    // Creating a list of integers using generics
    List<Integer> numbers = new ArrayList<>();

    // Adding integers to the list
    numbers.add(10);
    numbers.add(20);

    // Creating a list of strings using generics
    List<String> strings = new ArrayList<>();
    strings.add("Interview");
    strings.add("Happy");
  }
}
```

back to chapter index

# Q. What is Type Safety? How generics provide type safety?

- Type safety means that the compiler will validate types while compiling and show an error if you try to assign the wrong type to a variable.

- Generics provide type safety by showing a compile-time error if the wrong type is assigned to a generic class.



```
public class TypeSafetyEx {

  public static void main(String[] args) {
    String name = "Interview Happy";
    name = 10; // Comppile time error

    // Creating a list of integers using generics
    List<Integer> numbers = new ArrayList<>();

    numbers.add(10);
    // Comppile time error due to type safety,
    // compiler ensures that you must use the correct type
    numbers.add("Happy");
  }
}
```

back to chapter index

# Q. What is Type casting? How generics eliminates type casting?

- Type casting in Java is the process of converting a variable from one data type to another.

- Without generics collection, items are converted to object type and then stored in the collections. To get items, as their specific types, explicit casting is required.

- With generics collection, items are stored and retrieved in their specific types, eliminating the need for explicit casting.

```java
public class TypeCastingEx {

  public static void main(String[] args) {
    // Without generics
    List list = new ArrayList();
    list.add("Hello");
    String str = (String) list.get(0); // Needs explicit cast

    // With generics
    List<String> stringList = new ArrayList<>();
    stringList.add("Hello");
    String str1 = stringList.get(0); // No cast needed
  }
}
```

back to chapter index

# Q. What are the advantages of using Generics? V. IMP.

**1. Code Reusability**

Generic classes and methods can be used with different data types, reducing code duplication.

**2. Type Safety**

With generics, the compiler can enforce type safety at compile time.

**3. Eliminating Type Casting**

Generics eliminate the need for explicit type casting, making code cleaner and more readable.

**4. Efficient with Collections**

Generics are extensively used in Java's Collection Framework, allowing type-safe storage and retrieval of objects in collections.

back to chapter index

# 23. Lambda expression

Q. What is lambda expression and lambda operator? **V. IMP.**

Q. What is the difference between normal interface and functional interface?

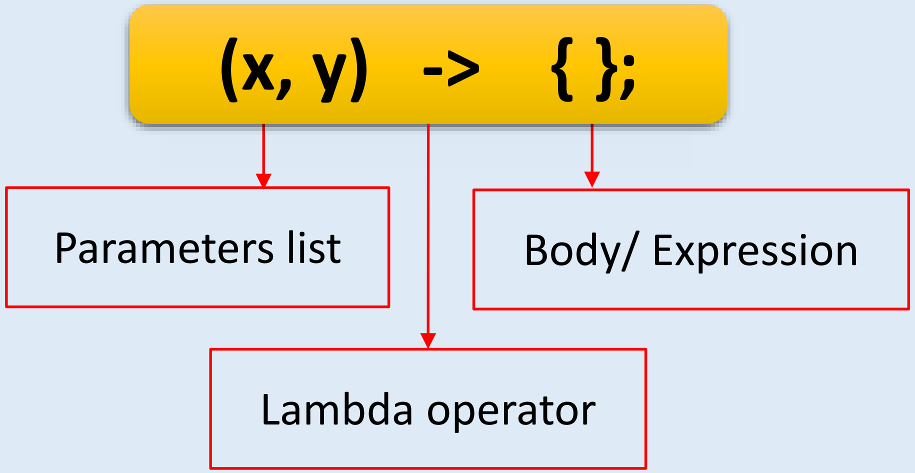
Q. How to use lambda expression to implement functional interfaces?

Q. Can we use lambda expression for non-functional interfaces?

Back to main index

# Q. What is lambda expression and lambda operator? V. IMP.

- A lambda expression is an **inline** and **shorthand** way of writing functionality.
- The lambda operator (->) separates the lambda expression's parameters from its body.
- When are lambda expressions mostly used? Lambda expressions are mostly used for implementing functional interfaces directly.



**Lambda Expression**

```java
public class ArrayListLambdaEx {

  public static void main(String[] args) {
    // Creating an ArrayList of strings
    List<String> list = new ArrayList<>();
    list.add("Apple");
    list.add("Banana");
    list.add("Orange");

    // Iterating over the list without lambda expression
    for (String fruit : list) {
      System.out.println(fruit);
    }
    // Output: Apple Banana Orange

    // Iterating over the list using lambda expression
    list.forEach((fruit) -> {System.out.println(fruit);});
    // Output: Apple Banana Orange
  }
}
```

back to chapter index

# Q. What is the difference between normal interface and functional interface?

- A normal interface in Java can contain multiple abstract methods.

```java
// Normal Interface

interface Shape {

  double area(); // First Abstract method

  double perimeter(); // Second Abstract method

}
```

- A functional interface is an interface that contains exactly one abstract method.

```java
// Functional Interface

// @FunctionalInterface(Optional)
interface Shape {

  double area(); // Single Abstract method

}
```

back to chapter index

# Q. How to use lambda expression to implement functional interfaces?

```java
// Without Lambda Expression Example

interface MyFunction { // Functional interface
  void sayHello(); // Single Abstract method
}

class MyFunctionImpl implements MyFunction {

  public void sayHello() {
    System.out.println("Interview Happy!");
  }
}

public class WithoutLambdaEx {

  public static void main(String[] args) {
    MyFunctionImpl myFunction = new MyFunctionImpl();

    myFunction.sayHello();
    // Output: Interview Happy!
  }
}
```

```java
// With Lambda Expression Example

interface MyFunction1 { // Functional interface
  void sayHello(); // Abstract method
}

public class WithLambdaEx {

  public static void main(String[] args) {
    // Creating an instance of the functional
    // interface using a lambda expression
    MyFunction1 myFunction = () -> {
      System.out.println("Interview Happy!");
    };

    myFunction.sayHello();
    // Output: Interview Happy!
  }
}
```

back to chapter index

# Q. Can we use lambda expression for non-functional interfaces?

- A lambda expression can be used for functional interfaces only.

```java
interface MyFunction { // Non-Functional interface
  void sayHello(); // First Abstract method
  void sayByee(); // First Abstract method
}

public class WithLambdaEx {

  public static void main(String[] args) {
    // Compile time error
    MyFunction myFunction = () -> {
      System.out.println("Interview Happy!");
    };
  }
}
```

# 24. Types of Classes - Inner class & Final class

Q. What are the types of classes in Java? V. IMP.

Q. What are Inner Classes (Nested Classes)? How to create instance of inner class?

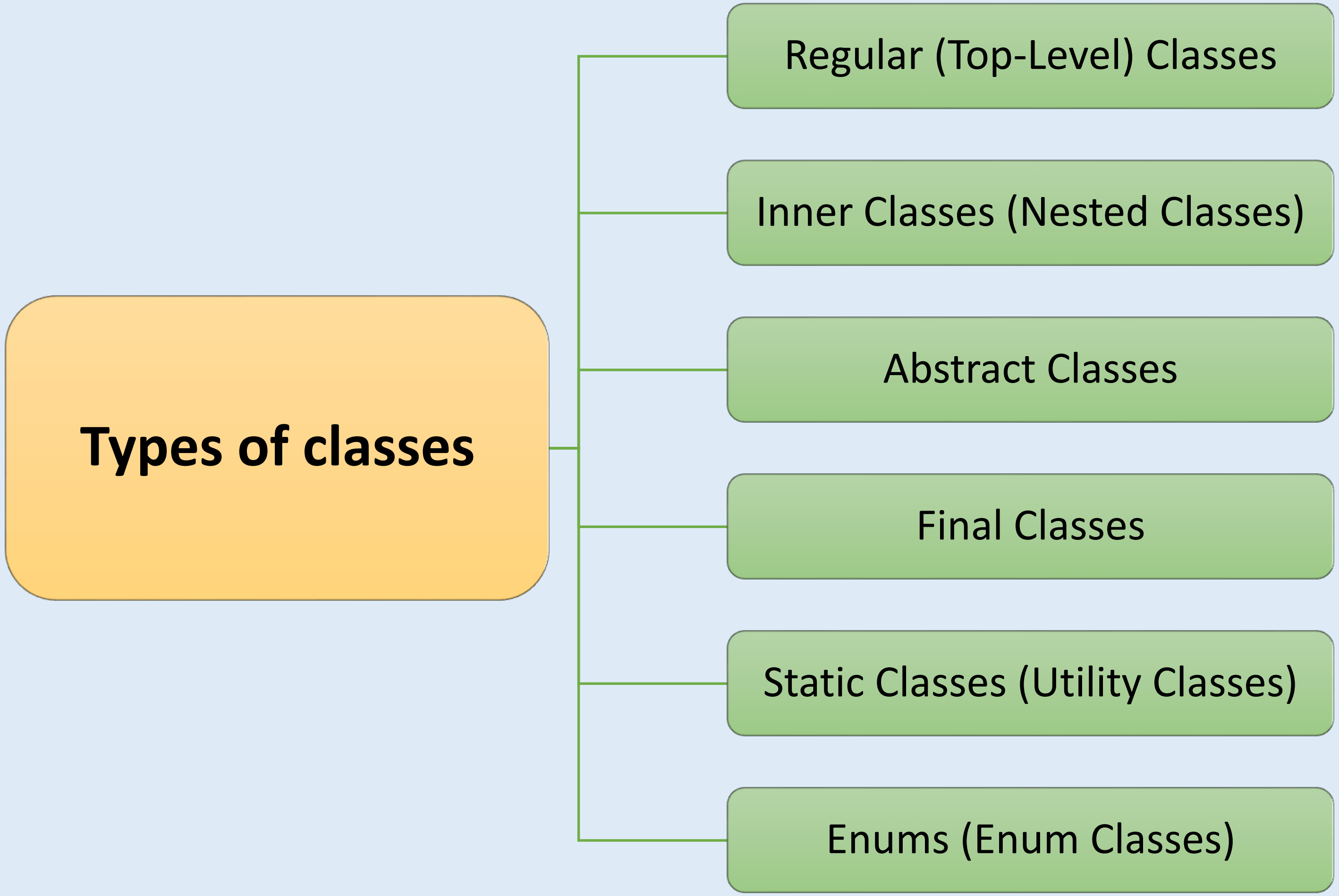
Q. What is Final class in Java?

Q. What is Final method in Java?

Q. What is the role of final keyword? V. IMP.

Back to main index

# Q. What are the types of classes in Java? V. IMP.



Types of classes

- Regular (Top-Level) Classes
- Inner Classes (Nested Classes)
- Abstract Classes
- Final Classes
- Static Classes (Utility Classes)
- Enums (Enum Classes)

back to chapter index

# Q. What are Inner Classes (Nested Classes)? How to create instance of inner class?

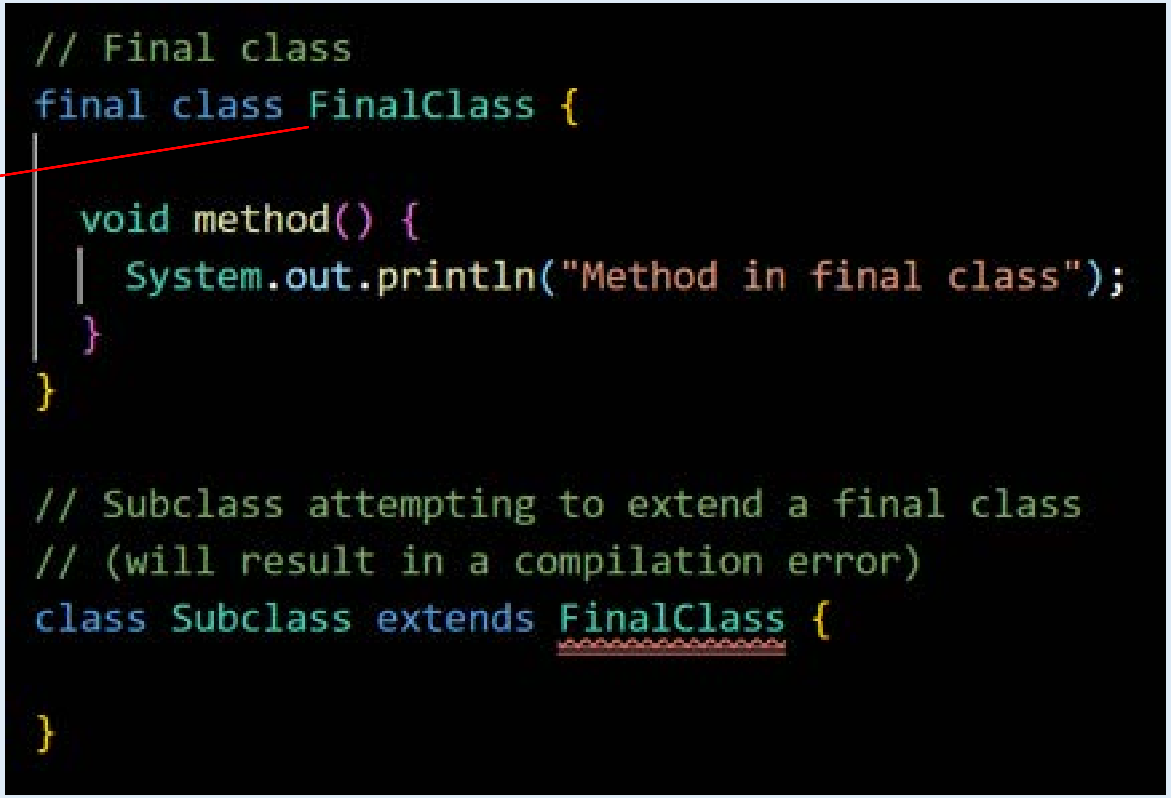
- An inner or nested class in Java is a class that is defined **within another class**.
- Inner classes have access to the members (fields and methods) of the outer class, including private members.

```java
// Outer Class
public class OuterInnerClassEx {

  private int outerField = 10;

  // Member inner class
  public class InnerClass {

    public void displayOuterField() {
      System.out.println(outerField);
    }
  }

  public static void main(String[] args) {
    // Create an instance of OuterInnerClassEx
    OuterInnerClassEx outer = new OuterInnerClassEx();

    // Create an instance of InnerClass
    // using the outer instance
    InnerClass inner = outer.new InnerClass();

    // Invoke method of InnerClass
    inner.displayOuterField(); // Output: 10
  }
}
```

back to chapter index

# Q. What is Final class in Java?

- final class is a class that cannot be subclassed or extended.
- This is useful when you want to prevent extension of a class for security reasons.

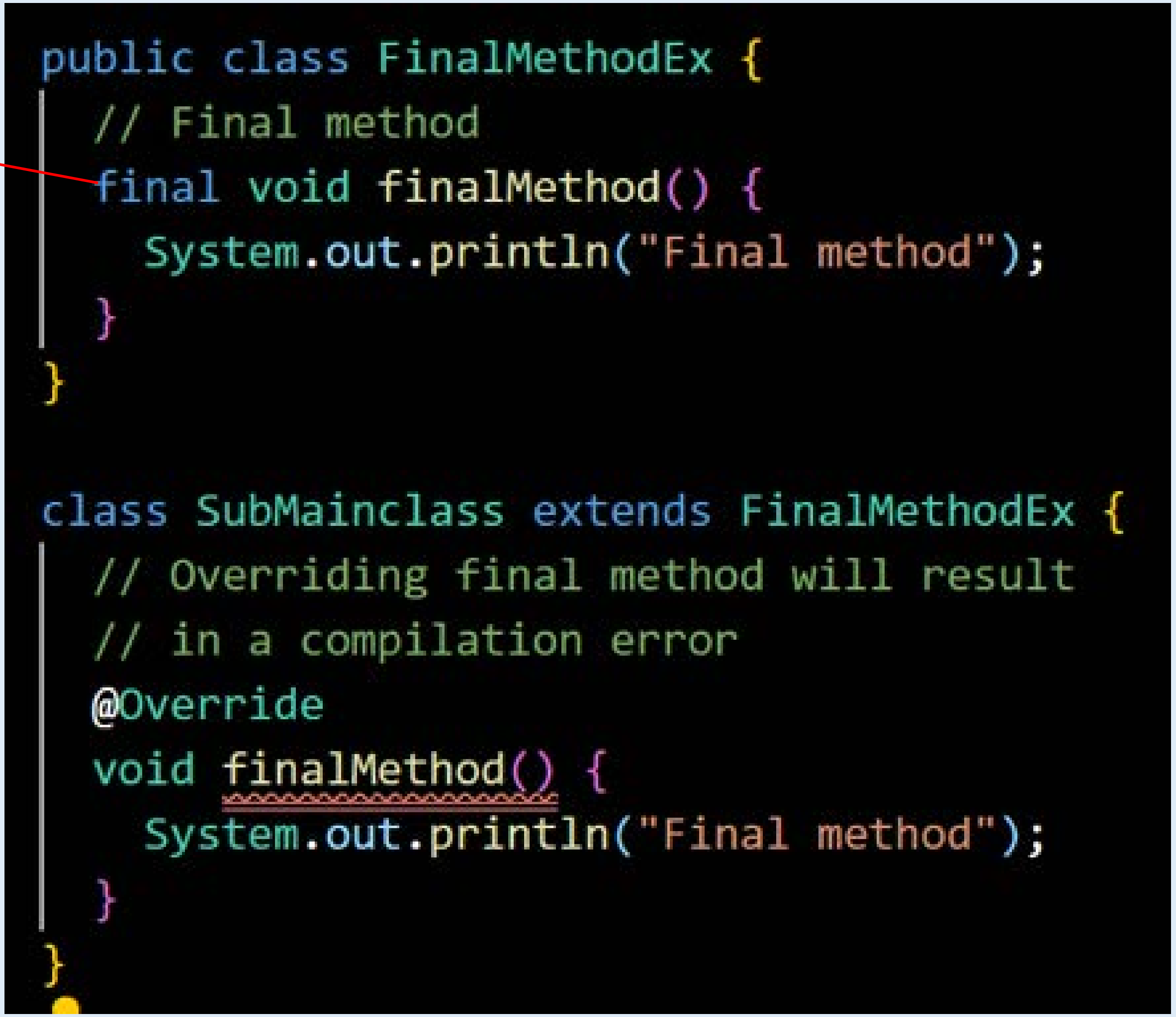


```java
// Final class
final class FinalClass {

    void method() {
        System.out.println("Method in final class");
    }
}

// Subclass attempting to extend a final class
// (will result in a compilation error)
class Subclass extends FinalClass {

}
```

back to chapter index

# Q. What is Final method in Java?

- A final method is a method that cannot be overridden by subclasses.



```
public class FinalMethodEx {
  // Final method
  final void finalMethod() {
    System.out.println("Final method");
  }
}

class SubMainclass extends FinalMethodEx {
  // Overriding final method will result
  // in a compilation error
  @Override
  void finalMethod() {
    System.out.println("Final method");
  }
}
```

back to chapter index

# Q. What is the role of final keyword? V. IMP.

- final keyword is used for creating:

**1. Final Class:** Final class cannot be subclassed or extended.

**2. Constants:** final variables cannot be changed once initialized.

**3. Final Method:** Final method can’t be overridden.

```java
// 1. Final class
final class FinalKeywordEx {

  // 2. Constant variables
  final int CONSTANT_INTEGER = 10;
  final String CONSTANT_STRING = "Hello";

  // 3. Final method
  final void finalMethod() {
    System.out.println("Final method");
  }
}
```

back to chapter index

# 25. Types of Classes - Static class & Enum

Back to main index

# Q. What is the role of Static method? V. IMP.

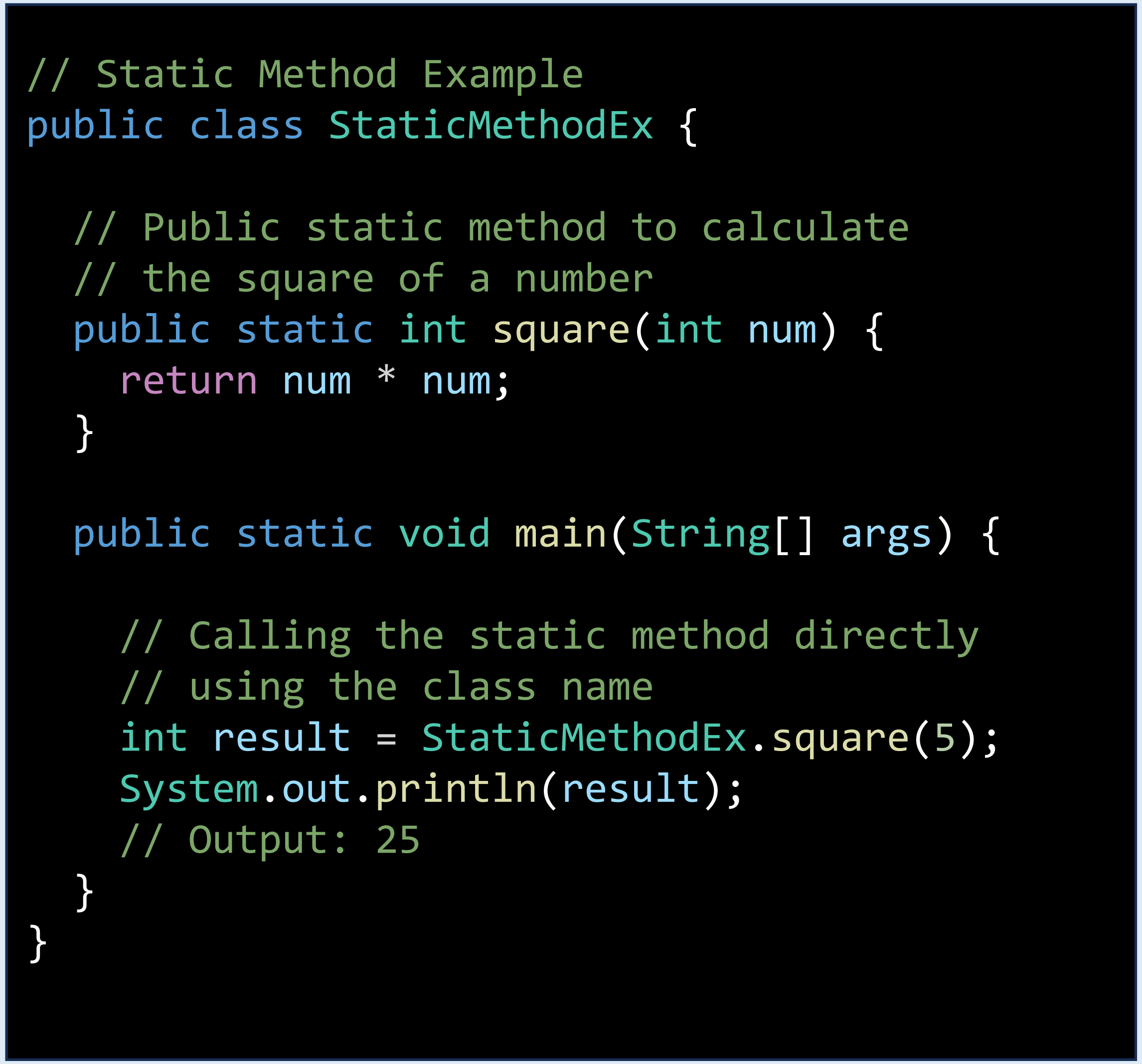
- A static keyword is used to define static methods and classes.
- A static method can be called directly on the class without needing an instance of the class.

```
// Static Method Example
public class StaticMethodEx {

  // Public static method to calculate
  // the square of a number
  public static int square(int num) {
    return num * num;
  }

  public static void main(String[] args) {

    // Calling the static method directly
    // using the class name
    int result = StaticMethodEx.square(5);
    System.out.println(result);
    // Output: 25
  }
}
```

back to chapter index

# Q. When to use static methods in real applications?

- Static methods are mostly used for defining **utility functions, and helper methods**. This is good for performance also, because then overhead of creating an instance is not required.
- Static methods are also used to implement the **Singleton design pattern**, ensuring only one instance of a class exists.

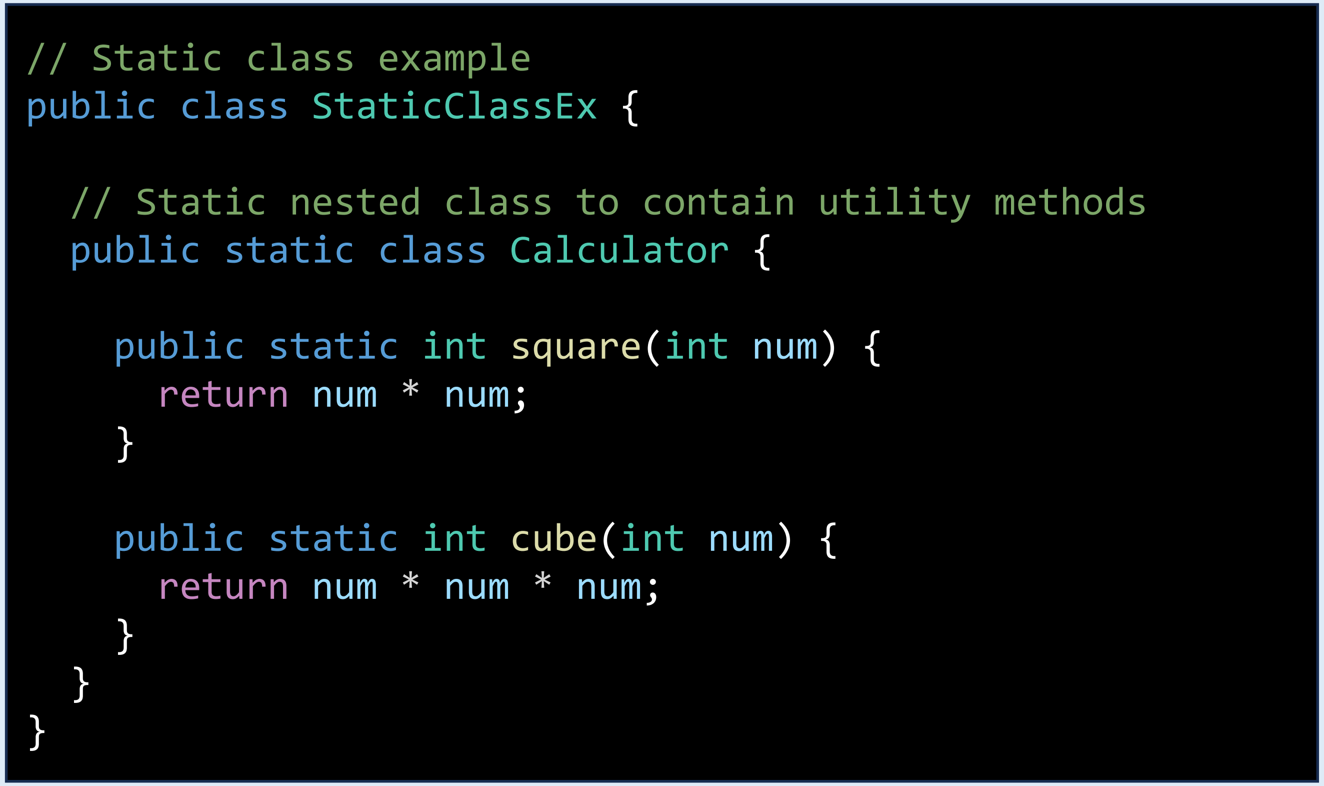


```java
// Static Method Example
public class StaticMethodEx {

  // Public static method to calculate
  // the square of a number
  public static int square(int num) {
    return num * num;
  }

  public static void main(String[] args) {

    // Calling the static method directly
    // using the class name
    int result = StaticMethodEx.square(5);
    System.out.println(result);
    // Output: 25
  }
}
```

back to chapter index

# Q. What is Static nested class?

- A static nested class in Java refers to a **nested class** declared with the static modifier inside another class.
- Use: Static class can be used to group related static (utility) functions.



```java
// Static class example
public class StaticClassEx {

  // Static nested class to contain utility methods
  public static class Calculator {

    public static int square(int num) {
      return num * num;
    }

    public static int cube(int num) {
      return num * num * num;
    }
  }
}
```

back to chapter index

# Q. What is the role of static keyword in Java? V. IMP.

- A static keyword is used to define static variables which can be accessed directly using the class name .
- A static variable's value, once assigned, will persist in memory throughout the application's lifetime, until the application ends.

- A static keyword is used to define static methods.

```java
public class StaticKeywordEx {

  // Static variable
  public static int count = 0;

  // Static methods
  public static int square(int num) {
    return num * num;
  }

  public static void main(String[] args) {

    System.out.println(StaticKeywordEx.count);
    // Output: 0

    int result = StaticKeywordEx.square(5);
    System.out.println(result);
    // Output: 25
  }
}
```

back to chapter index

# Q. What is Enum? What is the use of it in real applications?

- An enum in Java is a special data type that represents a fixed set of constants.
- Enums are special type of classes.
- Use: Enums are used for organizing the code in a better way, making code more readable.

```java
// Define an enum
public enum Priorities {
   LOW,
   MEDIUM,
   HIGH,
}
```

```java
// Define an enum
public enum Weekdays {
   SUNDAY,
   MONDAY,
   TUESDAY,
   WEDNESDAY,
   THURSDAY,
   FRIDAY,
   SATURDAY,
}
```

```java
public class Priorities {
   // Define constants for each priority
   public static final Priorities LOW = new Priorities("LOW");
   public static final Priorities MEDIUM = new Priorities("MEDIUM");
   public static final Priorities HIGH = new Priorities("HIGH");
   // Mode code.......

}
```

back to chapter index

# Q. How to use and implement Enums?

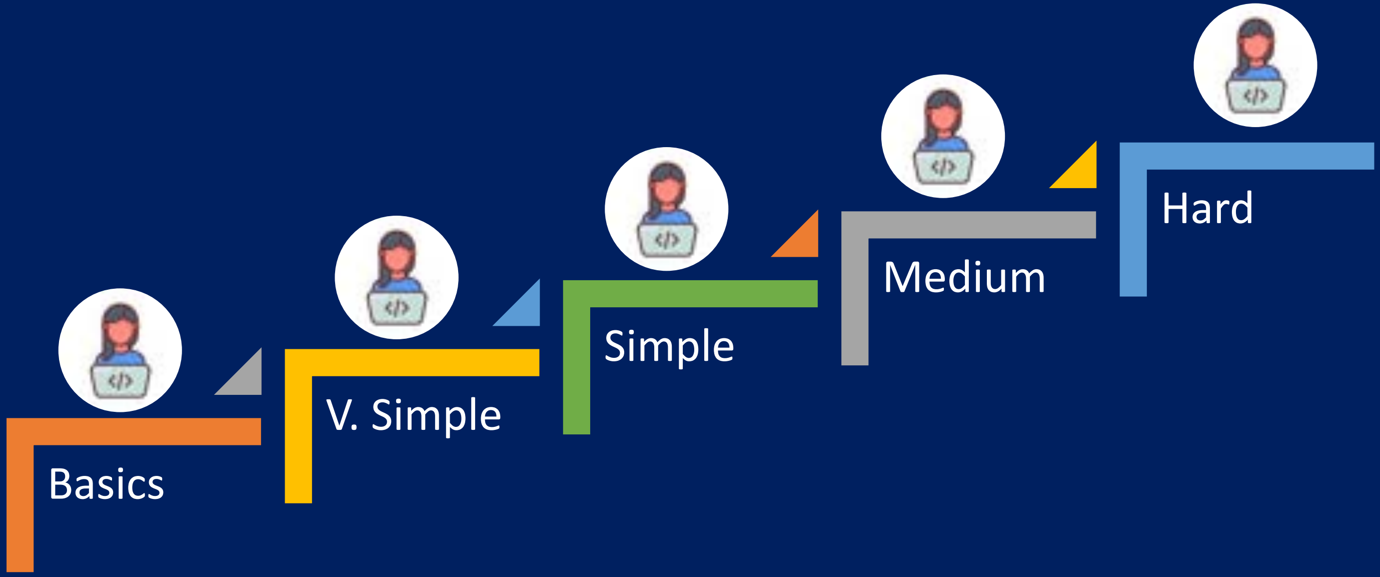
- Enums are often used with switch statements.

```java
// Define an enum for prioritites
public enum Priorities {
  LOW,
  MEDIUM,
  HIGH,
}
```

```java
public class EnumEx {
  // Method that takes an enum value as an argument
  public void printActivity(Priorities priority) {
    switch (priority) {
      case LOW:
        System.out.println("Resting");
        break;
      case MEDIUM:
        System.out.println("Learning");
        break;
      case HIGH:
        System.out.println("Working");
        break;
      default:
        System.out.println("Enjoying");
        break;
    }
  }

  public static void main(String[] args) {
    Priorities now = Priorities.MEDIUM; // enum constants
    EnumEx print = new EnumEx();
    print.printActivity(now); // Output: Learning
  }
}
```

back to chapter index

# Java Coding Problems



Basics

V. Simple

Simple

Medium

Hard

**Chapters**

Array Problems

String Problems

Number Problems

Back to main index

# Array Coding Problems

Q. What are the must-know pre-requisites for solving basic coding problems? **V. IMP.**

Q. What is the common approach for solving coding problems? **V. IMP.**

Q. Write a function to calculate the sum of all elements in an array.

Q. Write a function to calculate the average of an array of numbers.

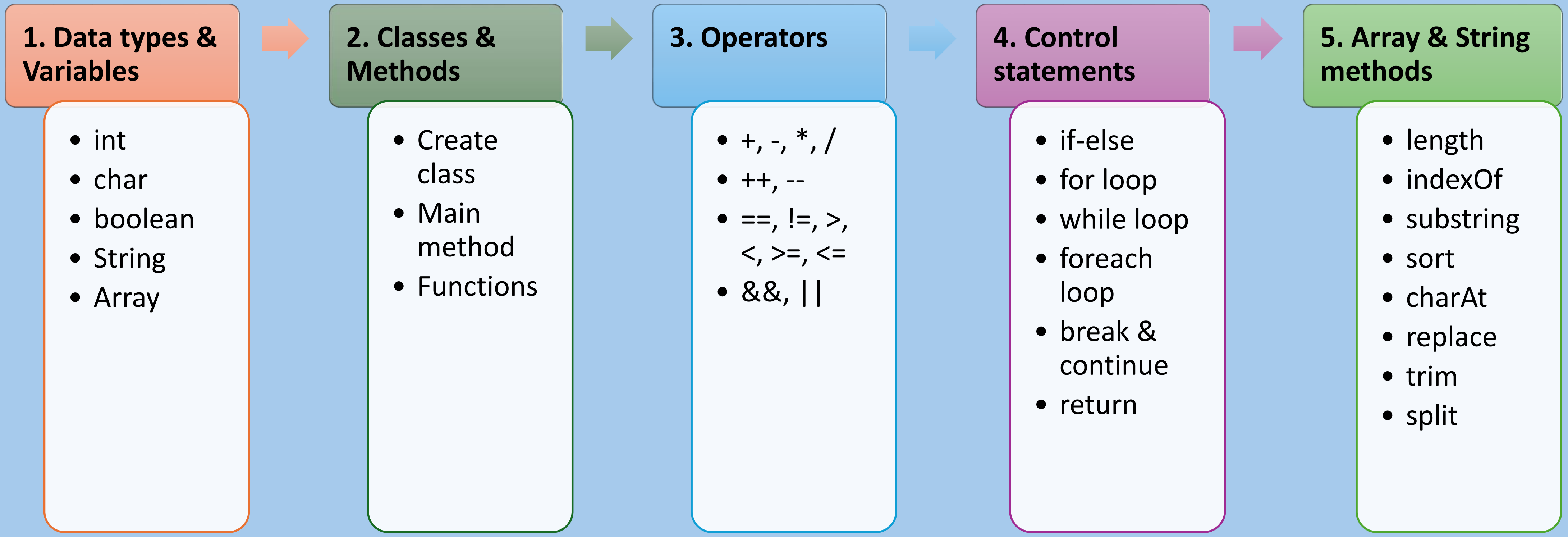
Q. Write a function to find the smallest number in an array.

Q. Write a function to find the largest number in an array.

Q. Write a function to find the second largest number in an array. **V. IMP.**

Back to main index

# Q. What are the must-know pre-requisites for solving basic coding problems? V. IMP.



**1. Data types & Variables**

- int
- char
- boolean
- String
- Array

**2. Classes & Methods**

- Create class
- Main method
- Functions

**3. Operators**

- +, -, *, /
- ++, --
- ==, !=, >, <, >=, <=
- &&, ||

**4. Control statements**

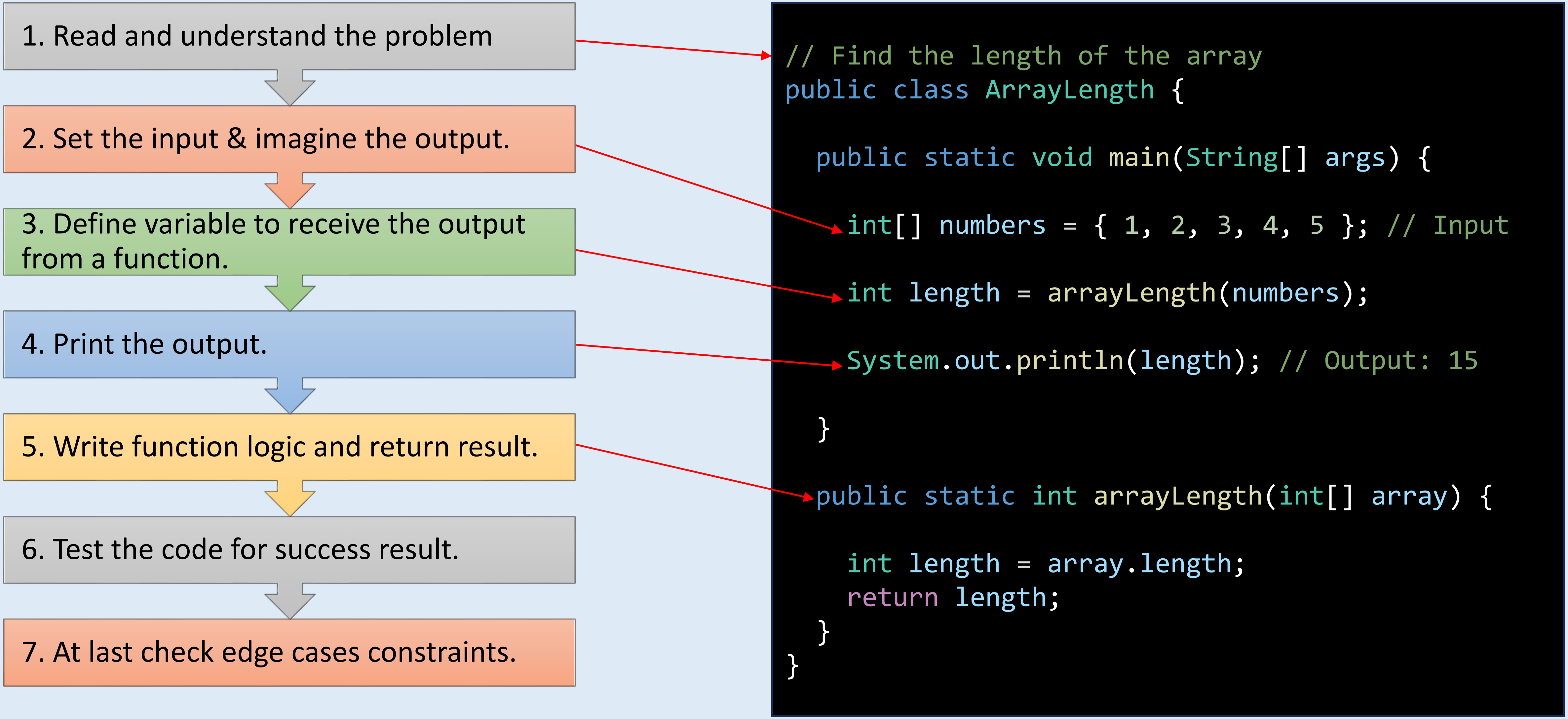
- if-else
- for loop
- while loop
- foreach loop
- break & continue
- return

**5. Array & String methods**

- length
- indexOf
- substring
- sort
- charAt
- replace
- trim
- split

back to chapter index

# Q. What is the common approach for solving coding problems? V. IMP.



1. Read and understand the problem

2. Set the input & imagine the output.

3. Define variable to receive the output from a function.

4. Print the output.

5. Write function logic and return result.

6. Test the code for success result.

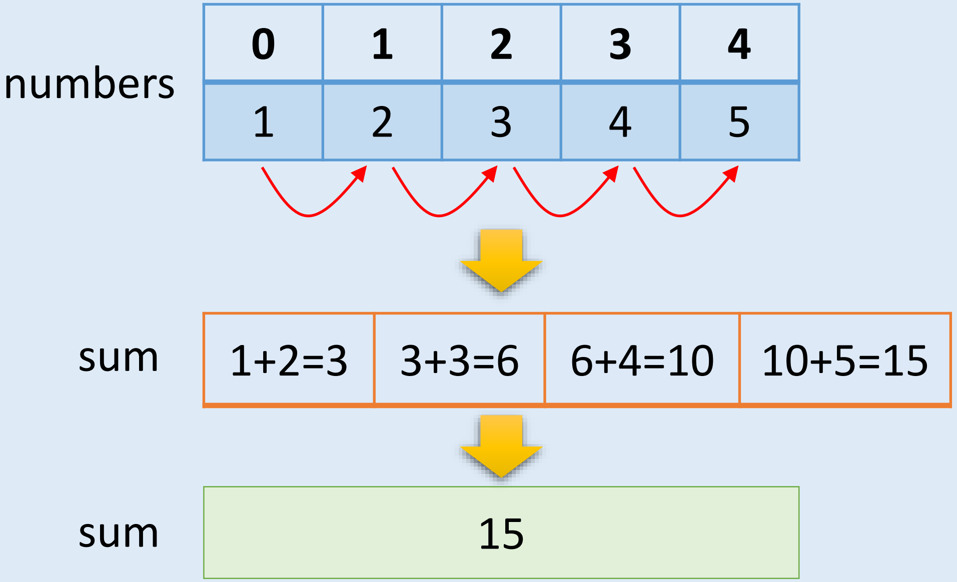
7. At last check edge cases constraints.

```
// Find the length of the array
public class ArrayLength {

  public static void main(String[] args) {

    int[] numbers = { 1, 2, 3, 4, 5 }; // Input

    int length = arrayLength(numbers);

    System.out.println(length); // Output: 15

  }

  public static int arrayLength(int[] array) {

    int length = array.length;
    return length;
  }
}
```

back to chapter index

# Q. Write a function to calculate the sum of all elements in an array.

- Tip: Break down the big problem into smaller problems and then solve them.



numbers

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 |

sum

| | | | |
|---|---|---|---|
| 1+2=3 | 3+3=6 | 6+4=10 | 10+5=15 |

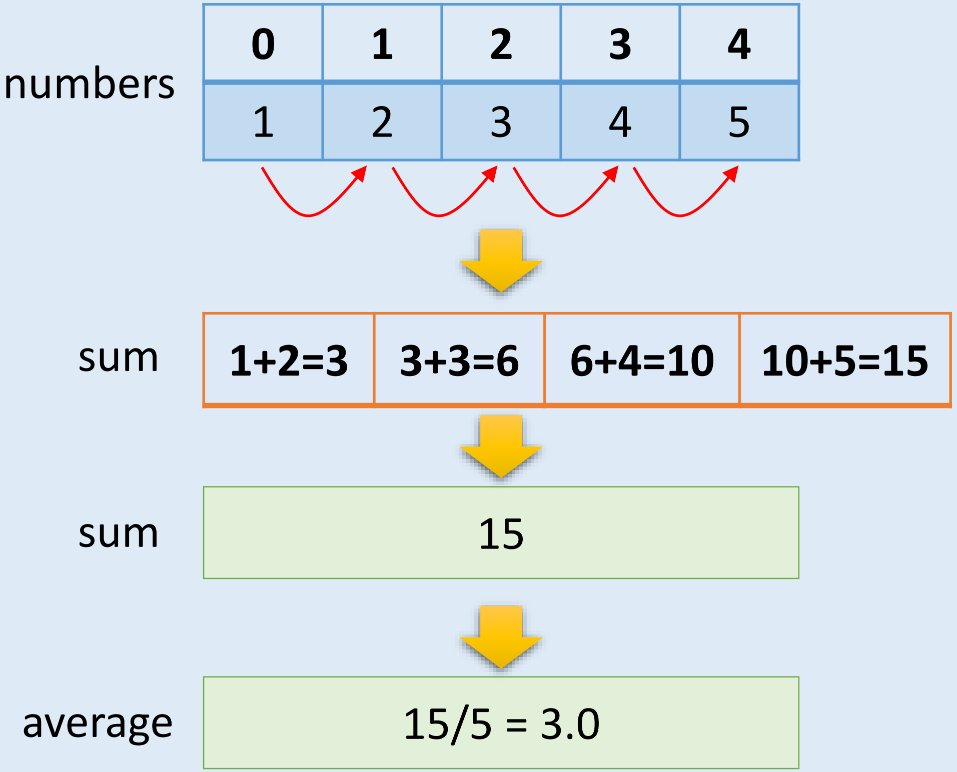
sum

15

```java
public class SumOfArrayElements {

  public static void main(String[] args) {

    int[] numbers = { 1, 2, 3, 4, 5 }; // Input

    int sum = calculateSum(numbers);

    System.out.println(sum); // Output: 15
  }

  public static int calculateSum(int[] array) {

    int sum = 0;

    for (int i = 0; i < array.length; i++) {
      sum = sum + array[i]; // Add each element to the sum
    }

    return sum;
  }
}
```

back to chapter index

# Q. Write a function to calculate the average of an array of numbers.



numbers

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 |

sum

| 1+2=3 | 3+3=6 | 6+4=10 | 10+5=15 |
|---|---|---|---|

sum

15

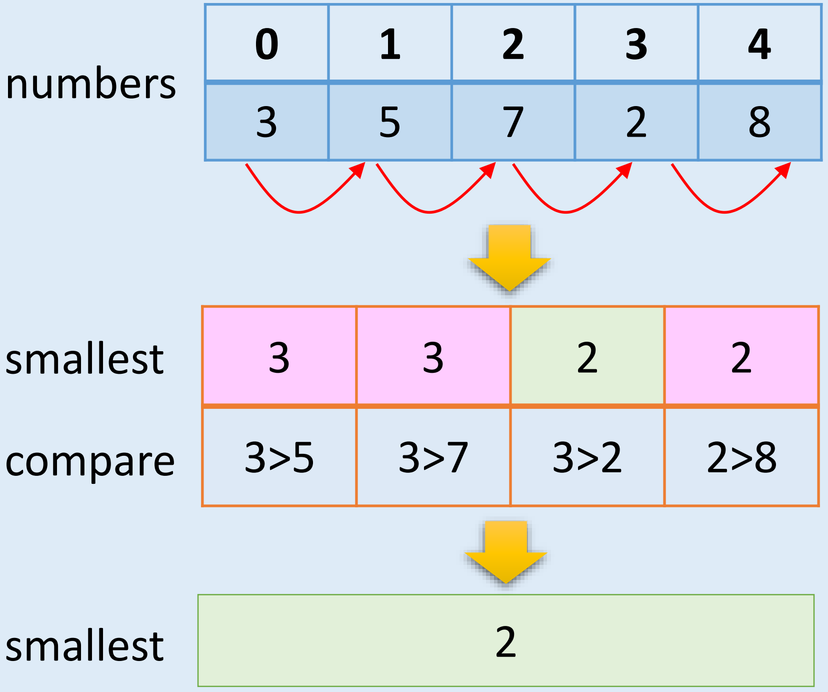
average

15/5 = 3.0

```java
public class AverageOfArrayElements {

  public static void main(String[] args) {

    int[] numbers = { 1, 2, 3, 4, 5 }; // Input

    double average = calculateAverage(numbers);

    System.out.println(average); // Output: 3.0
  }

  public static double calculateAverage(int[] array) {

    int sum = 0;

    for (int i = 0; i < array.length; i++) {
      sum = sum + array[i]; // Add each element to the sum
    }

    return (double) sum / array.length;
  }
}
```

back to chapter index

## Q. Write a function to find the smallest number in an array.



| numbers | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| | 3 | 5 | 7 | 2 | 8 |

| | | | | |
|---|---|---|---|---|
| smallest | 3 | 3 | 2 | 2 |
| compare | 3>5 | 3>7 | 3>2 | 2>8 |

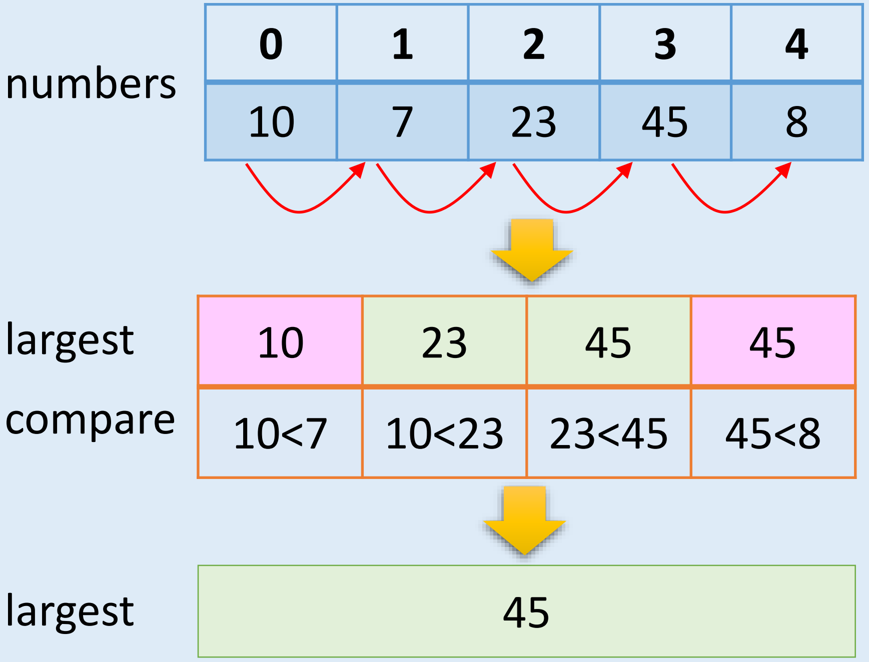
| | |
|---|---|
| smallest | 2 |

```java
public class SmallestNumberFinder {

  public static void main(String[] args) {
    int[] numbers = { 3, 5, 7, 2, 8 };
    int smallest = findSmallestNumber(numbers);
    System.out.println(smallest); // Output: 2
  }

  public static int findSmallestNumber(int[] array) {
    // Assume the first element as the smallest
    int smallest = array[0]; // 3

    // Iterate through the array to find the smallest number
    for (int i = 1; i < array.length; i++) {

      // Update the smallest number if a smaller number is found
       if (smallest > array[i]) { // 3>5, 3>7, 3>2, 2>8
        smallest = array[i];
      }
    }
    return smallest;
  }
}
```

- Tip: Most of the coding problems need a **for loop** for iterating over elements and **if condition** for comparing elements.

back to chapter index

# Q. Write a function to find the largest number in an array.



| numbers | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| | 10 | 7 | 23 | 45 | 8 |

| | | | | |
|---|---|---|---|---|
| largest | 10 | 23 | 45 | 45 |
| compare | 10<7 | 10<23 | 23<45 | 45<8 |

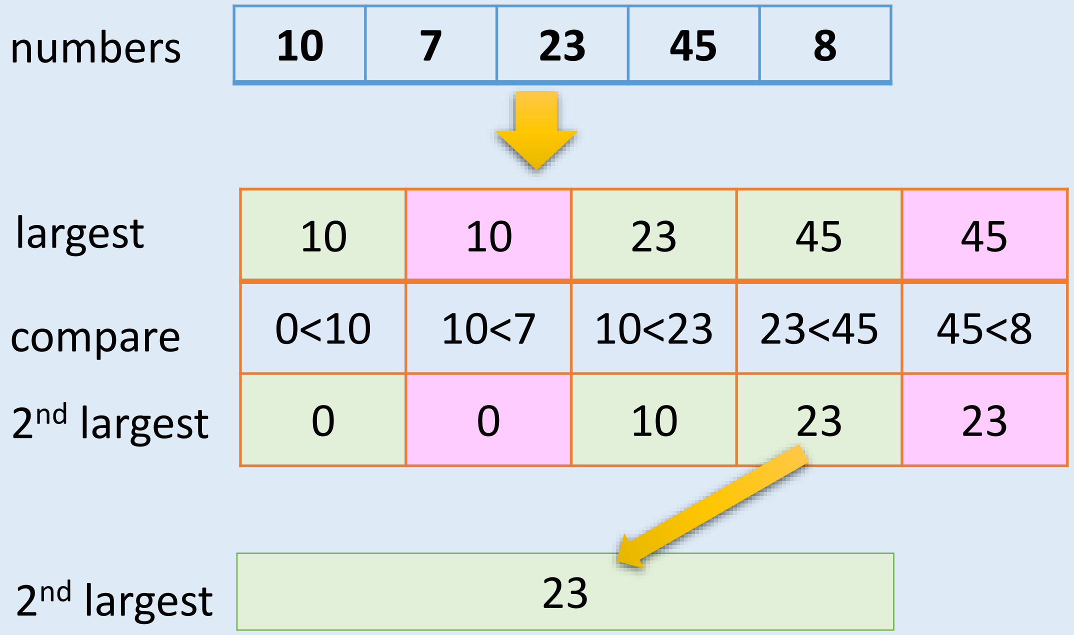
| | |
|---|---|
| largest | 45 |

- Tip: To find the result, first set result initial values with either array first value/ minimum value/ 0/ blank. Then by the logic update them with the new result values.

```java
public class LargestNumberFinder {

  public static void main(String[] args) {
    int[] numbers = { 10, 7, 23, 45, 8 }; // Input
    int largestNumber = findLargestNumber(numbers);
    System.out.println(largestNumber); // Output: 45
  }

  public static int findLargestNumber(int[] array) {
    // Assume the first element as the largest number
    int largest = array[0];

    // Iterate through the array to find the largest number
    for (int i = 1; i < array.length; i++) {

      if (largest < array[i]) {
        // Update the largest if a larger number is found
        largest = array[i];
      }
    }
    return largest;
  }
}
```

back to chapter index

# Q. Write a function to find the second largest number in an array. V. IMP.



| numbers | 10 | 7 | 23 | 45 | 8 |
|---|---|---|---|---|---|

| | | | | | |
|---|---|---|---|---|---|
| largest | 10 | 10 | 23 | 45 | 45 |
| compare | 0<10 | 10<7 | 10<23 | 23<45 | 45<8 |
| 2nd largest | 0 | 0 | 10 | 23 | 23 |

2nd largest 23

- Tip: Solve the simple case first.
- Tip: Solve the edge case or complex cases later.

```java
public class SecondLargestElement {

  public static void main(String[] args) {
    int[] array = { 10, 7, 23, 45, 8 };
    int secondLargest = findSecondLargest(array);
    System.out.println(secondLargest); // Output: 23
  }

  public static int findSecondLargest(int[] array) {
    int largest = Integer.MIN_VALUE; // OR 0
    int secondLargest = Integer.MIN_VALUE; // OR 0

    for (int i = 0; i < array.length; i++) {
      int num = array[i];
      if (largest < num) {
        secondLargest = largest;
        largest = num;
      } // { 10, 7, 23, 45, 30 }; // complex case
      else if (num > secondLargest && num != largest){
        secondLargest = num;
      }
    }
    return secondLargest;
  }
}
```

back to chapter index

# Array Coding Problems Using Functions

Q. What are the top 5 important array functions used in coding problems? **V. IMP.**

Q. Write a function to check whether two arrays are same or not?

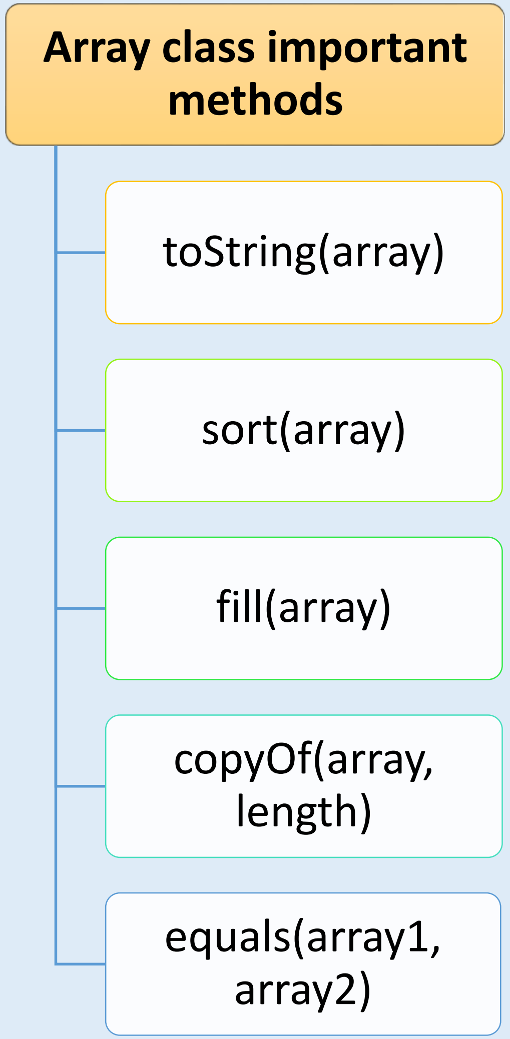
Q. Write a function to check if a given array is sorted in ascending order or not.

Q. Write a function to merge two arrays into a single sorted array. **V. IMP.**

Q. Write a function to remove a specific element from an array.

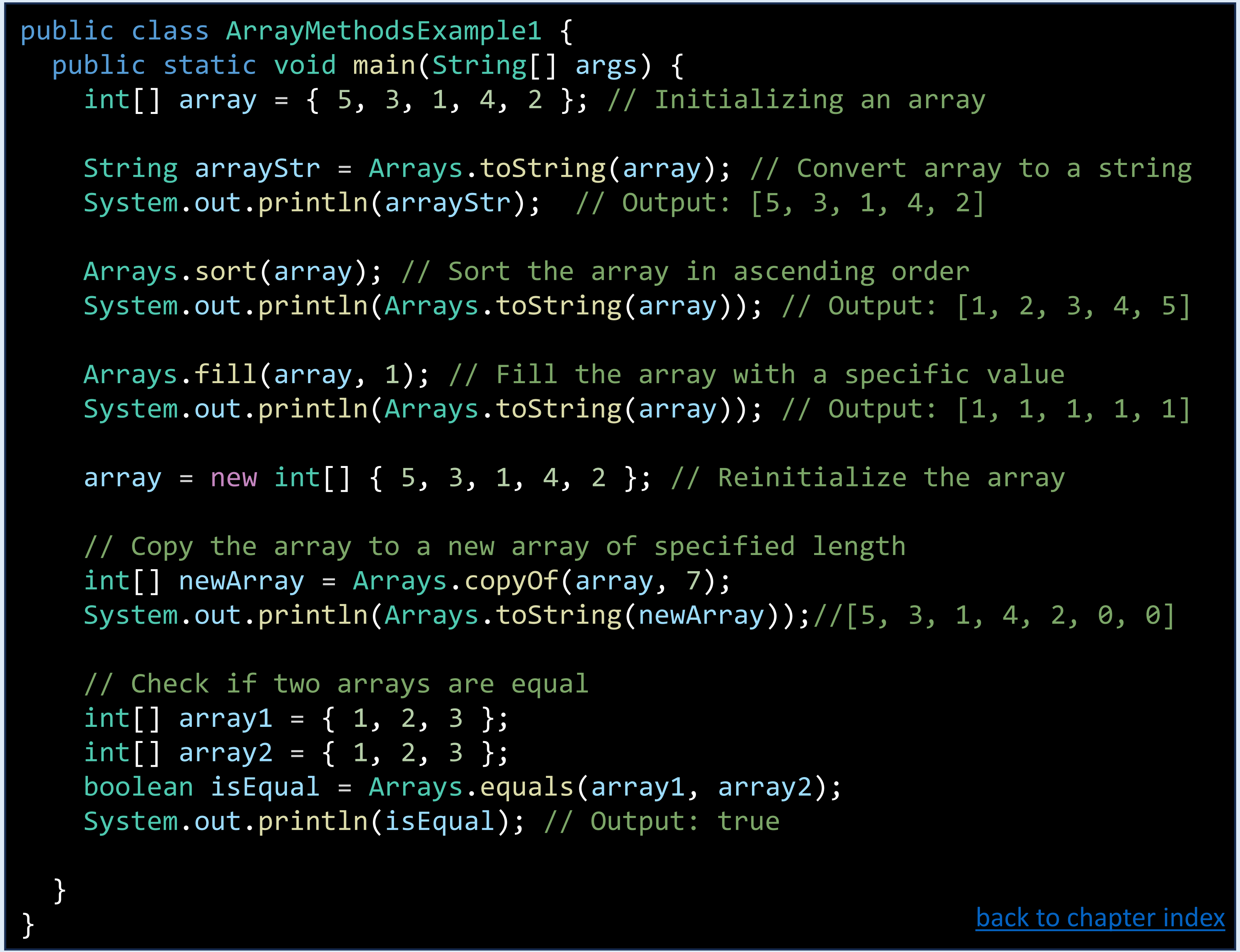
Back to main index

# Q. What are the top 5 important array functions used in coding problems? V. IMP.



Array class important methods

- toString(array)
- sort(array)
- fill(array)
- copyOf(array, length)
- equals(array1, array2)



```java
public class ArrayMethodsExample1 {
  public static void main(String[] args) {
    int[] array = { 5, 3, 1, 4, 2 }; // Initializing an array

    String arrayStr = Arrays.toString(array); // Convert array to a string
    System.out.println(arrayStr);  // Output: [5, 3, 1, 4, 2]

    Arrays.sort(array); // Sort the array in ascending order
    System.out.println(Arrays.toString(array)); // Output: [1, 2, 3, 4, 5]

    Arrays.fill(array, 1); // Fill the array with a specific value
    System.out.println(Arrays.toString(array)); // Output: [1, 1, 1, 1, 1]

    array = new int[] { 5, 3, 1, 4, 2 }; // Reinitialize the array

    // Copy the array to a new array of specified length
    int[] newArray = Arrays.copyOf(array, 7);
    System.out.println(Arrays.toString(newArray));//[5, 3, 1, 4, 2, 0, 0]

    // Check if two arrays are equal
    int[] array1 = { 1, 2, 3 };
    int[] array2 = { 1, 2, 3 };
    boolean isEqual = Arrays.equals(array1, array2);
    System.out.println(isEqual); // Output: true

  }
}
```

back to chapter index

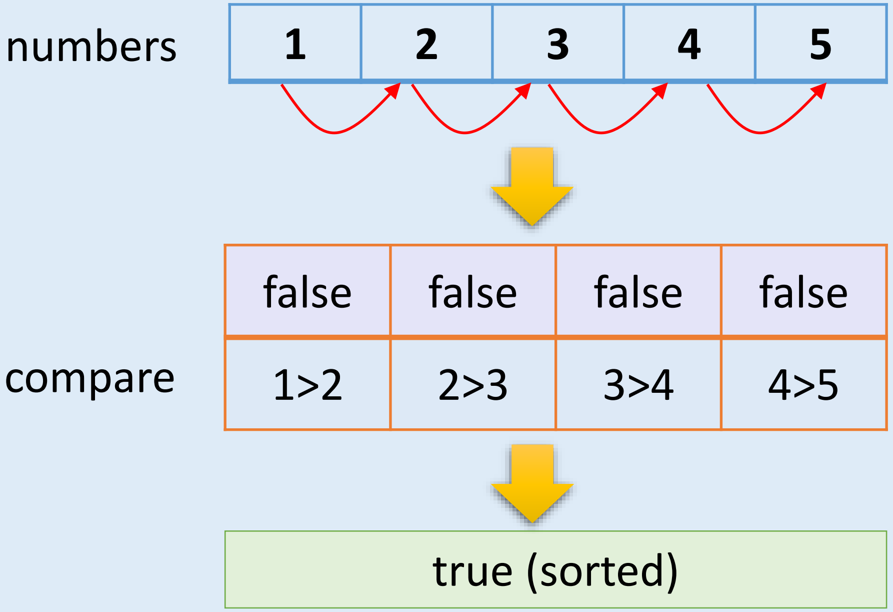
# Q. Write a function to check whether two arrays are same or not?

- Tip: Knowing important Array class methods is beneficial for solving coding problems.

```java
import java.util.Arrays;

public class CompareArrays {

  public static void main(String[] args) {
    int[] array1 = { 5, 2, 9, 1, 5, 6 }; // Input
    int[] array2 = { 5, 2, 9, 1, 5, 6 }; // Input

    System.out.println(Arrays.equals(array1, array2));
    // Output: true
  }
}
```

back to chapter index

# Q. Write a function to check if a given array is sorted in ascending order or not.



numbers

| 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|

compare

| false | false | false | false |
|---|---|---|---|
| 1>2 | 2>3 | 3>4 | 4>5 |

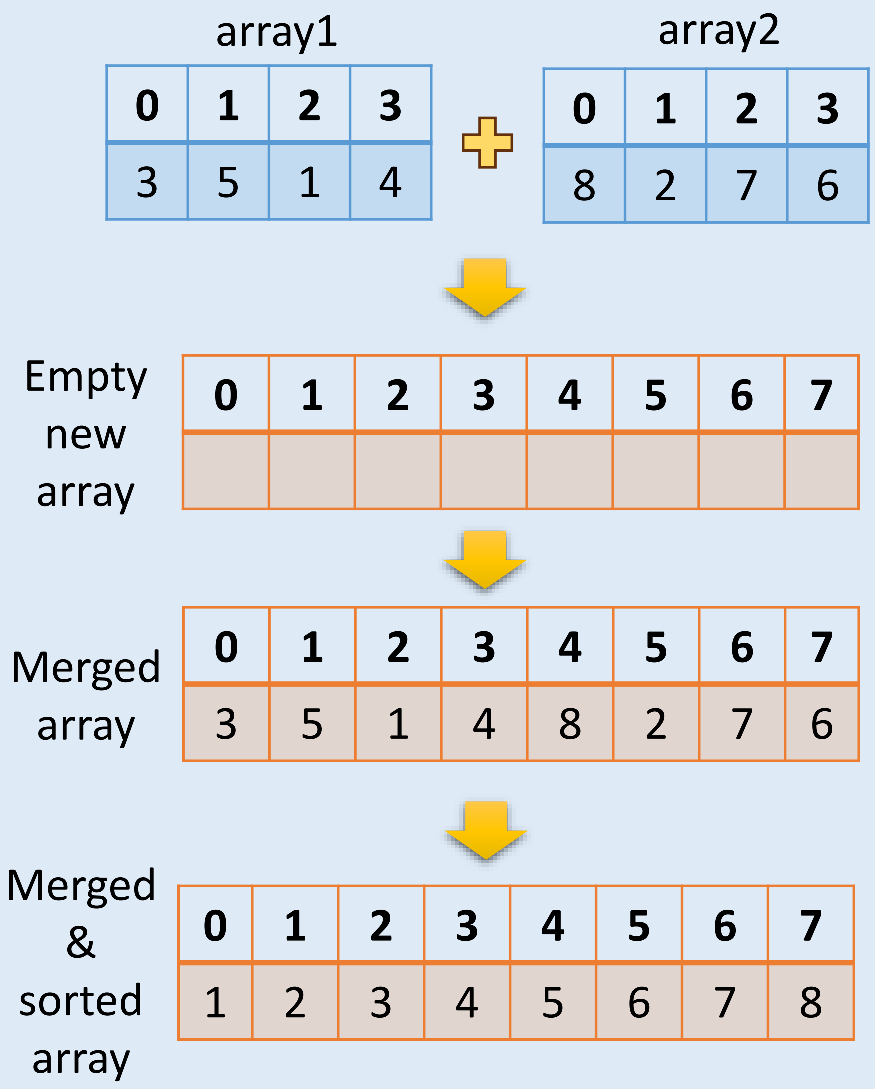
true (sorted)

- Don’t start code immediately. First read and understand the problem carefully and try to think about the input, output and the simplest logic to solve it.

```java
public class CheckIfArraySorted {

  public static void main(String[] args) {

    int[] array1 = { 1, 2, 3, 4, 5 };

    System.out.println(isSorted(array1));
    // Output: true
  }

  public static boolean isSorted(int[] array) {

    for (int i = 0; i < array.length - 1; i++) {

      if (array[i] > array[i + 1]) {
        return false;
      }
    }
    return true;
  }
}
```

back to chapter index

# Q. Write a function to merge two arrays into a single sorted array. V. IMP.



array1

| 0 | 1 | 2 | 3 |
|---|---|---|---|
| 3 | 5 | 1 | 4 |

array2

| 0 | 1 | 2 | 3 |
|---|---|---|---|
| 8 | 2 | 7 | 6 |

Empty new array

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
|  |  |  |  |  |  |  |  |

Merged array

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 3 | 5 | 1 | 4 | 8 | 2 | 7 | 6 |

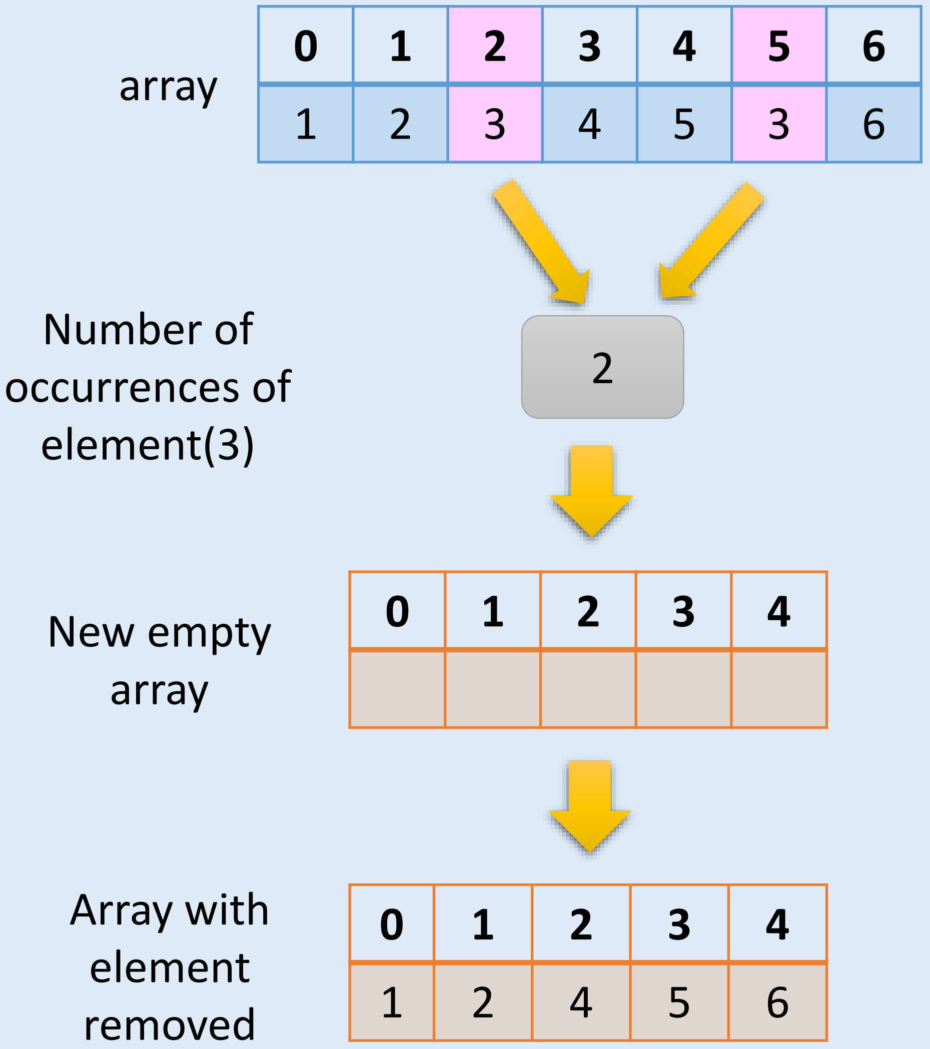
Merged & sorted array

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

```java
public class MergeAndSortArrays {
  public static void main(String[] args) {
    int[] array1 = { 3, 5, 1, 4 };
    int[] array2 = { 8, 2, 7, 6 };

    int[] mergedArray = mergeAndSortArrays(array1, array2);
    System.out.println(Arrays.toString(mergedArray));
  } // Output: [1, 2, 3, 4, 5, 6, 7, 8]

  public static int[] mergeAndSortArrays(int[] array1, int[] array2) {
    int len1 = array1.length;
    int len2 = array2.length;

    int[] mergedArray = new int[len1 + len2]; // Create a new array

    for (int i = 0; i < len1; i++) {// Copy elements from first array
      mergedArray[i] = array1[i];
    }

    for (int i = 0; i < len2; i++) {//Copy elements from second array
      mergedArray[len1 + i] = array2[i];
    }

    Arrays.sort(mergedArray); // Sort the merged array
    return mergedArray;
  }
}
```

back to chapter index

# Q. Write a function to remove a specific element from an array.



array

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 3 | 6 |

Number of occurrences of element(3)

2

New empty array

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
|  |  |  |  |  |

Array with element removed

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 2 | 4 | 5 | 6 |

```java
public class RemoveElementFromArray {
  public static void main(String[] args) {
    int[] array = { 1, 2, 3, 4, 5, 3, 6 };
    int elementToRemove = 3;
    int[] newArray = removeElement(array, elementToRemove);
    System.out.println(Arrays.toString(newArray));
  } // Output: [1, 2, 4, 5, 6]

public static int[] removeElement(int[] array, int element) {
    int count = 0;

    // Count occurrences of the element to be removed
    for (int item : array) {
      if (item == element) { count++; }
    }

    // Create a new array of the appropriate size
    int[] newArray = new int[array.length - count];

    int index = 0;
    // Copy elements except the one to be removed
    for (int item : array) {
      if (item != element) { newArray[index++] = item; }
    }
    return newArray;
  }
}
```

back to chapter index

# String Coding Problems

Q. Write a function that counts the number of characters in a string?

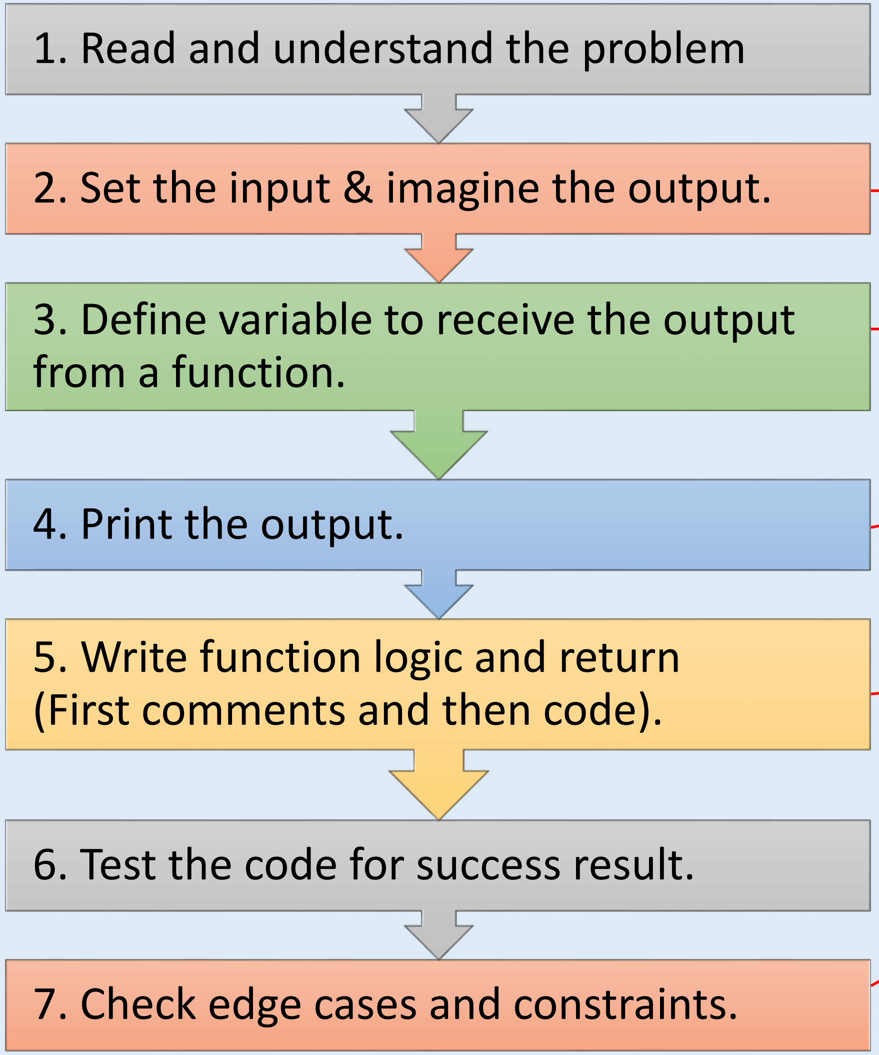
Q. How to iterate a string?

Q. Write a function that returns the reverse of a string? **V. IMP.**

Q. Write a function that checks whether a given string is a palindrome or not? **V. IMP.**

Back to main index

# Q. Write a function that counts the number of characters in a string?



1. Read and understand the problem

2. Set the input & imagine the output.

3. Define variable to receive the output from a function.

4. Print the output.

5. Write function logic and return (First comments and then code).

6. Test the code for success result.

7. Check edge cases and constraints.

```java
public class CharacterCounter {

  public static void main(String[] args) {
    // Set input
    String str = "Interview Happy";

    // Set result received by the function
    int characterCount = countCharacters(str);

    // Print result: Output: 15
    System.out.println(characterCount);
  }

  // Function
  public static int countCharacters(String input) {
    // Edge case
    if (input == null) {
      return 0; // Handle null input by returning 0
    }

    // Logic
    return input.length();
  }
}
```

back to chapter index

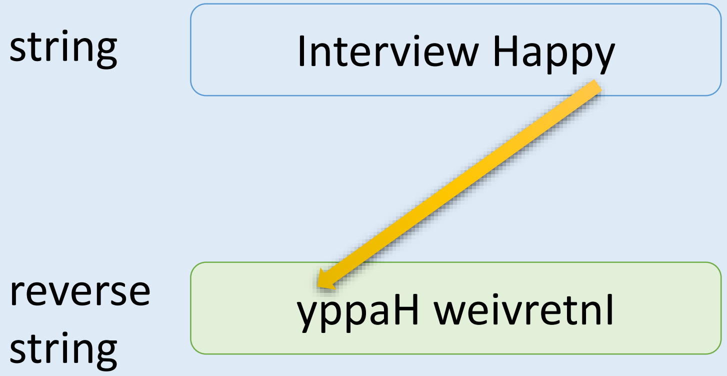
# Q. How to iterate a string?

- To iterate a string in Java, use a for loop(0 to string's length), and use charAt(i) method to access each character.

```java
public class IterateString {

  public static void main(String[] args) {
    String str = "Interview Happy";

    for (int i = 0; i < str.length(); i++) {
      char ch = str.charAt(i);
      System.out.print(ch + " ");
    }
  }
}
```

back to chapter index

# Q. Write a function that returns the reverse of a string? V. IMP.



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

    String inputStr = "Interview Happy"; // Set input

    String reversedString = reverseString(inputStr); // Get result

    System.out.println(reversedString); // Print result
  }

  public static String reverseString(String input) { // Function
    if (input == null) { // Edge case
      return null;
    }

    // Logic
    StringBuilder reversed = new StringBuilder(); // Empty string
    // Iterate all elements in reverse order
    for (int i = input.length() - 1; i >= 0; i--) {
      reversed.append(input.charAt(i)); // yppa...
    }
    return reversed.toString();
  }
} // Output: yppaH weivretnI
```

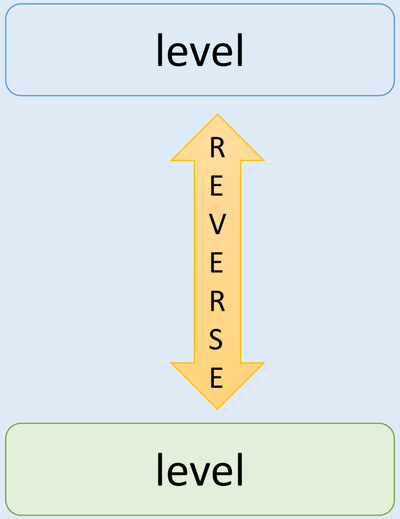
- Tip: Handle the edge cases after successfully running the program.

- Tip: Set the initial values blank rings and replace them with result later.

- Tip: Knowing basic and important string functions are very important.

back to chapter index

# Q. Write a function that checks whether a given string is a palindrome or not?



```
public class PalindromeChecker {

  public static void main(String[] args) {
    String testString = "abba"; // Set input
    boolean result = isPalindrome(testString); // Set result

    System.out.println(result); // Print result (Output: true)
  }

  public static boolean isPalindrome(String input) {
    if (input == null) { // Edge case
      return false;
    }
    StringBuilder reversed = new StringBuilder(); // Empty string
    for (int i = input.length() - 1; i >= 0; i--) { // Reverse the string
      reversed.append(input.charAt(i));
    }

    // Compare strings and return true or false
    if (input.equals(reversed.toString())) {
      return true;
    } else { return false; }
  }
}
```

- Tip: Break down the big problem into smaller problems and then solve them.

back to chapter index

# String Coding Problems Using Functions

Q. What are the important methods of String class?

Q. What are some more important methods of String class? **V. IMP.**

Q. Write a function that returns the longest word in the sentence.

Q. Write a function to remove all whitespace characters from a string.

Q. Write a function that counts the number of vowels in a string?

Q. Write a function that checks whether two strings are anagrams or not? **V. IMP.**

Back to main index

# Q. What are the important methods of String class?

```java
public class StringExample {

  public static void main(String[] args) {
    String str1 = "Interview";
    String str2 = "Happy";

    // 1.Length of the string
    int length = str1.length();
    System.out.println(length); // Output: 9

    // 2.Concatenation - joing two strings
    String result = str1.concat(", " + str2);
    System.out.println(result);
    // Output: Interview, Happy
```

```java
// 3. Substring - Retrieves substring from index 5
to 9
    String substring = result.substring(5, 9);
    System.out.println(substring); // Output: view

    // 4. Index of - // Finds index of character 'H'
    int index = result.indexOf('H');
    System.out.println(index); // Output: 11

    // 5. Equality - Checks if strings are equal
    boolean isEqual = str1.equals(str2);
    System.out.println(isEqual); // Output: false
  }
}
```

back to chapter index

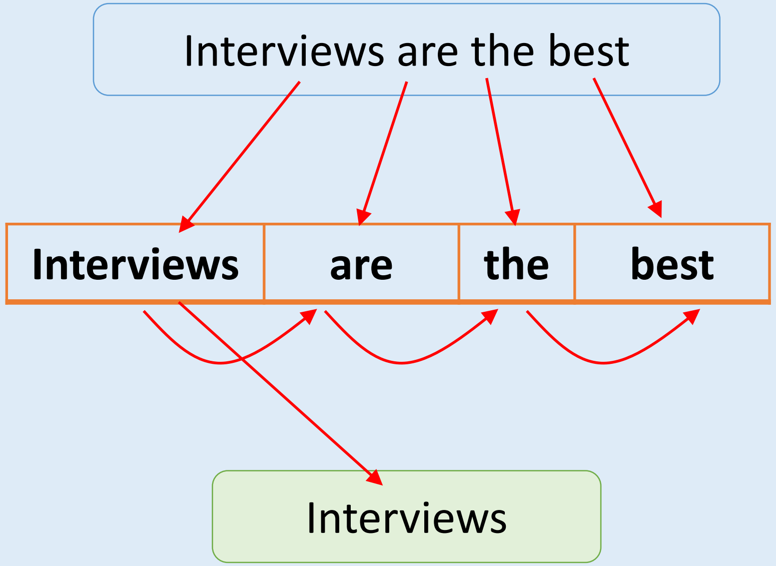
# Q. What are some more important methods of String class?

```java
public class StringMethodsExample {

  public static void main(String[] args) {
    // Example string
    String str = "   Interview Happy   ";

    // Returns the char value at the specified index
    char ch = str.charAt(7);
    System.out.println(ch); // Output: r

    // Removes leading and trailing whitespaces
    String trimmedStr = str.trim();
    System.out.println(trimmedStr);
    // Output: Interview Happy

    // Replaces occurrences of the specified target
    // sequence with the specified replacement sequence
    String replacedStr = str.replace("Happy", "Crack");
    System.out.println(replacedStr);
    // Output:    Interview Crack
```

```java
 // Splits the string into an array of substrings
    String[] parts = str.trim().split(" ");

    for (String part : parts) {
      System.out.println(part.trim());
      // Output: Interview Happy
    }

    // Converts the string to a character array
    char[] charArray = str.toCharArray();
    for (char c : charArray) {
      System.out.print(c);
     // Output:     Interview Happy
    }
  }
}
```

back to chapter index

# Q. Write a function that returns the longest word in the sentence.

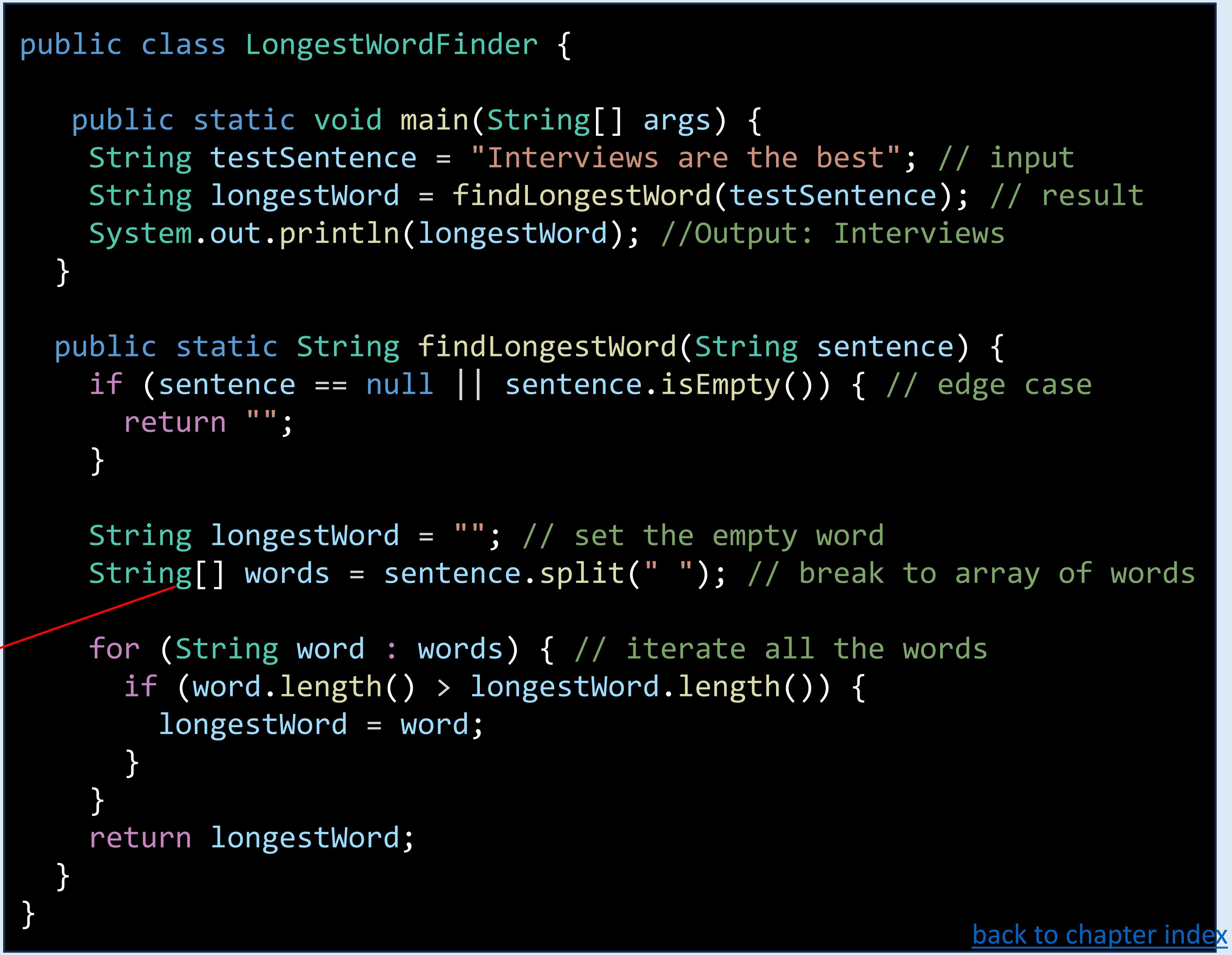


Interviews are the best

Interviews | are | the | best

Interviews

- Tip: Knowing how to convert string to array and vice versa is important.

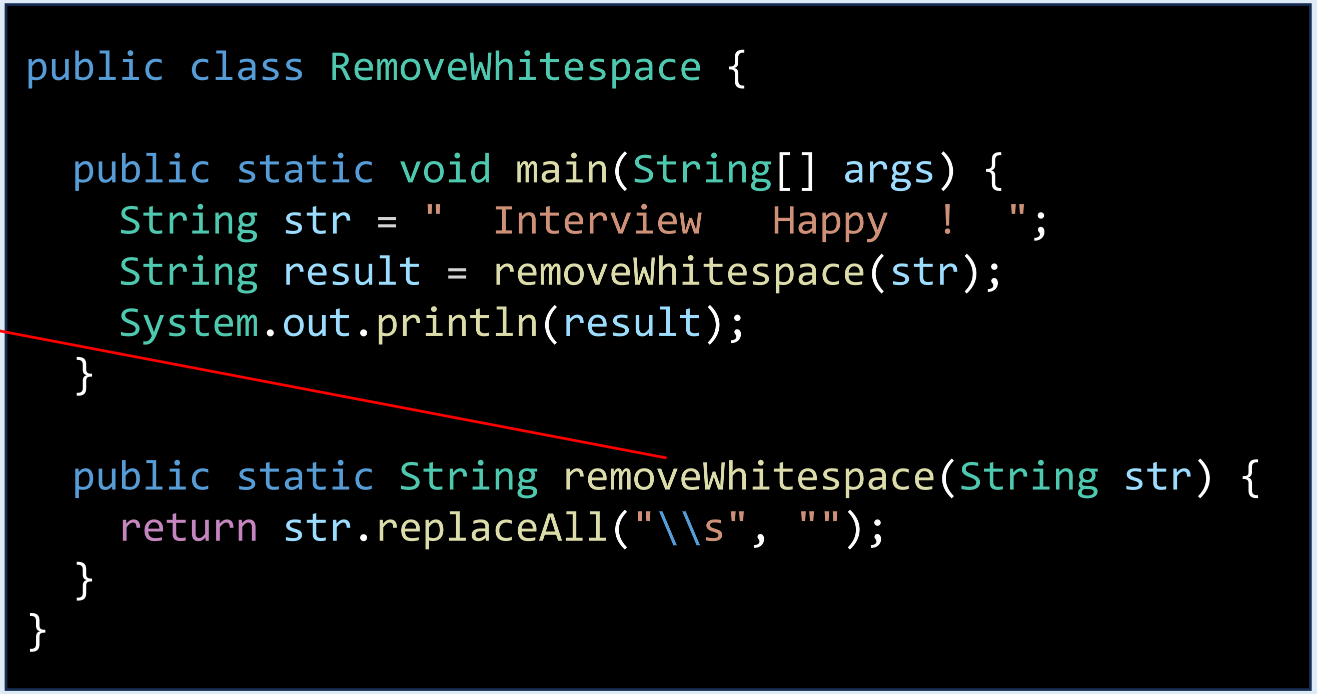


```java
public class LongestWordFinder {

   public static void main(String[] args) {
    String testSentence = "Interviews are the best"; // input
    String longestWord = findLongestWord(testSentence); // result
    System.out.println(longestWord); //Output: Interviews
  }

  public static String findLongestWord(String sentence) {
    if (sentence == null || sentence.isEmpty()) { // edge case
      return "";
    }

    String longestWord = ""; // set the empty word
    String[] words = sentence.split(" "); // break to array of words

    for (String word : words) { // iterate all the words
      if (word.length() > longestWord.length()) {
        longestWord = word;
      }
    }
    return longestWord;
  }
}
```

back to chapter index

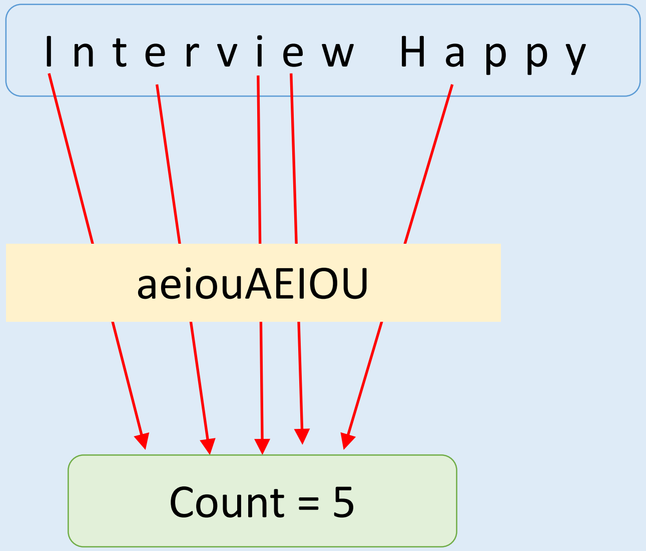
# Q. Write a function to remove all whitespace characters from a string.

- Tip: Use Meaningful Names for variables and functions.



```java
public class RemoveWhitespace {

  public static void main(String[] args) {
    String str = "  Interview   Happy  !  ";
    String result = removeWhitespace(str);
    System.out.println(result);
  }

  public static String removeWhitespace(String str) {
    return str.replaceAll("\\s", "");
  }
}
```

# Q. Write a function that counts the number of vowels in a string?



Interview Happy

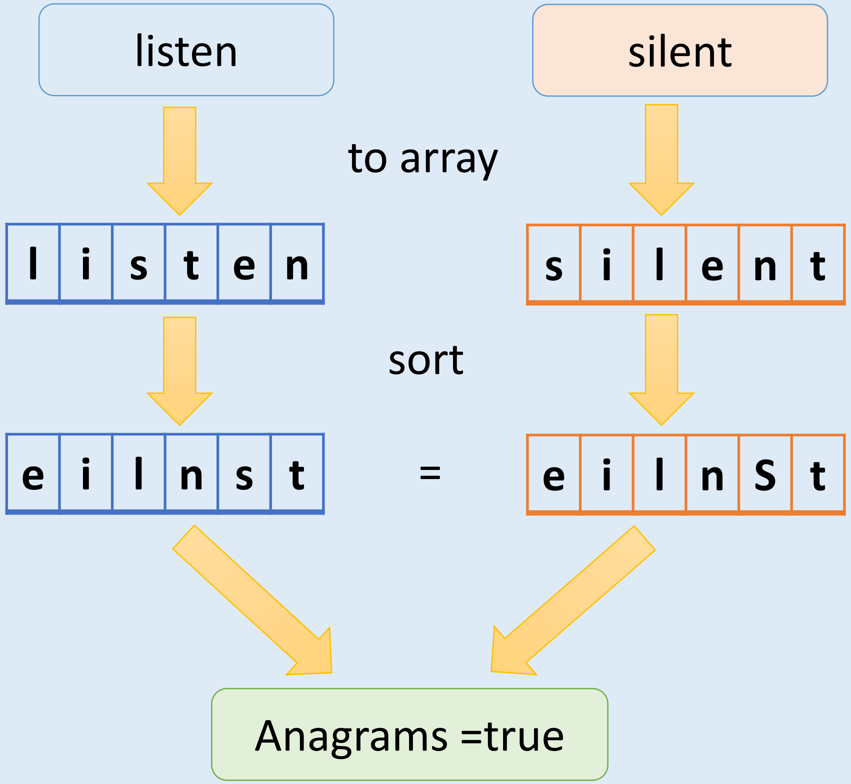
aeiouAEIOU

Count = 5

```java
public class VowelCounter {
  public static void main(String[] args) {
    String inputString = "Interview Happy"; // Input
    int numVowels = countVowels(inputString); // Result
    System.out.println(numVowels); // Output: 5
  }

  public static int countVowels(String str) {
    String vowels = "aeiouAEIOU";
    // Initialize a variable to count the number of vowels
    int vowelCount = 0;

    // Iterate through each character of the string
    for (int i = 0; i < str.length(); i++) {
      char ch = str.charAt(i);

      // Check if the character is a vowel
      if (vowels.indexOf(ch) != -1) {
        // Increment the vowel count if the character is a vowel
        vowelCount++;
      }
    }
    return vowelCount;
  }
}
```

back to chapter index

# Q. Write a function that checks whether two strings are anagrams or not? V. IMP.



listen

silent

to array

| l | i | s | t | e | n |
|---|---|---|---|---|---|

| s | i | l | e | n | t |
|---|---|---|---|---|---|

sort

| e | i | l | n | s | t |
|---|---|---|---|---|---|

=

| e | i | l | n | S | t |
|---|---|---|---|---|---|

Anagrams =true

```java
import java.util.Arrays;
public class AnagramChecker {

  public static void main(String[] args) {
    String s1 = "listen"; // Input 1
    String s2 = "silent"; // Input 2

    System.out.println(areAnagrams(s1, s2)); // Output: true
  }

  public static boolean areAnagrams(String str1, String str2) {
    // Convert strings to character arrays and sort them
    char[] charArray1 = str1.toCharArray();
    char[] charArray2 = str2.toCharArray();

    Arrays.sort(charArray1); // eilnst
    Arrays.sort(charArray2); // eilnst

    // Compare sorted character arrays
    return Arrays.equals(charArray1, charArray2);
  }
}
```

back to chapter index

# Number Coding Problems

Q. Write a function to calculate the factorial of a number. **V. IMP.**

Q. What is the difference between ++i and i++? **V. IMP.**

Q. Write a function that checks whether a number is prime or not?

Q. How to swap two numbers in Java?

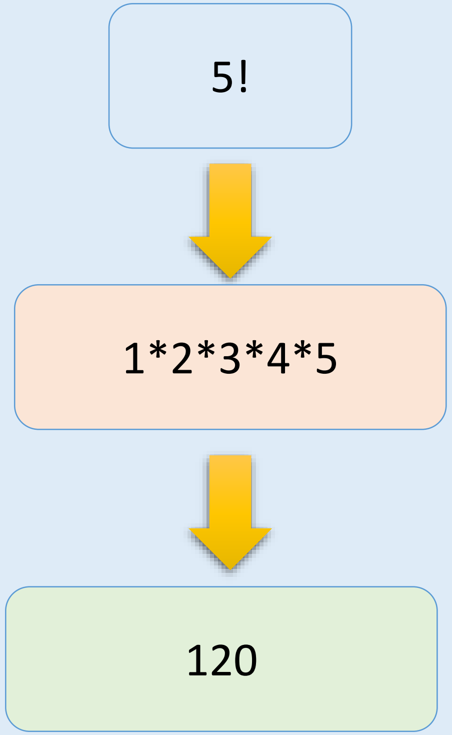
Q. Write a function to calculate the GCD for two numbers.

Q. Write a function to sum the digits of a number.

Q. Write a function to calculate the Fibonacci sequence up to a given number. **V. IMP.**

Back to main index

# Q. Write a function to calculate the factorial of a number. V. IMP.



```
public class Factorial {

  public static void main(String[] args) {
    int number = 5; // Input

    long factorial = calculateFactorial(number); // Result

    System.out.println(factorial); // output: 120
  }

  public static long calculateFactorial(int num) {

    if (num == 0) { // Edge case
      return 1;
    }

    long result = 1; // Set initial value

    for (int i = 1; i <= num; i++) {
      result = result * i;
    }
    return result;
  }
}
```

- Tip: Edge cases are handled before the normal logic of the program, because for edge cases normal logic is not applicable.

back to chapter index

# Q. What is the difference between ++i and i++? V. IMP.

- ++i (Pre-increment): it increments i first and then uses the incremented value.

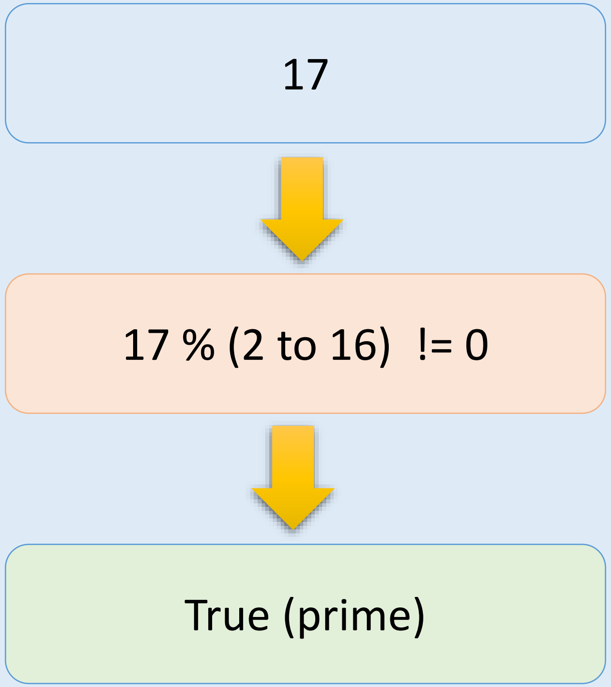
```
int i = 5;
int result = ++i; // result = 6, i = 6
```

- i++ (Post-increment): it uses the current value of i first and then increments i afterward.

```
int i = 5;
int result = i++; // result = 5, i = 6
```

back to chapter index

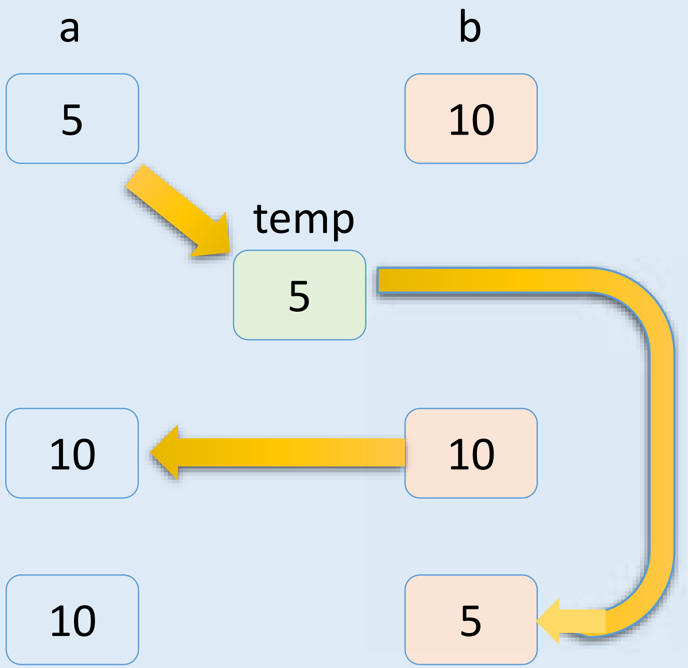
# Q. Write a function that checks whether a number is prime or not?



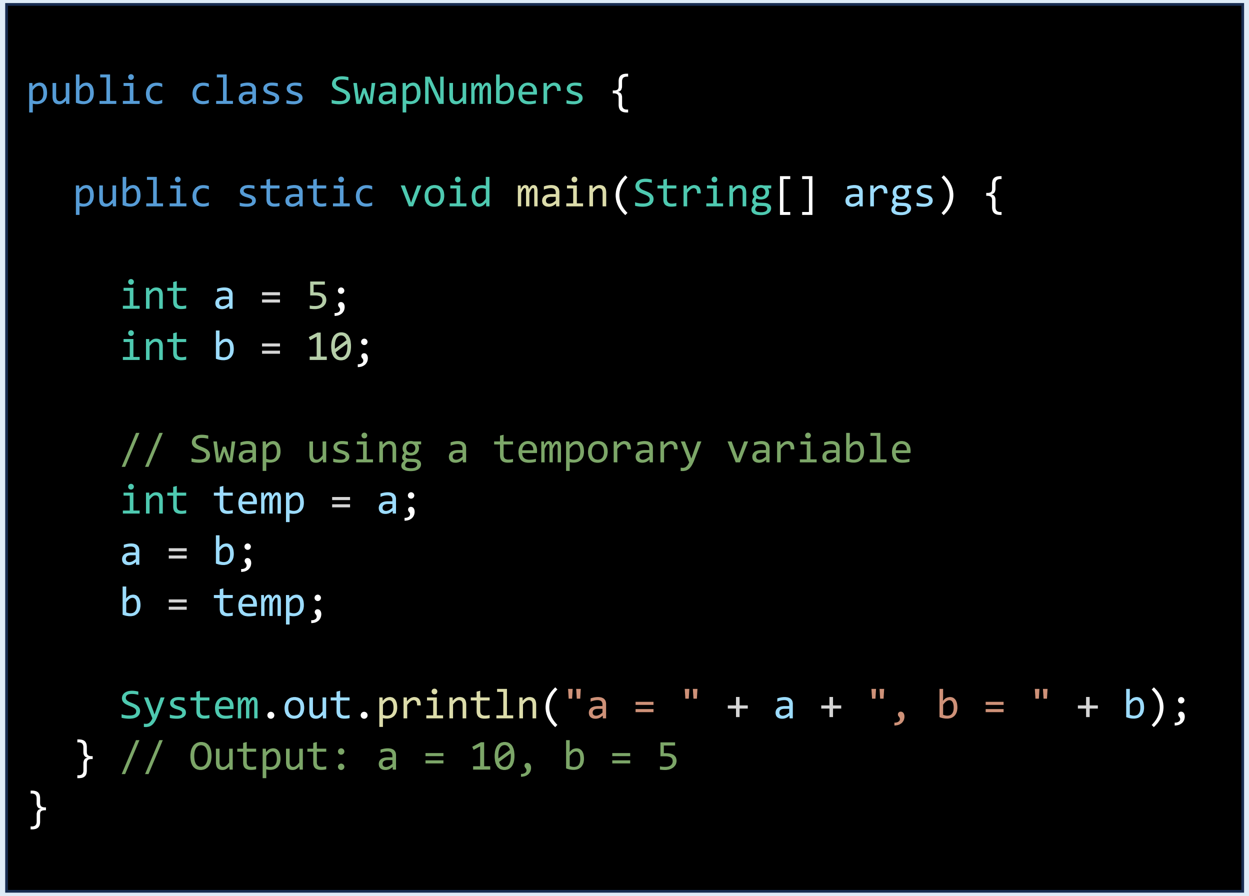
```java
public class PrimeChecker {

  public static void main(String[] args) {
    int num = 17; // input

    System.out.println(isPrime(num)); // print output
  }

  public static boolean isPrime(int number) {
    // Edge case: Check if the number is less than or equal to 1
    if (number <= 1) {
      return false; // 1 and numbers <= 1 are not prime
    }

    // Check for divisibility from 2 to square root of the number
    for (int i = 2; i <= number / 2; i++) {
      if (number % i == 0) {
        return false; // Found a divisor other than 1 and itself
      }
    }
    return true;
  }
} // Output: true
```

back to chapter index

# Q. How to swap two numbers in Java? V. IMP.



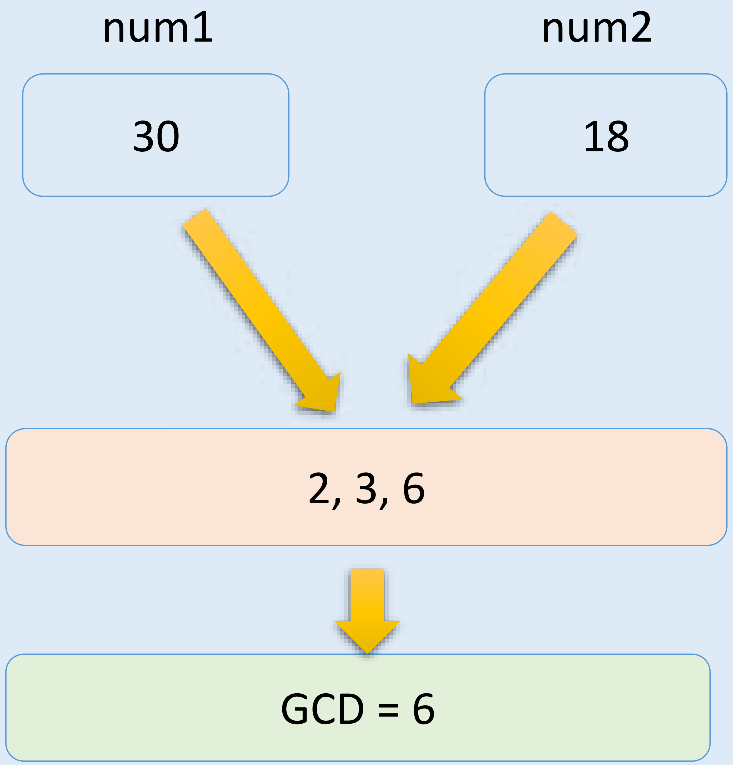
- Tip: swapping will be used in many other coding problems.

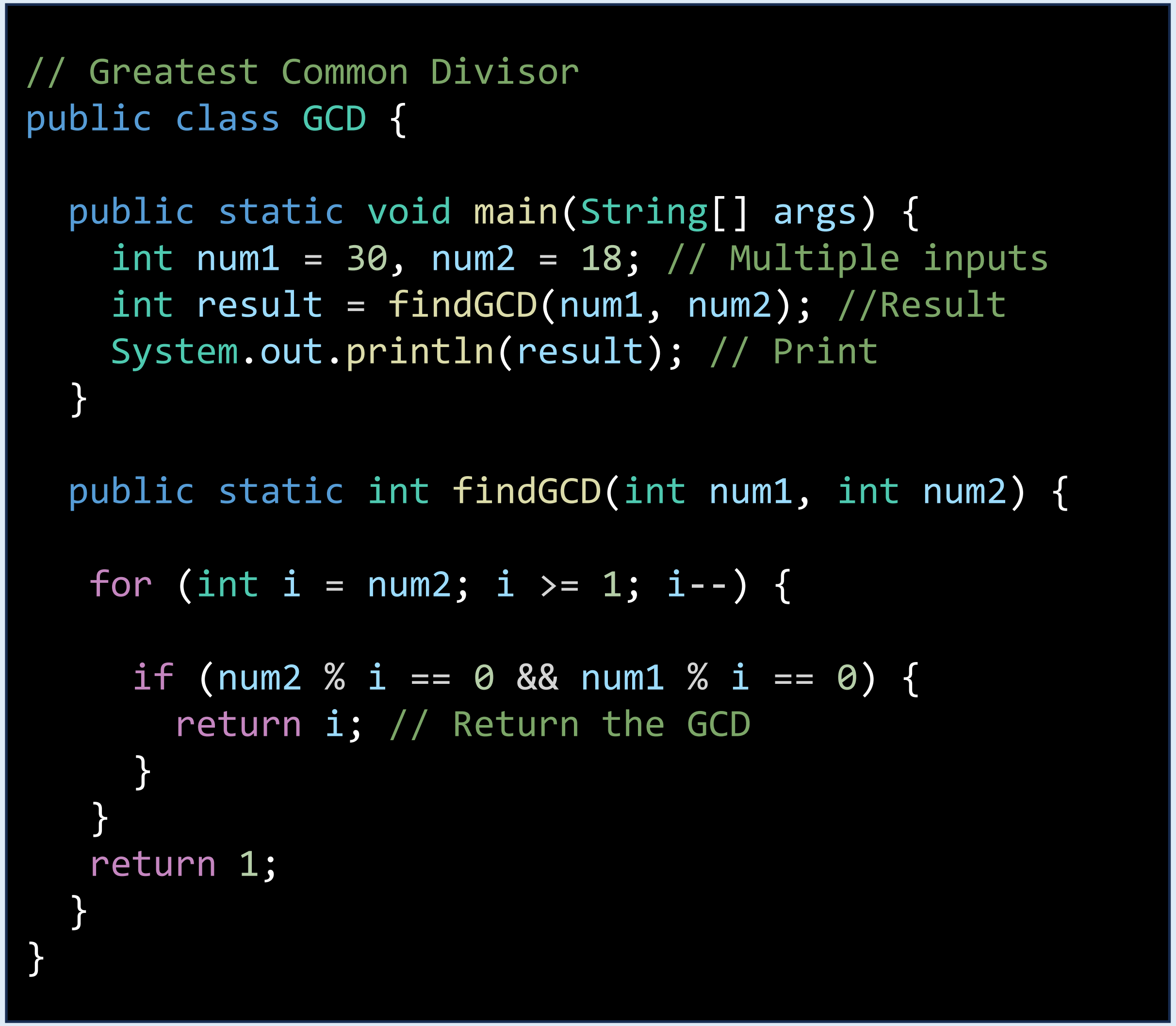


```
public class SwapNumbers {

  public static void main(String[] args) {

    int a = 5;
    int b = 10;

    // Swap using a temporary variable
    int temp = a;
    a = b;
    b = temp;

    System.out.println("a = " + a + ", b = " + b);
  } // Output: a = 10, b = 5
}
```

back to chapter index

# Q. Write a function to calculate the GCD for two numbers.

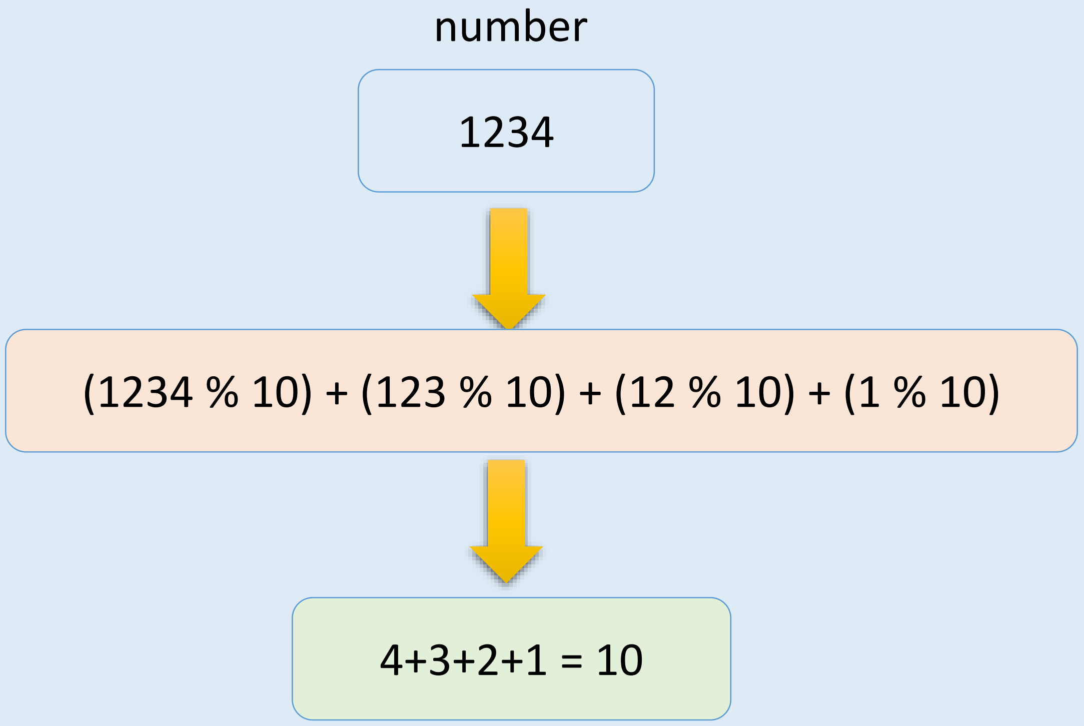




```java
// Greatest Common Divisor
public class GCD {

  public static void main(String[] args) {
    int num1 = 30, num2 = 18; // Multiple inputs
    int result = findGCD(num1, num2); //Result
    System.out.println(result); // Print
  }

  public static int findGCD(int num1, int num2) {

   for (int i = num2; i >= 1; i--) {

     if (num2 % i == 0 && num1 % i == 0) {
       return i; // Return the GCD
     }
   }
   return 1;
  }
}
```

back to chapter index

# Q. Write a function to sum the digits of a number.



number

1234

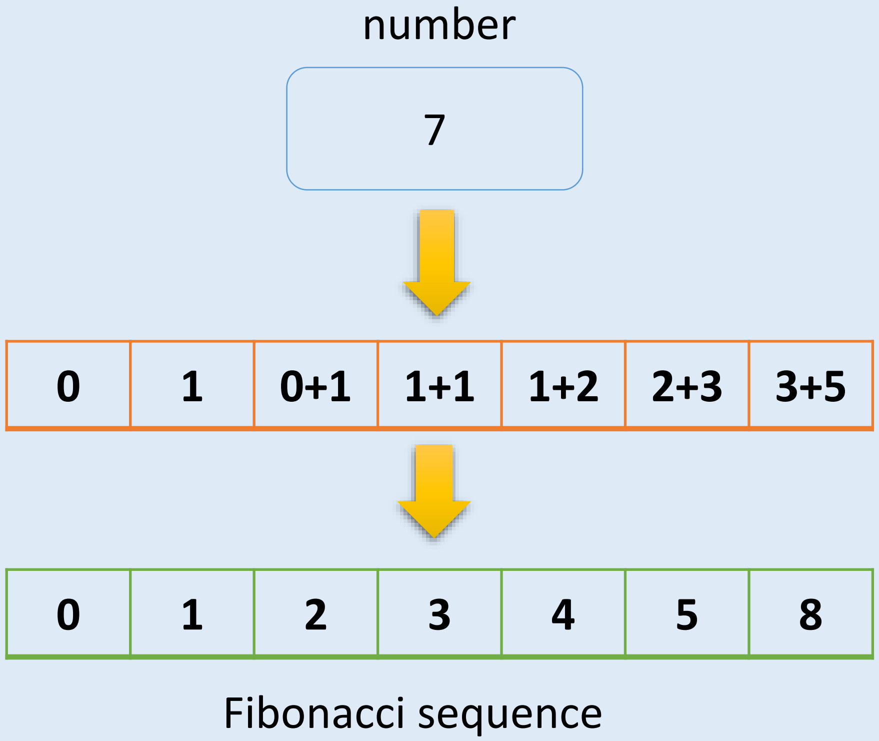
(1234 % 10) + (123 % 10) + (12 % 10) + (1 % 10)

4+3+2+1 = 10

```java
public class SumOfDigits {

  public static void main(String[] args) {
    int number = 1234;
    int sum = sumOfDigits(number);
    System.out.println(sum); // Output: 10
  }

  public static int sumOfDigits(int num) {
    int sum = 0;

    while (num != 0) {
      int reminder = num % 10; // 4, 3, 2, 1
      sum = sum + reminder; // 4, 7, 9, 10
      num = num / 10; // 123, 12, 1
    }
    return sum;
  }
}
```

back to chapter index

# Q. Write a function to calculate the Fibonacci sequence up to a given number. V. IMP.



number

7

|   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|
| 0 | 1 | 0+1 | 1+1 | 1+2 | 2+3 | 3+5 |

|   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 8 |

Fibonacci sequence

```java
public class Fibonacci {

    public static void main(String[] args) {
        int n = 7; // input
        generateFibonacci(n);
    }

    public static void generateFibonacci(int n) {

        int a = 0, b = 1;
        System.out.print(+ a + ", " + b);

        for (int i = 2; i < n; i++) {
            int next = a + b;
            System.out.print(", " + next);
            a = b;
            b = next;
        }
    } // Output: 0, 1, 1, 2, 3, 5, 8
}
```

back to chapter index

# Coding Algorithms

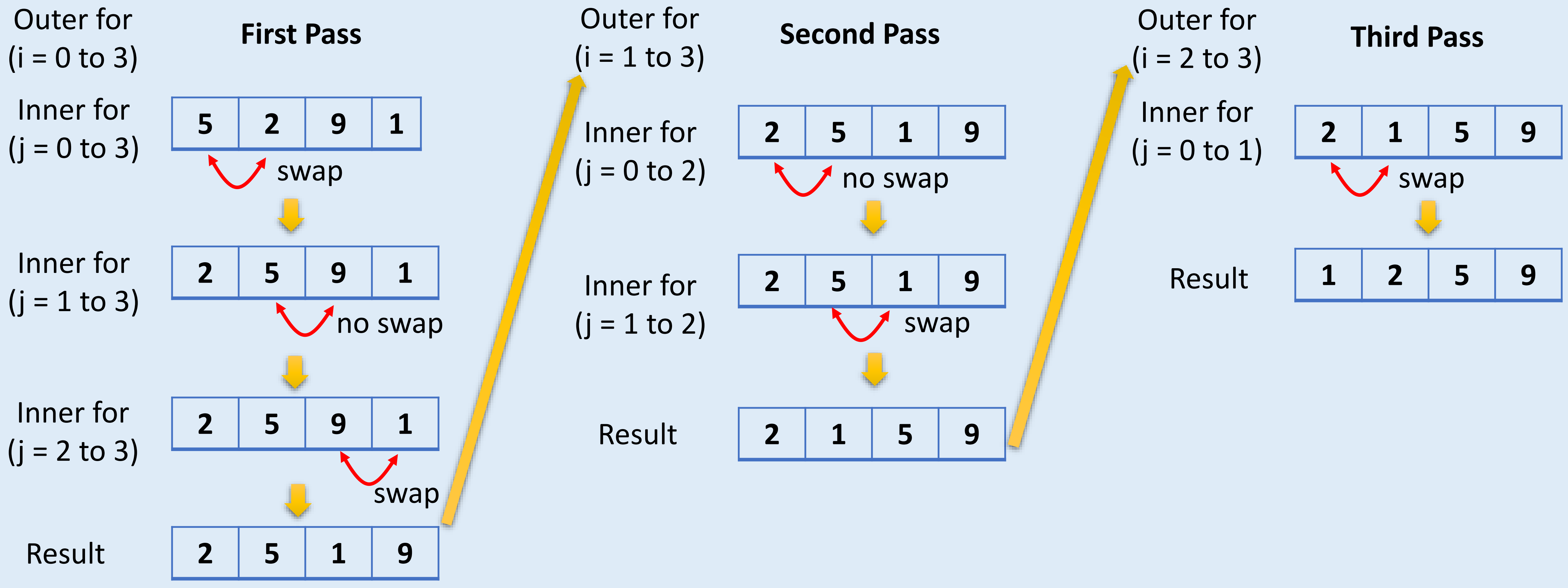
Q. Write a function to sort an array of numbers in ascending order(Bubble Sort). **V. IMP.**

Q. Write a function to search an element in any array(Binary Search)? **V. IMP.**

Back to main index

# Q. Write a function to sort an array of numbers in ascending order(Bubble Sort). V. IMP.

- In bubble sort, compare and swap adjacent elements if they are not ordered. Repeat these steps to move the largest to the end until the array is sorted.



Outer for (i = 0 to 3)

**First Pass**

Inner for (j = 0 to 3)

| 5 | 2 | 9 | 1 |
|---|---|---|---|

swap

Inner for (j = 1 to 3)

| 2 | 5 | 9 | 1 |
|---|---|---|---|

no swap

Inner for (j = 2 to 3)

| 2 | 5 | 9 | 1 |
|---|---|---|---|

swap

Result

| 2 | 5 | 1 | 9 |
|---|---|---|---|

Outer for (i = 1 to 3)

**Second Pass**

Inner for (j = 0 to 2)

| 2 | 5 | 1 | 9 |
|---|---|---|---|

no swap

Inner for (j = 1 to 2)

| 2 | 5 | 1 | 9 |
|---|---|---|---|

swap

Result

| 2 | 1 | 5 | 9 |
|---|---|---|---|

Outer for (i = 2 to 3)

**Third Pass**

Inner for (j = 0 to 1)

| 2 | 1 | 5 | 9 |
|---|---|---|---|

swap

Result

| 1 | 2 | 5 | 9 |
|---|---|---|---|

back to chapter index

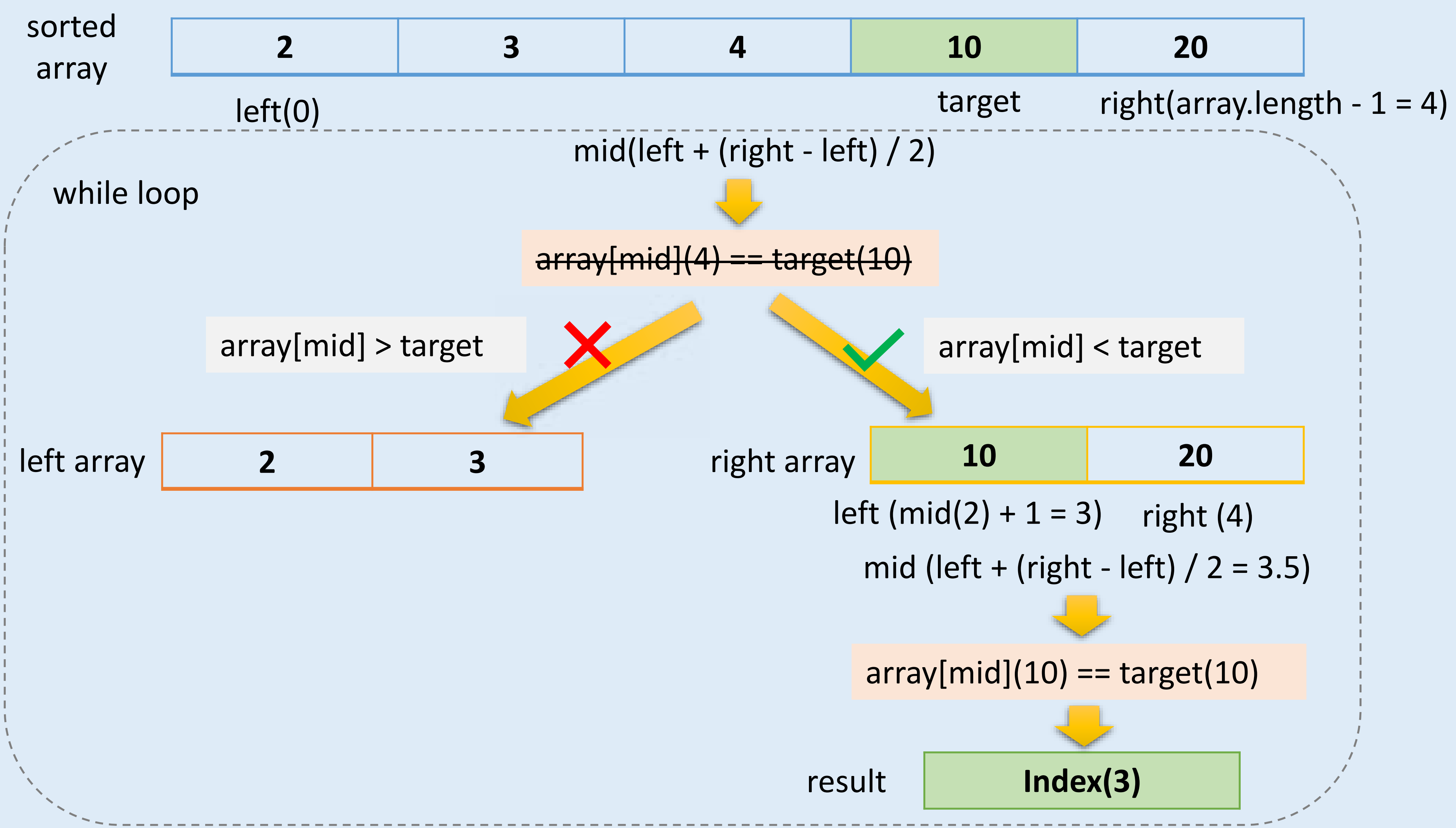
# Q. Write a function to sort an array of numbers in ascending order(Bubble Sort). V. IMP.

```java
public class SortArray {

  public static void main(String[] args) {
    int[] numbers = { 5, 2, 9, 1 };

    // Arrays.sort(numbers);

    bubbleSortArray(numbers);
    System.out.println(Arrays.toString(numbers));
    // Output: [1, 2, 5, 9]
  }
```

```java
  public static void bubbleSortArray(int[] array) {
    int n = array.length;
    boolean swapped;

    // outer for loop
    for (int i = 0; i < n - 1; i++) {
      swapped = false;
     // inner for loop
      for (int j = 0; j < n - 1 - i; j++) {
        if (array[j] > array[j + 1]) {
          // Swap array[j] and array[j + 1]
          int temp = array[j];
          array[j] = array[j + 1];
          array[j + 1] = temp;
          swapped = true;
        }
      }
      // If no two elements were swapped by inner loop, then break
      if (!swapped) break;
    }
  }
}
```

back to chapter index

# Q. Write a function to search an element in any array(Binary Search)? V. IMP.



sorted array

| 2 | 3 | 4 | 10 | 20 |
|---|---|---|---|---|

left(0)

target

right(array.length - 1 = 4)

while loop

mid(left + (right - left) / 2)

~~array[mid](4) == target(10)~~

array[mid] > target

array[mid] < target

left array

| 2 | 3 |
|---|---|

right array

| 10 | 20 |
|---|---|

left (mid(2) + 1 = 3)

right (4)

mid (left + (right - left) / 2 = 3.5)

array[mid](10) == target(10)

result

**Index(3)**

back to chapter index

# Q. Write a function to search an element in any array(Binary Search)? V. IMP.

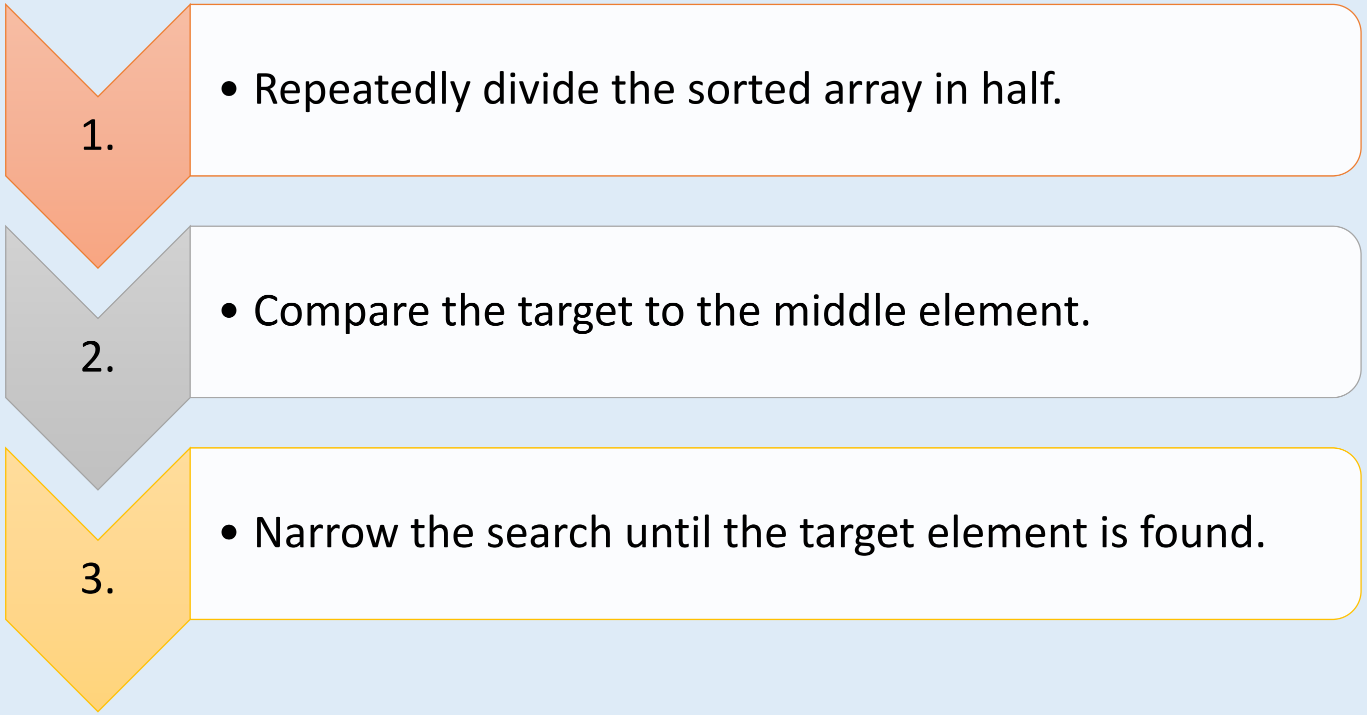
```java
public class BinarySearch {

  public static void main(String[] args) {
    int[] numbers = { 2, 3, 4, 10, 40 };
    int target = 10;
    int result = binarySearch(numbers, target);

    if (result == -1) {
      System.out.println("Element not present
         in the array");
    } else {
      System.out.println("Element found at
         index " + result);
    } // Element found at index 3
  }
```

```java
  // Binary search function
  public static int binarySearch(int[] array, int target) {
    int left = 0;
    int right = array.length - 1; // 4

    while (left <= right) { // 0 <= 4, 3 <= 4
      int mid = left + (right - left) / 2; // 2, 3

      // Check if target is present at mid
      if (array[mid] == target) { // 4 == 10, 10 == 10
        return mid; // Target found at index mid
      }

      // If target is greater, ignore the left half
      if (array[mid] < target) { // 4 < 10
        left = mid + 1; // 3
      }
      else { // If target is smaller, ignore the right half
        right = mid - 1;
      }
    }
    return -1; // Target not found in array
  }
}
```

back to chapter index

# Q. Write a function to search an element in any array(Binary Search)? V. IMP.

- Binary search steps:



1. Repeatedly divide the sorted array in half.

2. Compare the target to the middle element.

3. Narrow the search until the target element is found.

back to chapter index

# Java Chapters

## SPRING

1.Basics, IoC & DI
2.Components & Beans
3.Configuration & Annotations
4.Scopes of a bean
5.Others

## SPRING BOOT

6. Basics
7. Project structure,
   Configuration & Actuator

## SPRING MVC

8. Basics
9. Important Annotations

Back to main index

# Spring - Basics, IoC & DI

Q. What is Spring framework? What are the core features of Spring? **V. IMP.**

Q. What is Inversion of Control (IoC)? What is Loose coupling? **V. IMP.**

Q. How to achieve IoC? What is the difference between Principle & Design Pattern?

Q. What is Dependency Injection? How to implement it? **V. IMP.**

Q. What are the must to know things before learning Dependency Injection in Spring?

Q. How to implement Dependency Injection in code using Spring? **V. IMP.**

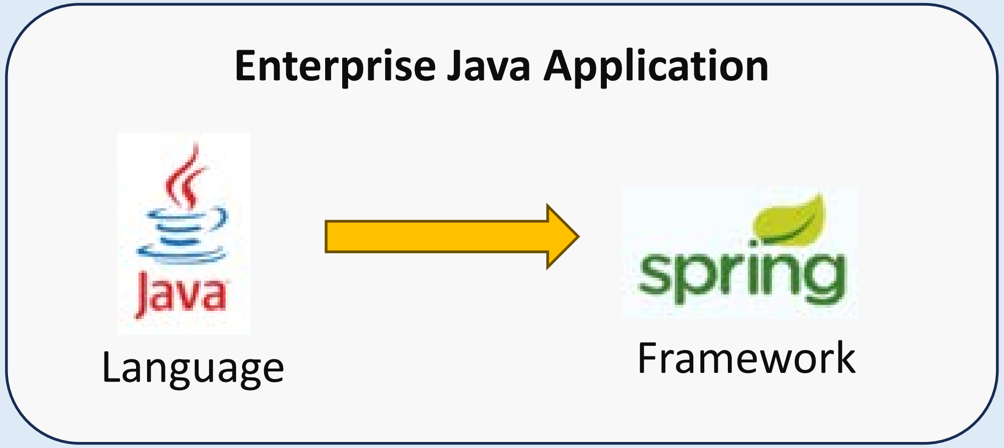
Q. What are the 4 ways of implementing Dependency Injection in Spring?

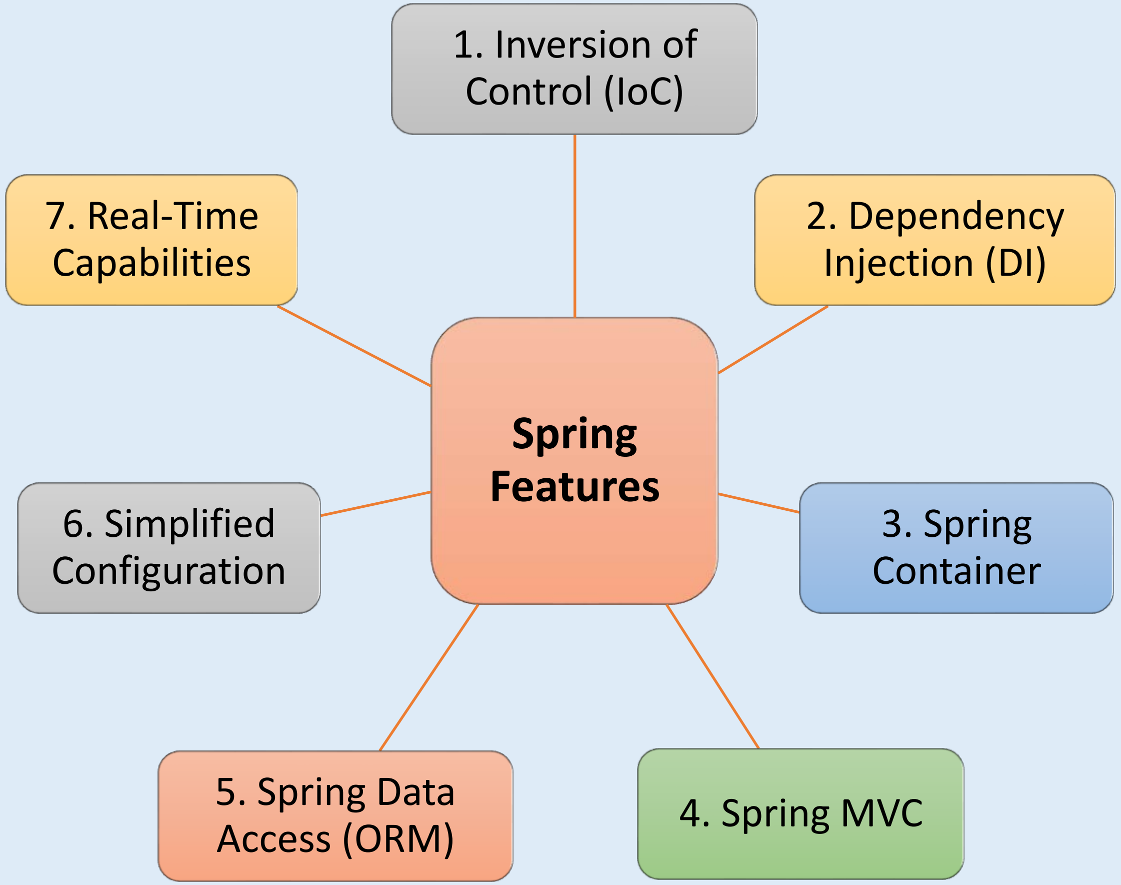
Q. When to use which type of dependency injection?

Back to main index

# Q. What is Spring framework? What are the core features of Spring? V. IMP.

- The Spring Framework is an open-source framework which is used for building enterprise Java applications.



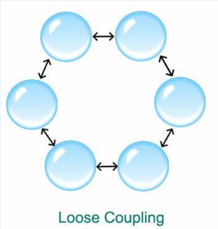

Spring Features
1. Inversion of Control (IoC)
2. Dependency Injection (DI)
3. Spring Container
4. Spring MVC
5. Spring Data Access (ORM)
6. Simplified Configuration
7. Real-Time Capabilities

back to chapter index

# Q. What is Inversion of Control (IoC)? What is Loose coupling? V. IMP.

- IoC (Inversion of Control) is a design principle used for creating **loosely coupled architecture**. **OR**
- IoC (Inversion of Control) is a design principle where the framework **controls object creation** and dependency management, enhancing modularity and loose coupling in applications.

- Loose coupling means classes or components are **independent** of each other and replacing one class will have minimum impact on other.



School Management Application

```
// Dependency class
public class MathStudent {

  public int GetStudentCount() {
    return 50;
  }
}
```

```
// Dependendent class
public class School {
  public static void main(String[] args) {

    MathStudent student = new MathStudent();
    int count = student.GetStudentCount();
    System.out.println(count);
  }
}
```

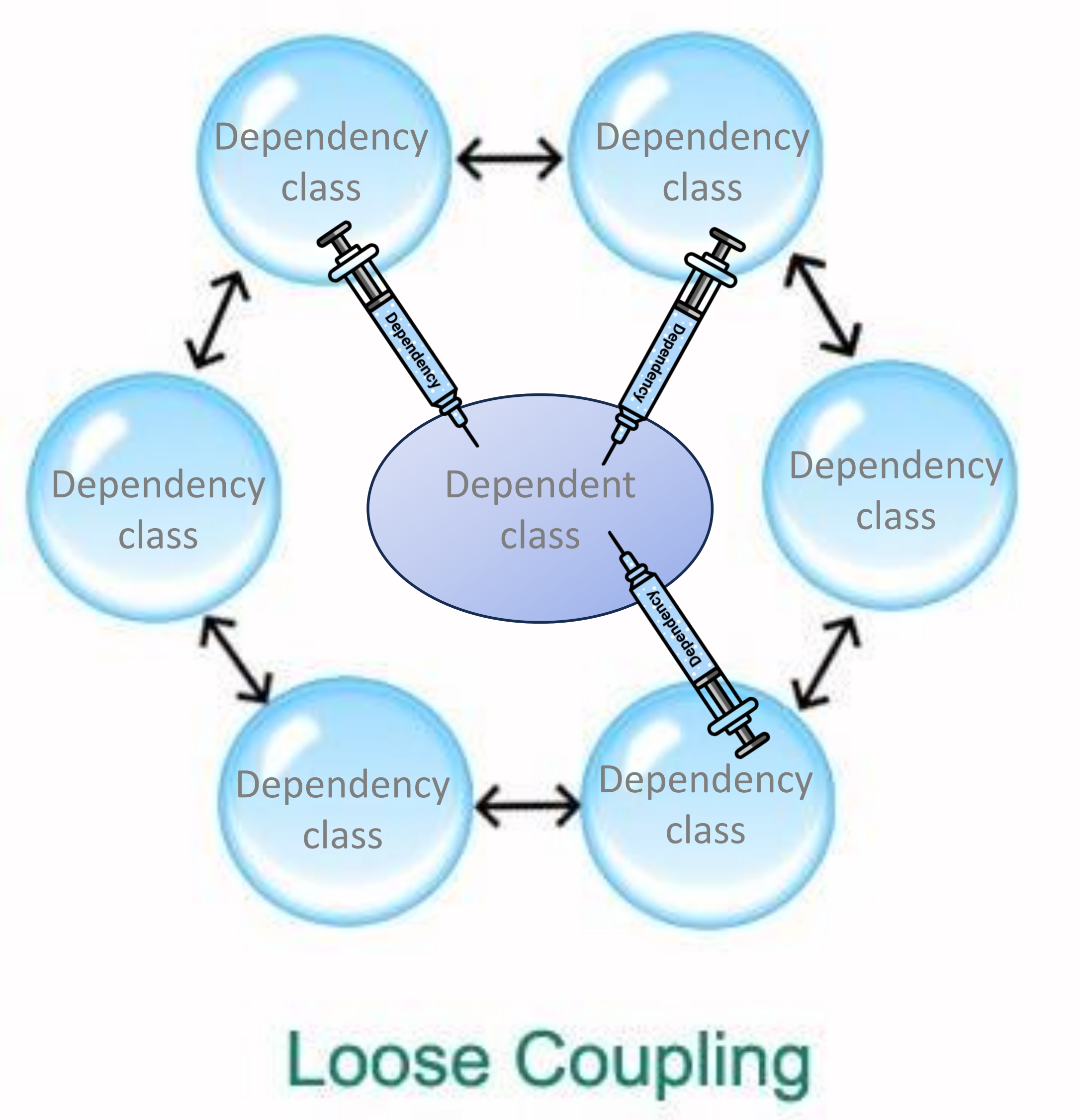
Violation of IoC

back to chapter index

# Q. How to achieve IoC? What is the difference between Principle & Design Pattern? V. IMP.

- Dependency Injection is a **design pattern** which is used to implement **Inversion of Control** principle(Loose coupling).

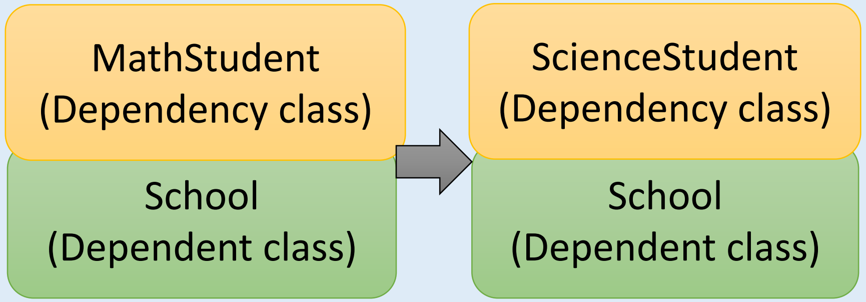
- Principles are fundamental ideas(ideologies) that we should follow in our projects. If we go against these principles, it could negatively impact our project. Examples IoC principle, SOLID principles, the DRY principle, KISS principle etc.
- Design patterns are the specific, practical ways to implement these principles in software development.



back to chapter index

# Q. What is Dependency Injection? How to implement it? V. IMP. V. IMP.

- Dependency Injection is a **design pattern** which is used to implement **Inversion of Control** principle(Loose coupling) in Java.



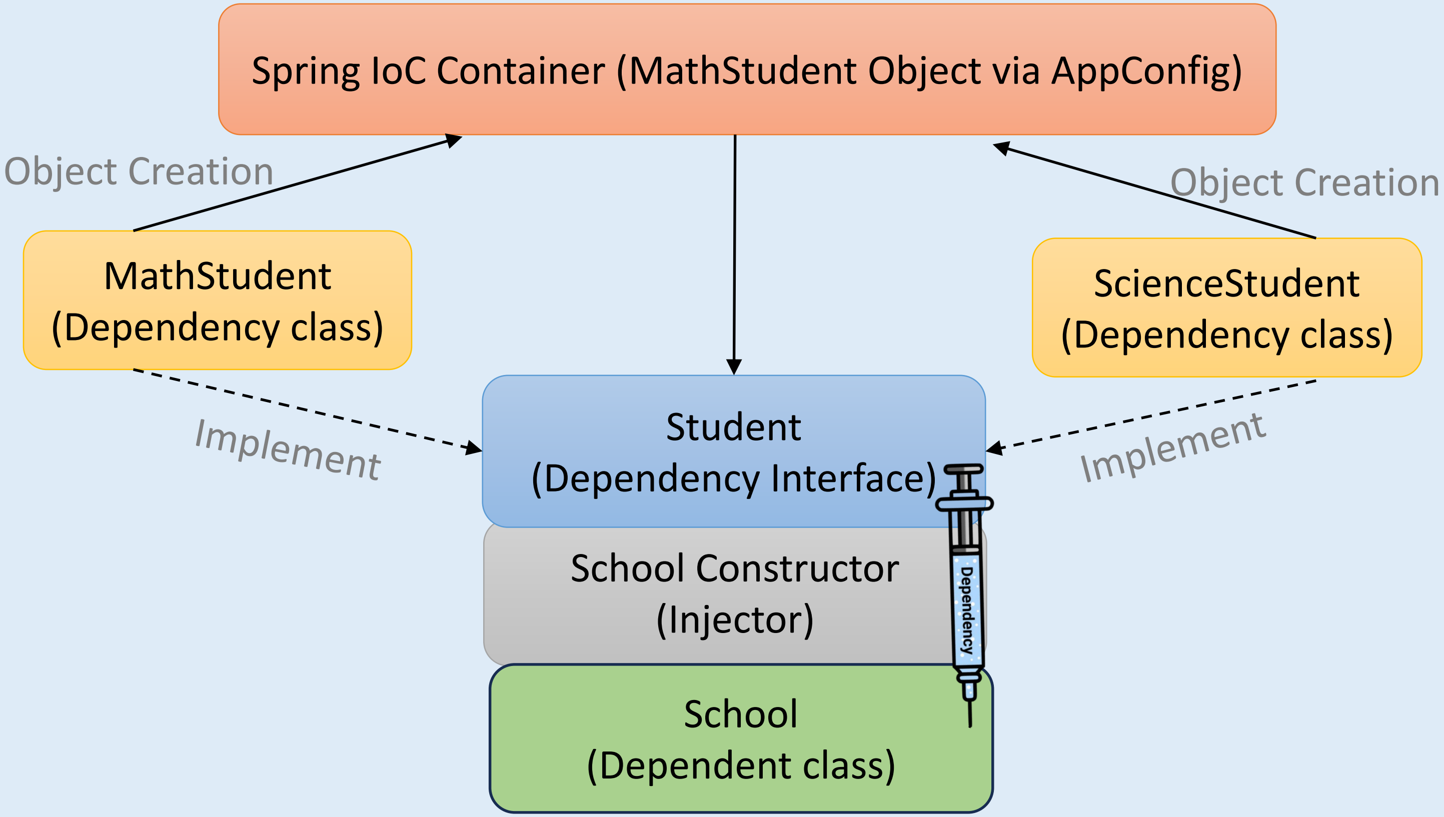
MathStudent (Dependency class)

School (Dependent class)

ScienceStudent (Dependency class)

School (Dependent class)

- Without Dependency Injection: Services are tightly coupled with client because of direct object creation



Spring IoC Container (MathStudent Object via AppConfig)

Object Creation

Object Creation

MathStudent (Dependency class)

ScienceStudent (Dependency class)

Implement

Implement

Student (Dependency Interface)

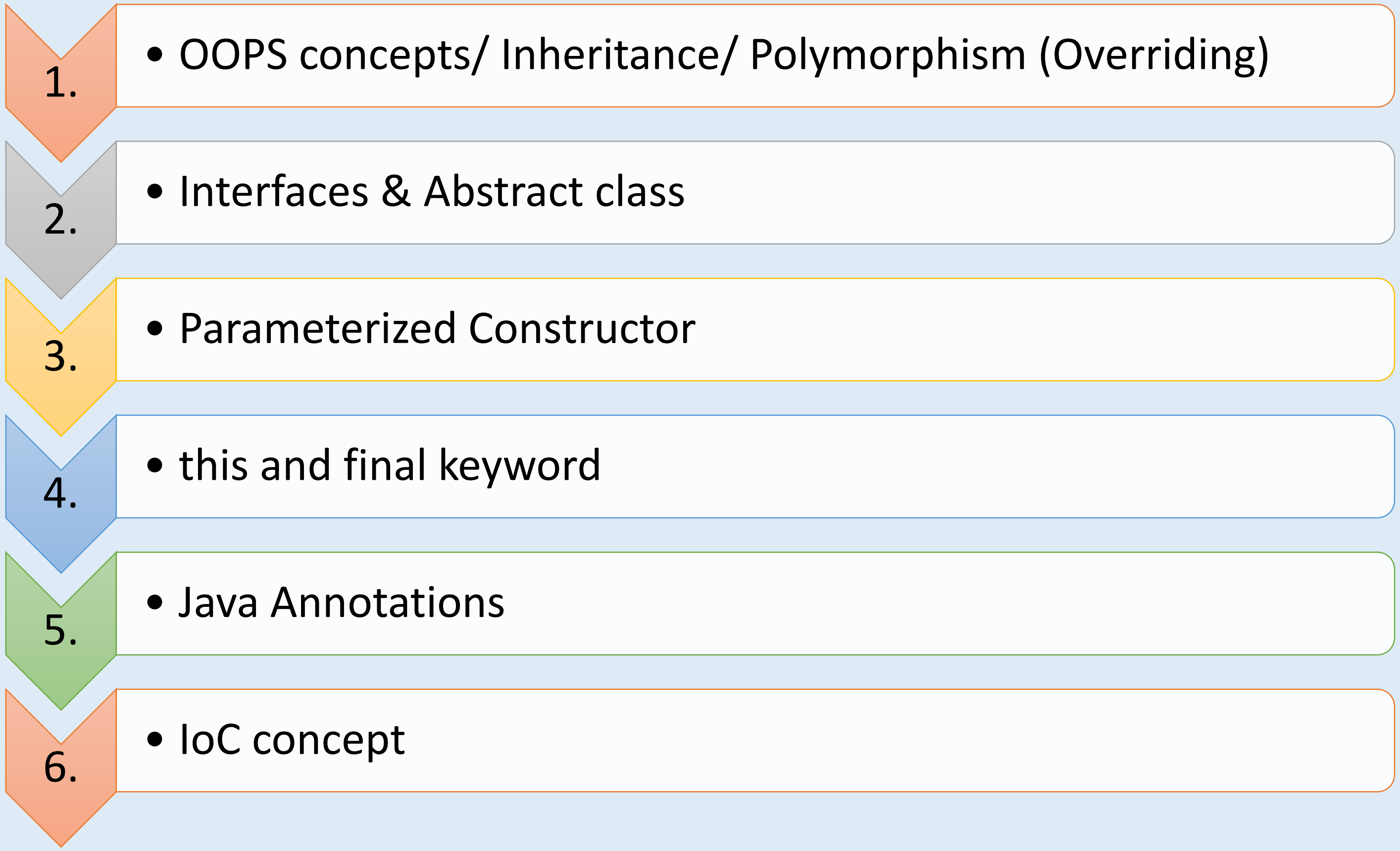
School Constructor (Injector)

School (Dependent class)

- With Dependency Injection: Services are loosely coupled with client because objection creation is handled externally.

back to chapter index

# Q. What are the must to know things before learning Dependency Injection in Spring?



1.
- OOPS concepts/ Inheritance/ Polymorphism (Overriding)

2.
- Interfaces & Abstract class

3.
- Parameterized Constructor

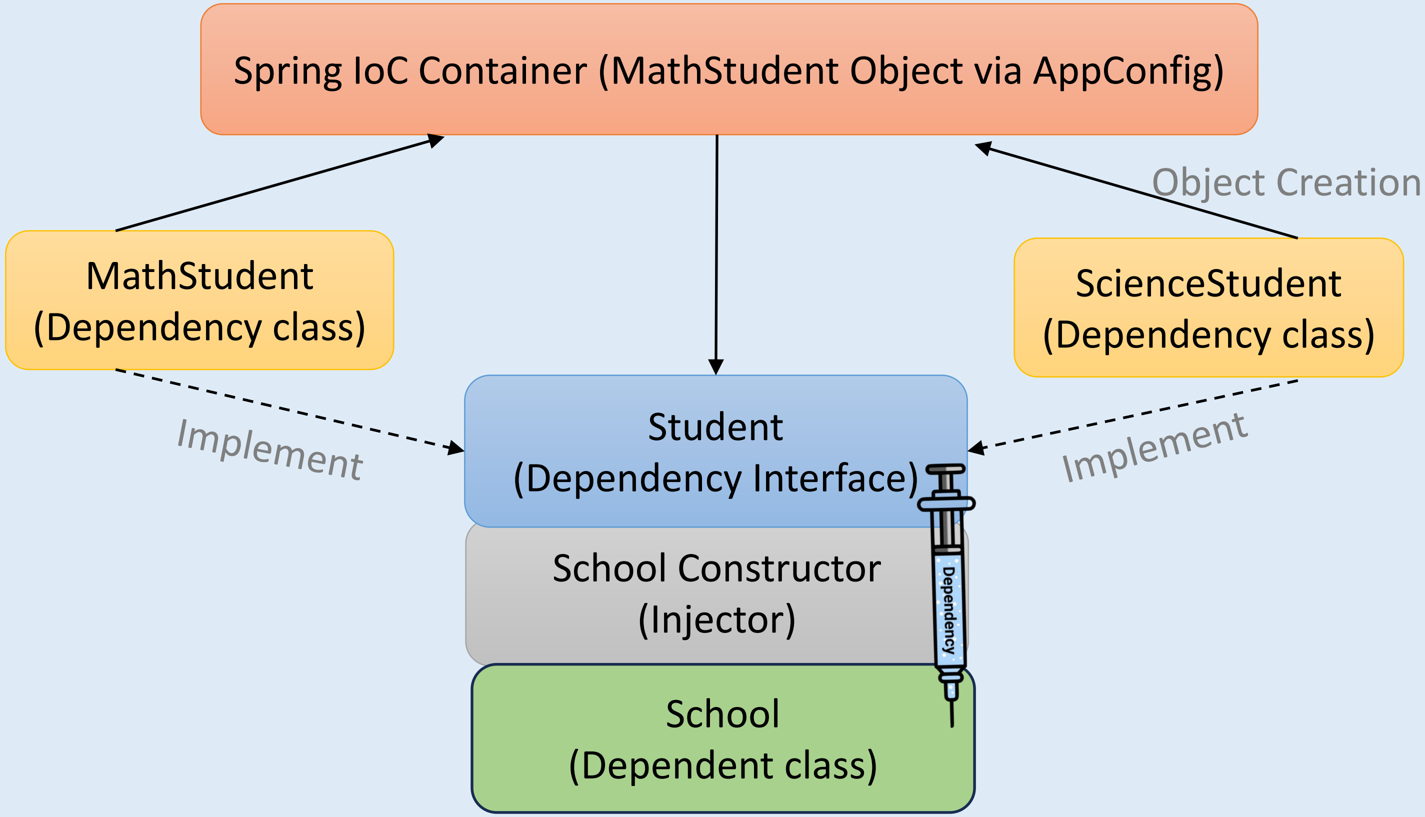
4.
- this and final keyword

5.
- Java Annotations

6.
- IoC concept

back to chapter index

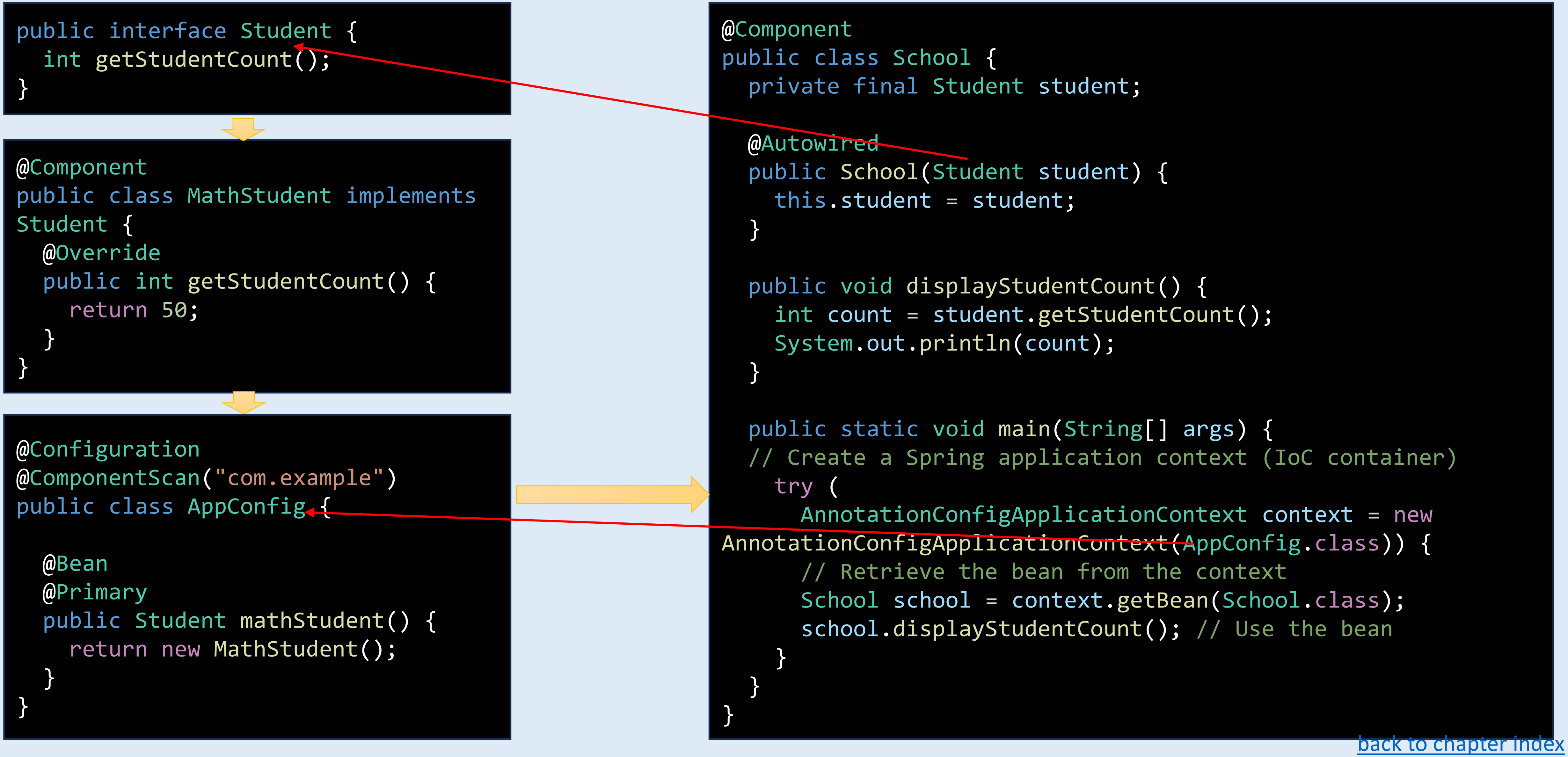
# Q. How to implement Dependency Injection in code using Spring? V. IMP.


Spring IoC Container (MathStudent Object via AppConfig)
Object Creation
MathStudent (Dependency class)
ScienceStudent (Dependency class)
Implement
Implement
Student (Dependency Interface)
School Constructor (Injector)
Dependency
School (Dependent class)

- Dependency Injection Implementation

back to chapter index

# Q. How to implement Dependency Injection in code using Spring? V. IMP.



```java
public interface Student {
  int getStudentCount();
}
```

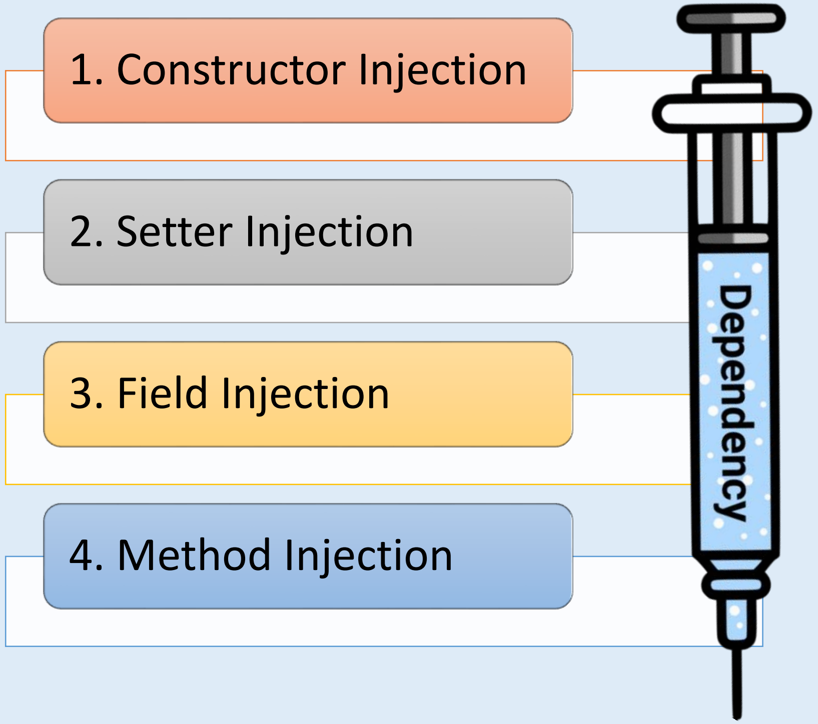
```java
@Component
public class MathStudent implements
Student {
  @Override
  public int getStudentCount() {
    return 50;
  }
}
```

```java
@Configuration
@ComponentScan("com.example")
public class AppConfig {

  @Bean
  @Primary
  public Student mathStudent() {
    return new MathStudent();
  }
}
```

```java
@Component
public class School {
  private final Student student;

  @Autowired
  public School(Student student) {
    this.student = student;
  }

  public void displayStudentCount() {
    int count = student.getStudentCount();
    System.out.println(count);
  }

  public static void main(String[] args) {
  // Create a Spring application context (IoC container)
    try (
      AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext(AppConfig.class)) {
      // Retrieve the bean from the context
      School school = context.getBean(School.class);
      school.displayStudentCount(); // Use the bean
    }
  }
}
```

back to chapter index

# Q. What are the 4 ways of implementing Dependency Injection in Spring? V. IMP.



1. Constructor Injection

2. Setter Injection

3. Field Injection

4. Method Injection

```java
@Service
public class MyService {

    @Autowired // Field Injection
    private MyRepository repository;

    private final MyOtherService otherService;
    private MyThirdService thirdService;
    private MyFourthService fourthService;

    @Autowired // Constructor Injection
    public MyService(MyOtherService otherService) {
        this.otherService = otherService;
    }

    @Autowired // Setter Injection
    public void setThirdService(MyThirdService thirdService) {
        this.thirdService = thirdService;
    }

    @Autowired // Method Injection
    public void configure(MyFourthService fourthService) {
        // Use fourthService as needed
    }
}
```

back to chapter index

# Q. When to use which type of dependency injection?

**1. Constructor Injection**

Use for **mandatory dependencies** and should be provided at object creation time.

**2. Setter Injection**

Use for **optional dependencies** or when dependencies can change after object creation, allowing flexibility in re-injecting dependencies.

**3. Field Injection**

Generally, avoid due to **limitations in testability** and visibility of dependencies.

**4. Method Injection**

Use for **injecting dependencies dynamically** or conditionally, providing more flexibility compared to constructor or setter injection.

back to chapter index

# Spring - Components & Beans

Q. What is the role of Component and @Component annotation in Spring?

Q. What are Bean & @Bean annotation? Difference between Bean & Component?

Q. What are the ways to define a Bean in Spring?

Q. What is the life cycle of a Bean in Spring? **V. IMP.**

Back to main index

# Q. What is the role of Component and @Component annotation in Spring?

- The @Component annotation is used to indicate that a class is a **Spring-managed component**.

- In Spring, a Component refers to a class or bean that is managed by the Spring IoC .

```
@Component
public class MathStudent implements Student {
  @Override
  public int getStudentCount() {
    return 50;
  }
}
```

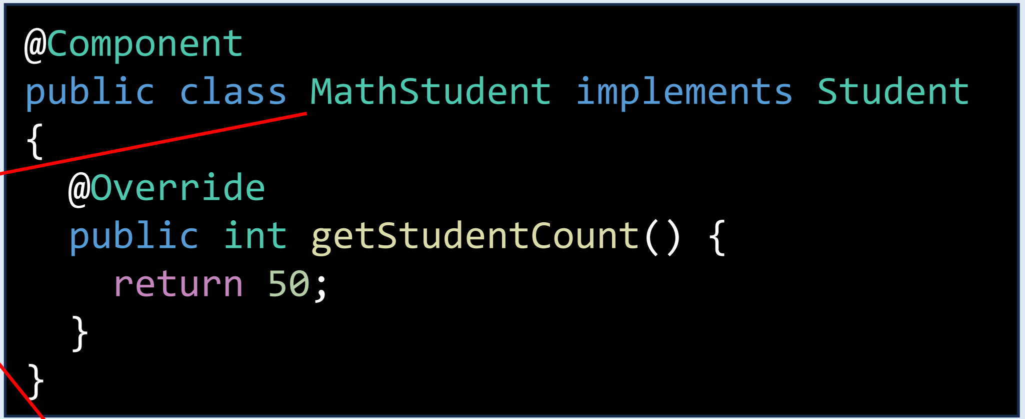
```
@Component
public class School {
  private final Student student;

  @Autowired
  public School(Student student) {
    this.student = student;
  }

  public void displayStudentCount() {
    int count = student.getStudentCount();
    System.out.println(count);
  }

  public static void main(String[] args) {
  // Create a Spring application context(IoC container)
    try (
      AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext(AppConfig.class)
    ) { // Retrieve the bean from the context
      School school = context.getBean(School.class);
      school.displayStudentCount(); // Use the bean
    }
  }
}
```

back to chapter index

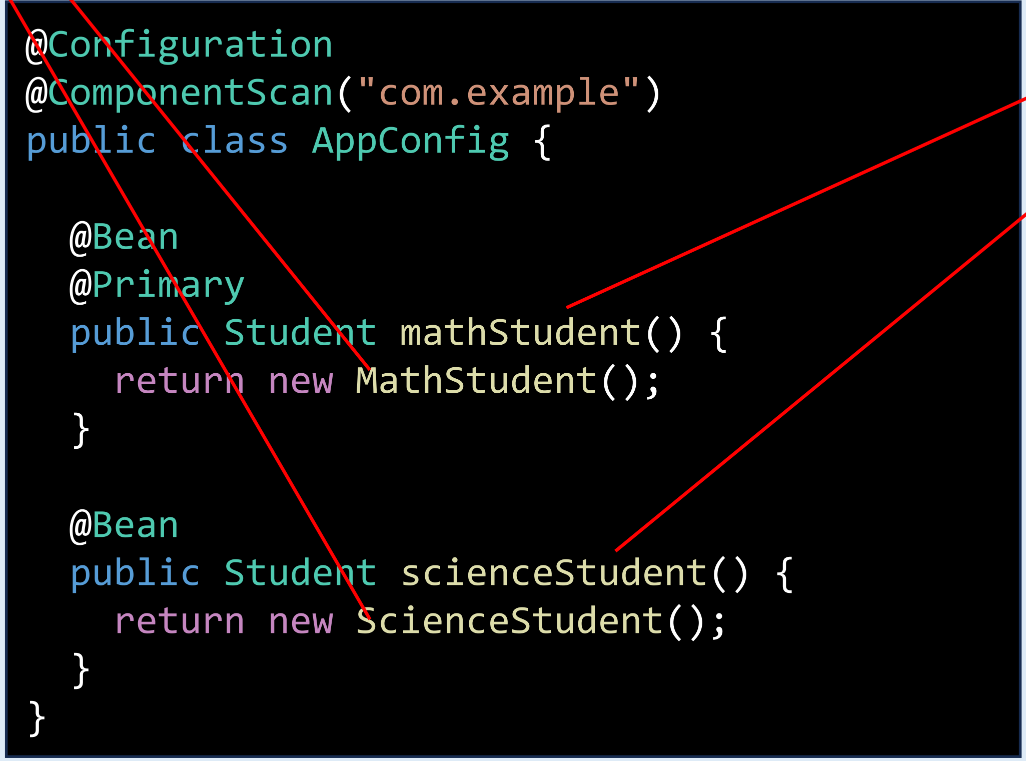
# Q. What are Bean & @Bean annotation? Difference between Bean & Component?

- A Spring bean is an object that is instantiated, configured, and managed by the Spring IoC container. These beans are the building blocks of a Spring application.

- Every component in spring is a bean, but every bean is not a component.



```java
@Component
public class MathStudent implements Student
{
  @Override
  public int getStudentCount() {
    return 50;
  }
}
```



```java
@Configuration
@ComponentScan("com.example")
public class AppConfig {

  @Bean
  @Primary
  public Student mathStudent() {
    return new MathStudent();
  }

  @Bean
  public Student scienceStudent() {
    return new ScienceStudent();
  }
}
```

- @Bean annotation in Spring is used to indicate that a method is a **Bean method**, and it produces or returns a bean to be managed by the Spring container.

back to chapter index

# Q. What are the ways to define a Bean in Spring?

1. Annotate a class with **@Component, @Service, @Repository, or @Controller**. Spring will automatically detect and register these classes as Beans through component scanning.

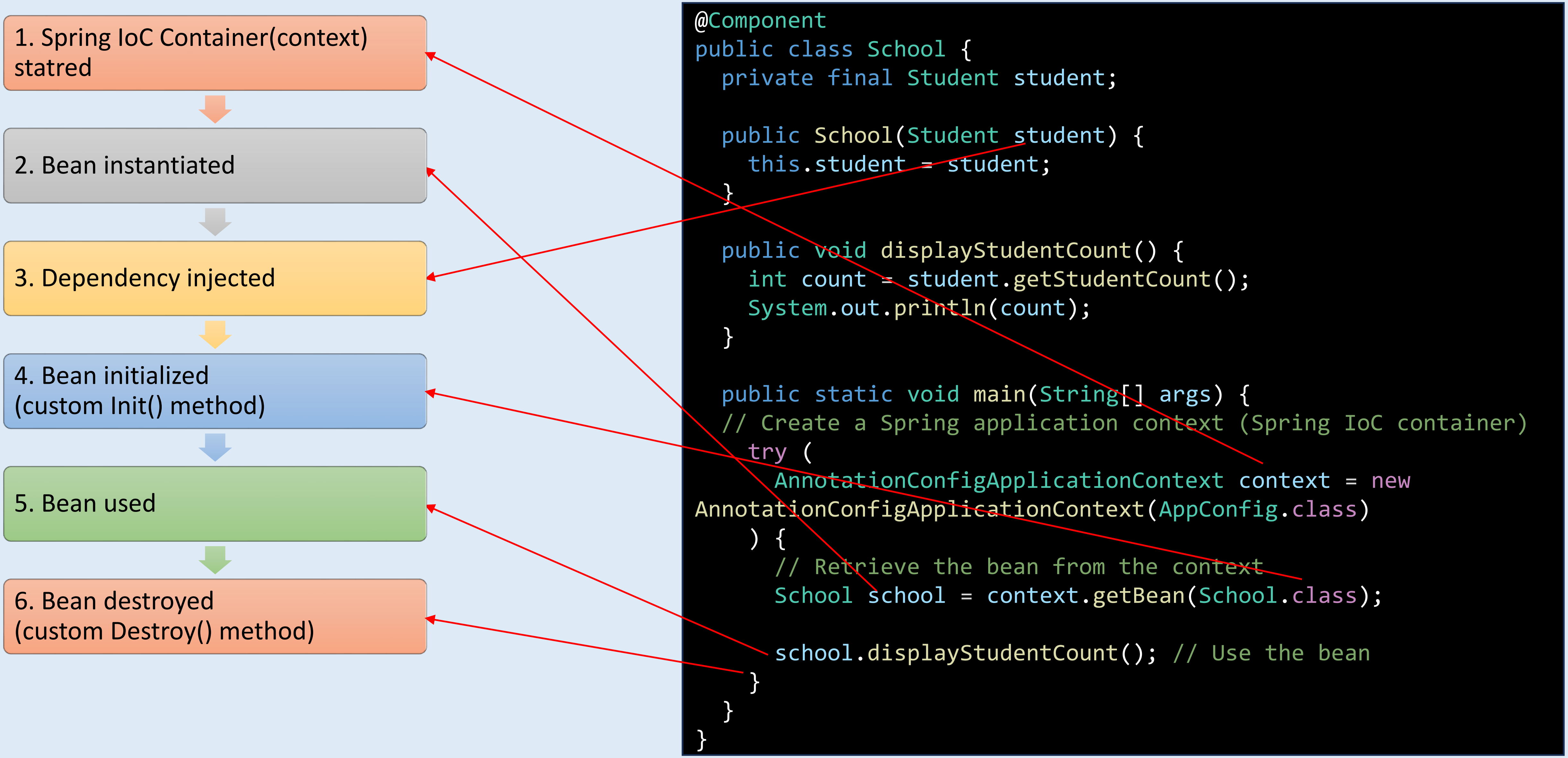
```java
@Component // Used for general purpose
public class MyComponent {}

@Service // Used for business logic
public class MyService {}

@Repository // Used for database operations
public class MyRepository {}

@Controller // Used for handling HTTP requests
public class MyController {}
```

2. Using **@Configuration** and **@Bean Annotation** for defining method beans in a @Configuration Class. Method Bean returns the Bean instance.

```java
@Configuration // Used for containing Method Beans
public class AppConfig {

    @Bean // Method Bean used to return a Spring Bean
    public MyBean myBean() {
        return new MyBean();
    }
}
```

back to chapter index

# Q. What is the life cycle of a Bean in Spring? V. IMP.



1. Spring IoC Container(context) statred

2. Bean instantiated

3. Dependency injected

4. Bean initialized (custom Init() method)

5. Bean used

6. Bean destroyed (custom Destroy() method)

```
@Component
public class School {
  private final Student student;

  public School(Student student) {
    this.student = student;
  }

  public void displayStudentCount() {
    int count = student.getStudentCount();
    System.out.println(count);
  }

  public static void main(String[] args) {
  // Create a Spring application context (Spring IoC container)
    try (
      AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext(AppConfig.class)
    ) {
      // Retrieve the bean from the context
      School school = context.getBean(School.class);

      school.displayStudentCount(); // Use the bean
    }
  }
}
```

back to chapter index

# Q. What is the life cycle of a Bean in Spring? V. IMP.

**1. Spring IoC Container(context) Started**

During the container startup, Spring reads the bean definitions (either from XML configuration, or Java configuration) and registers them within the application context.

**2. Bean Instantated**

When a bean is needed (typically when requested by the application), the Spring container creates an instance of the bean.

**3. Dependency Injected**

After the bean is instantiated, the Spring container performs dependency injection to inject any required dependencies into the bean. This can be done using constructor injection, setter injection, or field injection, based on the configuration.

**4. Bean Initialization**

After dependency injection, if specified, Spring invokes initialization methods perform any necessary setup tasks.

**5. Bean Usage**

Components can access and use this bean to perform business logic, data access, or other tasks.

**6. Bean Destruction**

Optionally, performing cleanup when the bean is no longer needed.

back to chapter index

# Spring - Configuration & Annotations

Q. What is AppConfig, @Configuration and @ComponentScan?

Q. What is the role of @Primary annotation in AppConfig? **V. IMP.**

Q. Do we need @Primary annotation if there is only one bean method?

Q. What is Spring IoC container? What is AnnotationConfigApplicationContext? **V. IMP.**

Q. What is the XML-based Configuration?

Q. What is the role of @Autowired annotation? **V. IMP.**

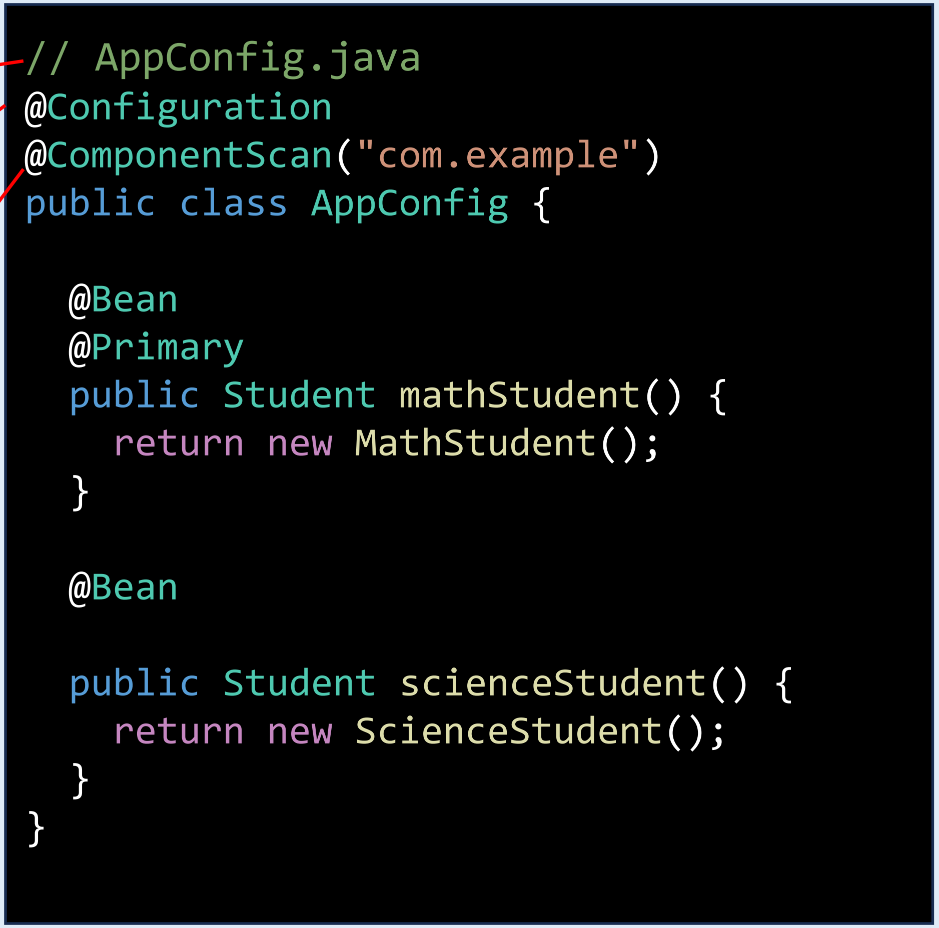
Back to main index

# Q. What is AppConfig, @Configuration and @ComponentScan?

- AppConfig is a Spring configuration class that defines beans.

- @Configuration is an annotation in the Spring used to indicate that the class declares one or more @Bean methods.

- @ComponentScan is an annotation used to indicate that Spring will automatically scan and register beans from the specified package(comp.example).



```java
// AppConfig.java
@Configuration
@ComponentScan("com.example")
public class AppConfig {

  @Bean
  @Primary
  public Student mathStudent() {
    return new MathStudent();
  }

  @Bean

  public Student scienceStudent() {
    return new ScienceStudent();
  }
}
```

back to chapter index

# Q. What is the role of @Primary annotation in AppConfig? V. IMP.

- @Primary annotation marks method bean as the primary bean to be injected when multiple beans of the same type are present.

```java
// AppConfig.java
@Configuration
@ComponentScan("com.example")
public class AppConfig {
  @Bean
  @Primary
  public Student mathStudent() {
    return new MathStudent();
  }

  @Bean
  public Student scienceStudent() {
    return new ScienceStudent();
  }
}
```

```java
@Component
public class School {
  private final Student student;

  @Autowired
  public School(Student student) {
    this.student = student;
  }

  public void displayStudentCount() {
    int count = student.getStudentCount();
    System.out.println(count);
  }

  public static void main(String[] args) {
  // Create a Spring application context(IoC container)
    try (
      AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext(AppConfig.class)
    ) { // Retrieve the bean from the context
      School school = context.getBean(School.class);
      school.displayStudentCount(); // Use the bean
    }
  }
}
```

back to chapter index

# Q. Do we need @Primary annotation if there is only one bean method?

- No, you do not need the @Primary annotation if there is only one bean.

```
// AppConfig.java
@Configuration
@ComponentScan("com.example")
public class AppConfig {

  @Bean
  public Student mathStudent() {
    return new MathStudent();
  }
}
```

back to chapter index

# Q. What is Spring IoC container? What is AnnotationConfigApplicationContext? V. IMP.

- The Spring IoC container(context) is responsible for managing beans based on the configuration provided in AppConfig.java.
- AnnotationConfigApplicationContext is the inbuilt class provided by Spring framework for creating IoC container(context).

```java
@Component
public class School {
  private final Student student;

  @Autowired
  public School(Student student) {
    this.student = student;
  }

  public void displayStudentCount() {
    int count = student.getStudentCount();
    System.out.println(count);
  }

  public static void main(String[] args) {
  // Create a Spring application context(IoC container)
    try (
      AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext(AppConfig.class)
    ) { // Retrieve the bean from the context
      School school = context.getBean(School.class);
      school.displayStudentCount(); // Use the bean
    }
  }
}
```

back to chapter index

# Q. What is the XML-based Configuration?

- XML-based configuration in Spring refers to the traditional approach of configuring Spring beans and their dependencies using XML files, typically named applicationContext.xml

```
<!-- applicationContext.xml -->

<!-- Enable component scanning to automatically detect Spring components
-->
<context:component-scan base-package="com.example" />

<!-- Define bean definitions for mathStudent and scienceStudent -->
<bean id="mathStudent" class="com.example" primary="true" />
<bean id="scienceStudent" class="com.example" />
```

back to chapter index

# Q. What is the role of @Autowired annotation? V. IMP.

```
@Component
public class School {
  private final Student student;

  @Autowired
  public School(Student student) {
    this.student = student;
  }
  public void displayStudentCount() {
    int count = student.getStudentCount();
    System.out.println(“Students: " + count);
  }

  public static void main(String[] args) {
  // Create a Spring application context
    try (
      AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext(AppConfig.class)
    ) { // Retrieve the bean from the context
     School school = context.getBean(School.class);
      school.displayStudentCount(); // Use the bean
    }
  }
}
```

```
@Component
public class School {
  private final Student student;

  // Constructor for manual dependency injection
  public School(Student student) {
    this.student = student;
  }
  public void displayStudentCount() {
    int count = student.getStudentCount();
    System.out.println(count);
  }

  public static void main(String[] args) {
    // Create Spring application context
    try (
      AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext(AppConfig.class)) {
   // Manually retrieve and inject the dependency
   Student student = context.getBean(Student.class);

   School school = new School(student); //Manual injection

   school.displayStudentCount();
    }
  }
}
```

back to chapter index

# Q. What is the role of @Autowired annotation? V. IMP.

- @Autowired **automatically injects dependencies** into Spring-managed beans, eliminating the need for manual instantiation.

```
@Component
public class School {
  private final Student student;

  @Autowired
  public School(Student student) {
    this.student = student;
  }

  public void displayStudentCount() {
    int count = student.getStudentCount();
    System.out.println(count);
  }

  public static void main(String[] args) {
  // Create a Spring application context
    try (
      AnnotationConfigApplicationContext context = new
         AnnotationConfigApplicationContext(AppConfig.class)
    ) { // Retrieve the bean from the context
      School school = context.getBean(School.class);
      school.displayStudentCount(); // Use the bean
    }
  }
}
```

back to chapter index

# Spring - Scopes of a bean

Q. What is Scope of a bean? What is the default scope of a bean? **V. IMP.**

Q. What are the different types of Spring bean scopes? What is @Scope annotation? **V. IMP.**

Q. What is Singleton Scope? **V. IMP.**

Q. When to use Singleton scope? What are its pros and cons?

Q. What is Prototype Scope?

Q. When to use Prototype scope? What are its pros and cons?

Q. What is Request Scope? When to use it?

Back to main index

# Q. What is Scope of a bean? What is the default scope of a bean? V. IMP.

- The scope of a bean in Spring refers to the lifecycle and visibility of bean instances. For example, how many instances of a bean are created and managed by the Spring IoC container.
- Singleton is the default scope of a bean.

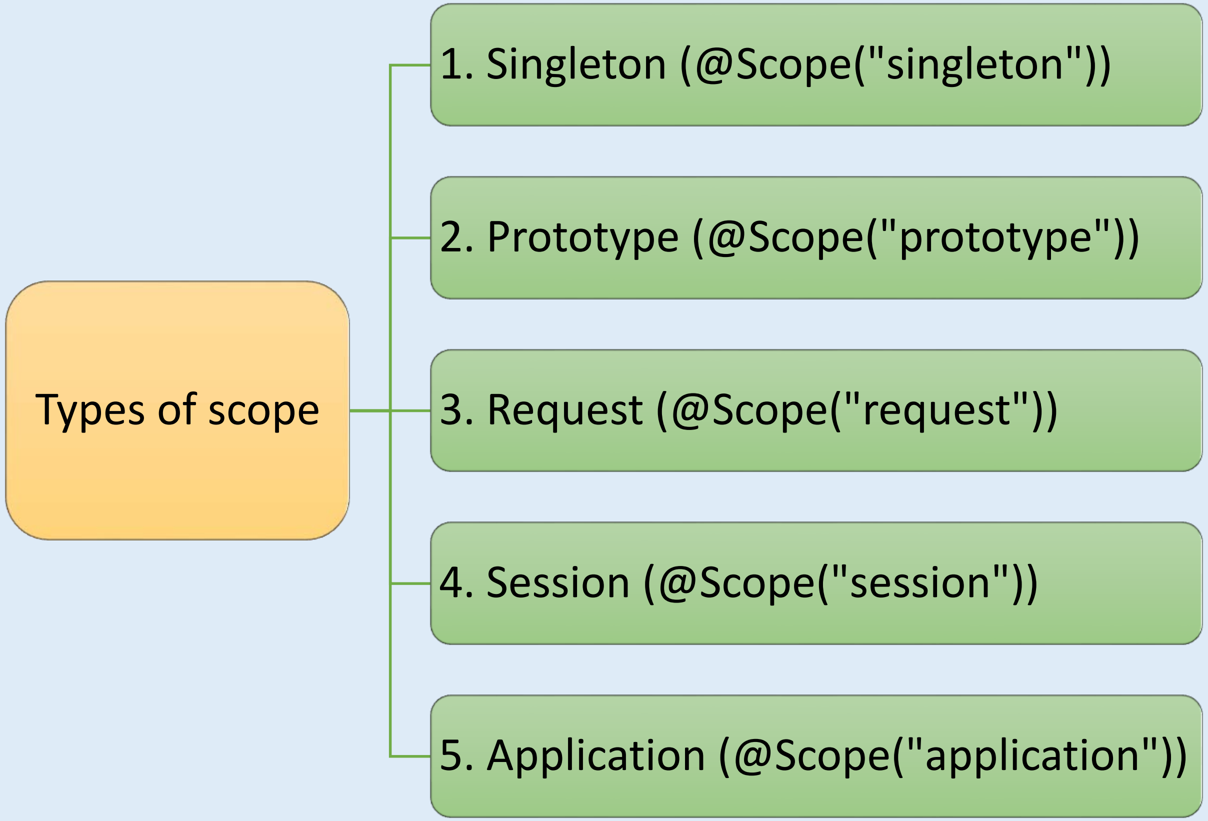
```
@Configuration
@Component
public class AppConfig {

  @Bean
  public Student scienceStudent() {
    return new ScienceStudent();
  }
}
```

=

```
@Configuration
@Component
@Scope("singleton")
public class AppConfig {

  @Bean
  public Student scienceStudent() {
    return new ScienceStudent();
  }
}
```

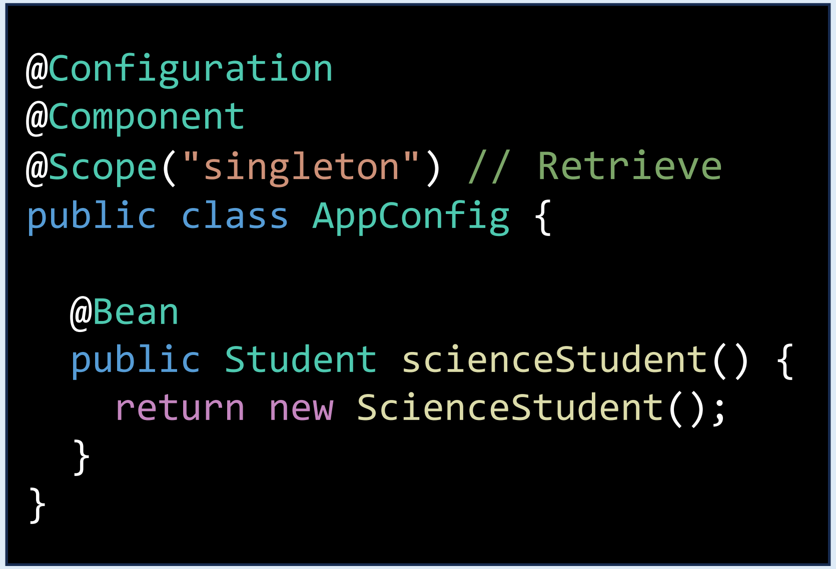
back to chapter index

# Q. What are the different types of Spring bean scopes? What is @Scope annotation? V. IMP.



Types of scope

1. Singleton (@Scope("singleton"))
2. Prototype (@Scope("prototype"))
3. Request (@Scope("request"))
4. Session (@Scope("session"))
5. Application (@Scope("application"))

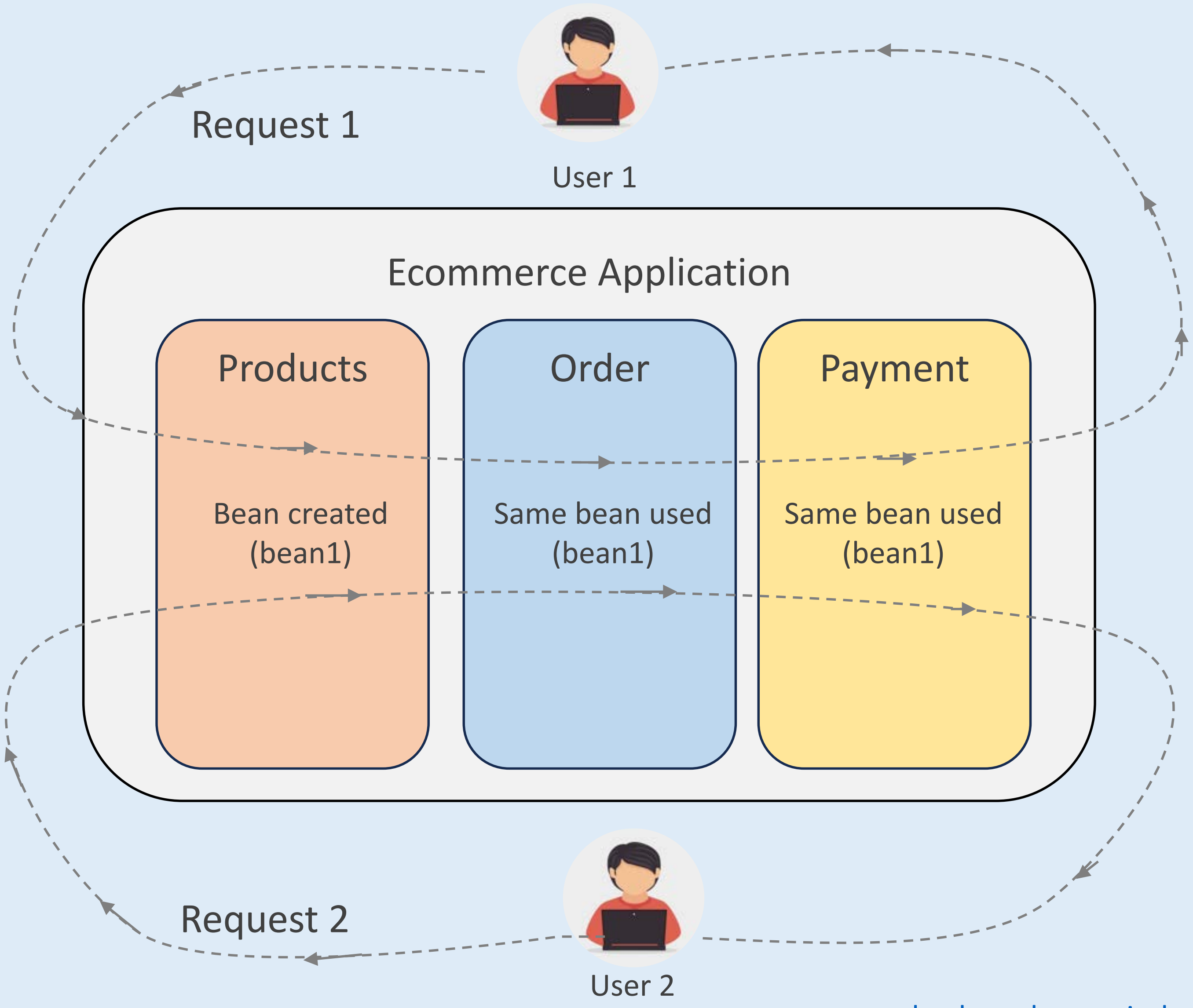


```
@Configuration
@Component
@Scope("singleton") // Retrieve
public class AppConfig {

  @Bean
  public Student scienceStudent() {
    return new ScienceStudent();
  }
}
```

back to chapter index

# Q. What is Singleton Scope? V. IMP.

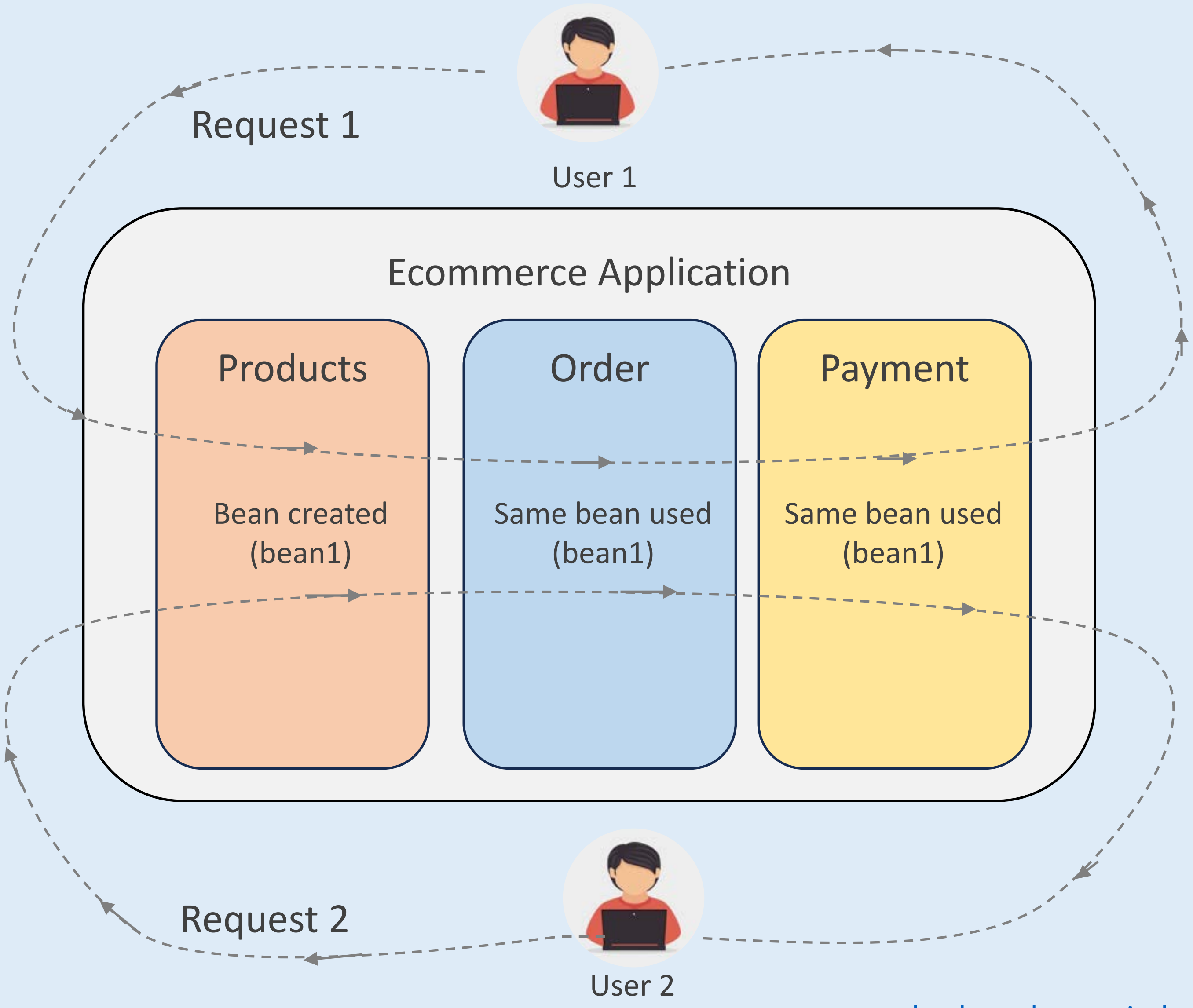
- `@Scope("singleton")` : When a bean is configured to have a Singleton scope, Spring IoC container will create exactly one instance of the bean and will **reuse this same instance** every time the bean is requested (or injected) throughout the lifetime of the application.



back to chapter index

# Q. When to use Singleton scope? What are its pros and cons?

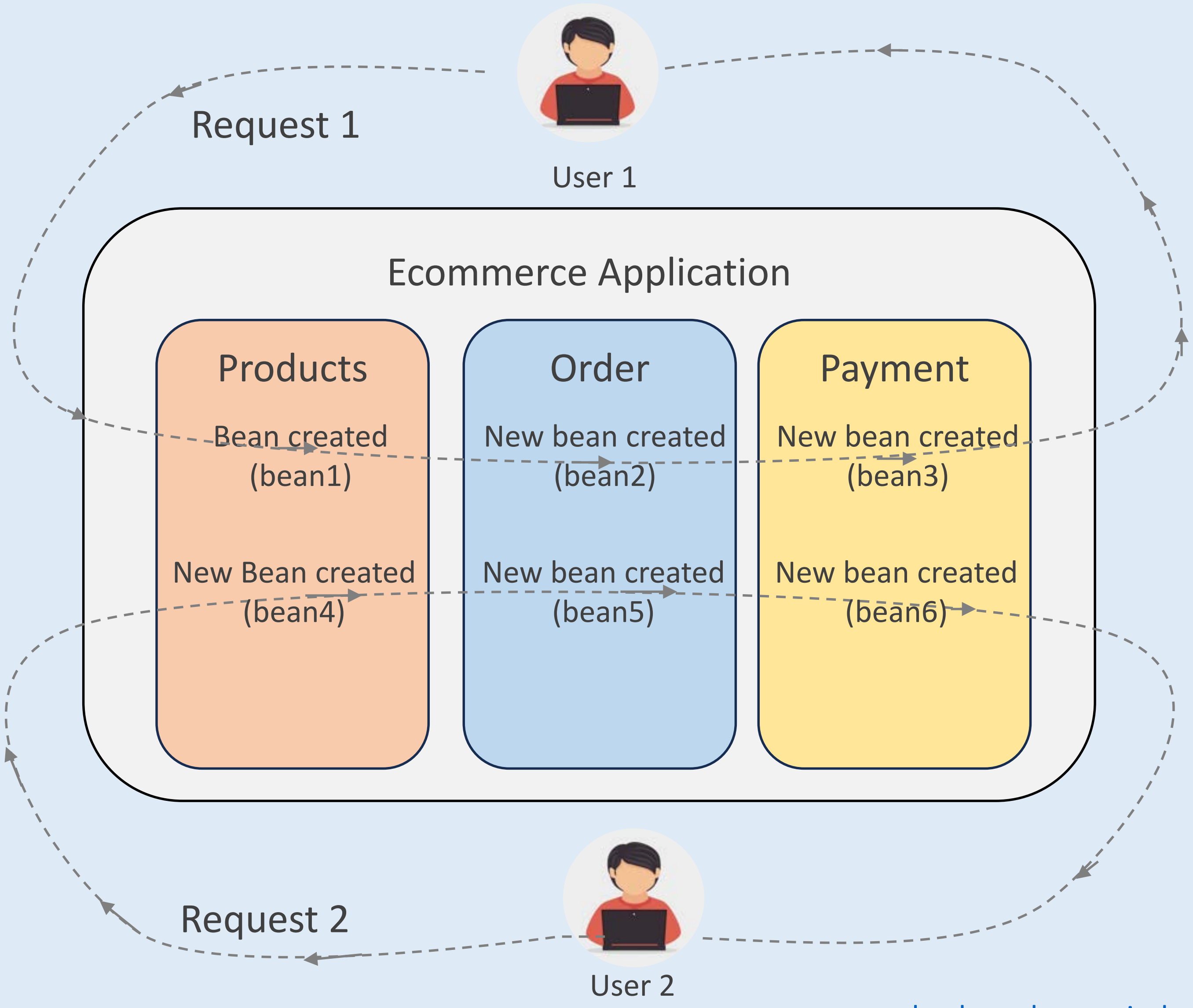
- **Advantage**: Singleton beans are created once and reused for subsequent requests, which can improve **performance** by reducing object creation overhead.
- **Disadvantage:** Singleton beans are shared across multiple threads, so **thread safety** is a concern.
- **Use:** Good for **stateless** based application because when request don't have state(data), then sharing instance is never a problem.



back to chapter index

# Q. What is Prototype Scope?

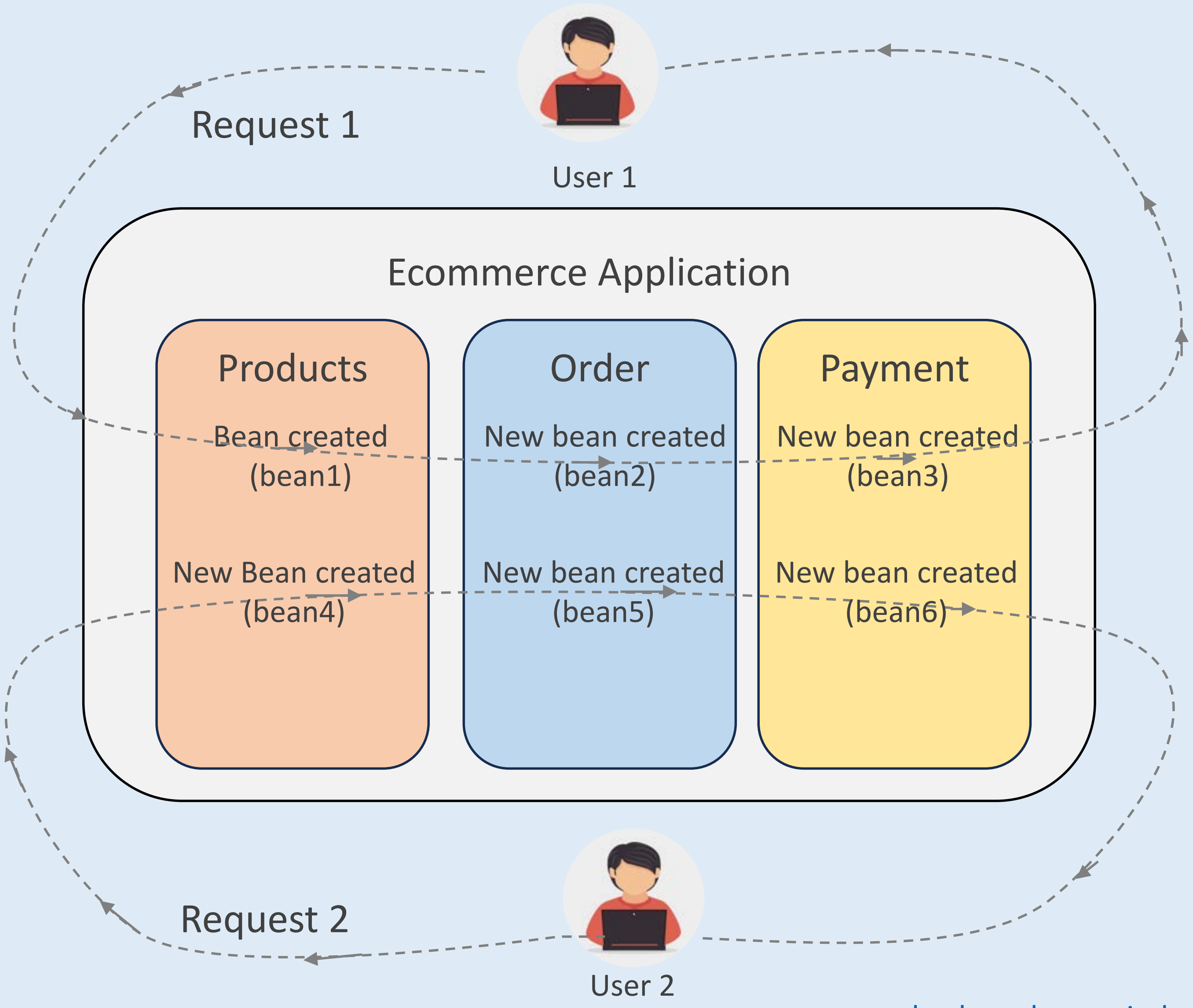
- `@Scope(“prototype")` : When a bean is configured with Prototype scope, Spring IoC container will create a new instance of the bean each time it is requested or injected.
- Prototype-scoped beans are not shared.



back to chapter index

# Q. When to use Prototype scope? What are its pros and cons?

- **Advantage**: Good for **stateful application** because when request maintain states then sharing instances between Requests can create problems.
- **Disadvantage:** Creating a new instance of a prototype bean for every request can impact performance.
- **Use:** Good for **stateful** applications.

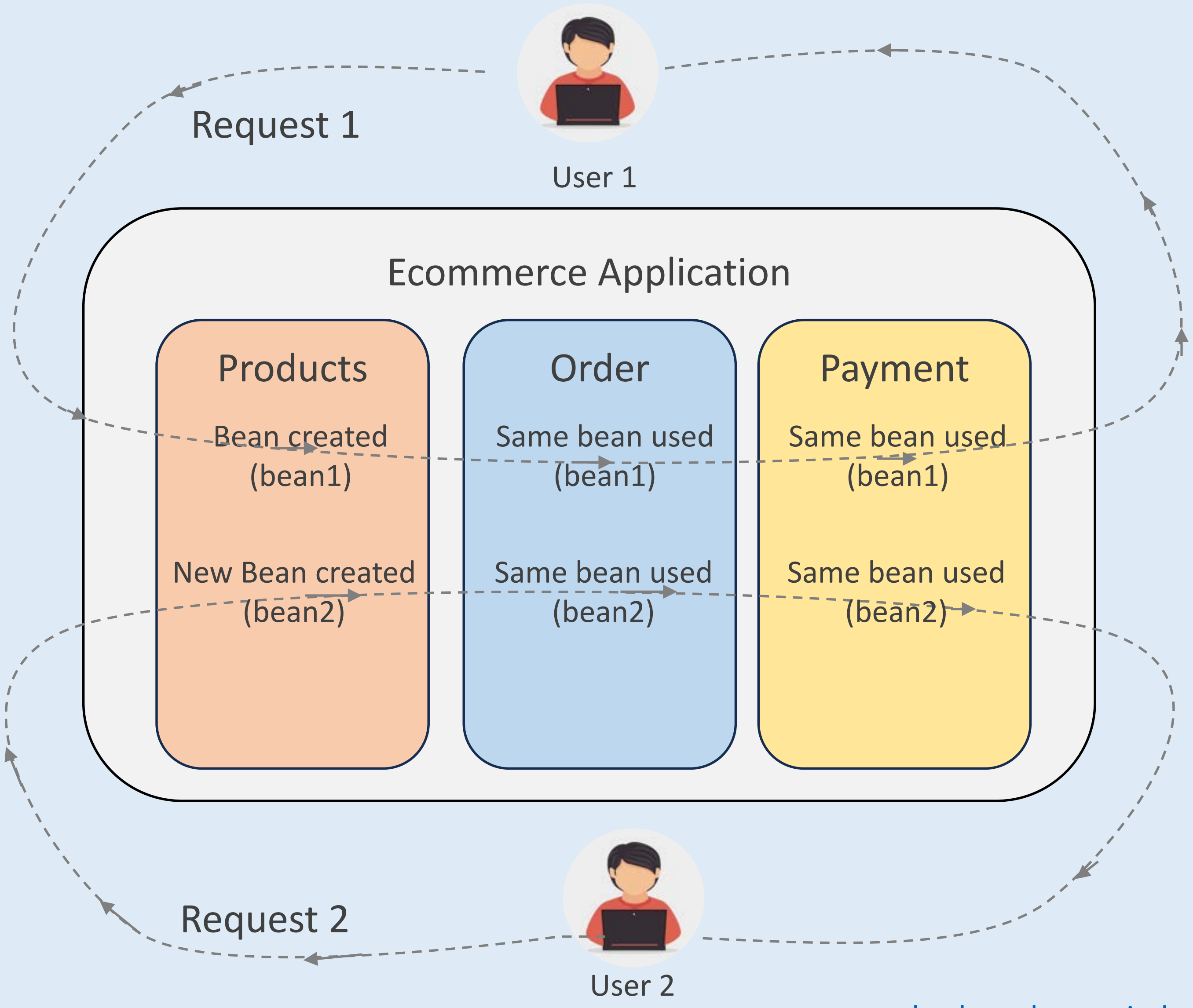


back to chapter index

# Q. What is Request Scope? When to use it?

- `@Scope(“request")`: When a bean is configured with Request scope, the Spring IoC container will create a new instance of the bean for each HTTP request.

- **Use**: Good for applications which maintain state or data specific to individual HTTP requests.



back to chapter index

# Spring - Others

Q. What is Spring AOP? **V. IMP.**

Q. What are circular dependencies? How does Spring handle it?

Q. Can we have multiple Spring configuration files in one project?

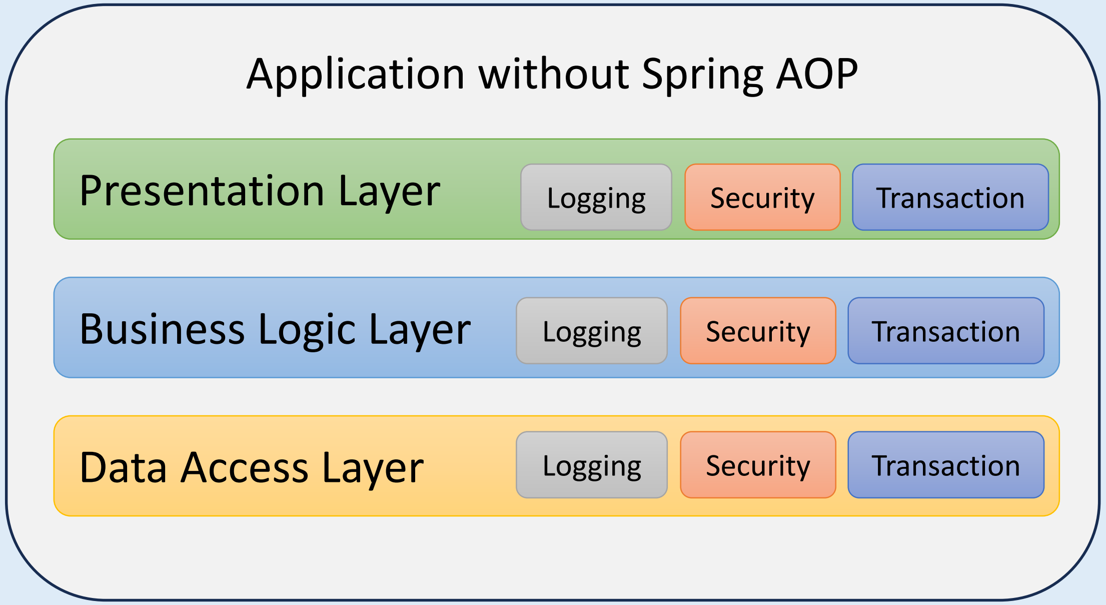
Q. What is BeanFactory in Spring?

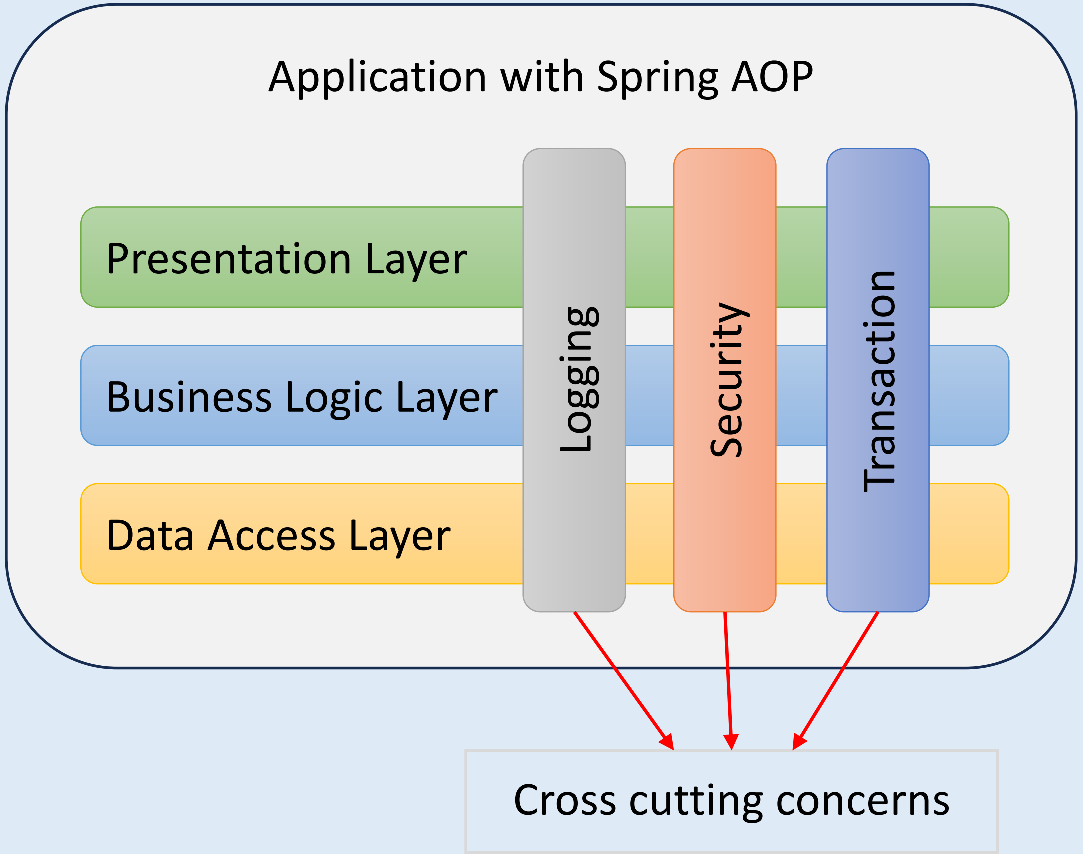
Q. What is the difference between BeanFactory and ApplicationContext? **V. IMP.**

Back to main index

# Q. What is Spring AOP? V. IMP.

- Spring AOP (Aspect-Oriented Programming) allows you to **separate cross-cutting concerns** from the business logic of your application.

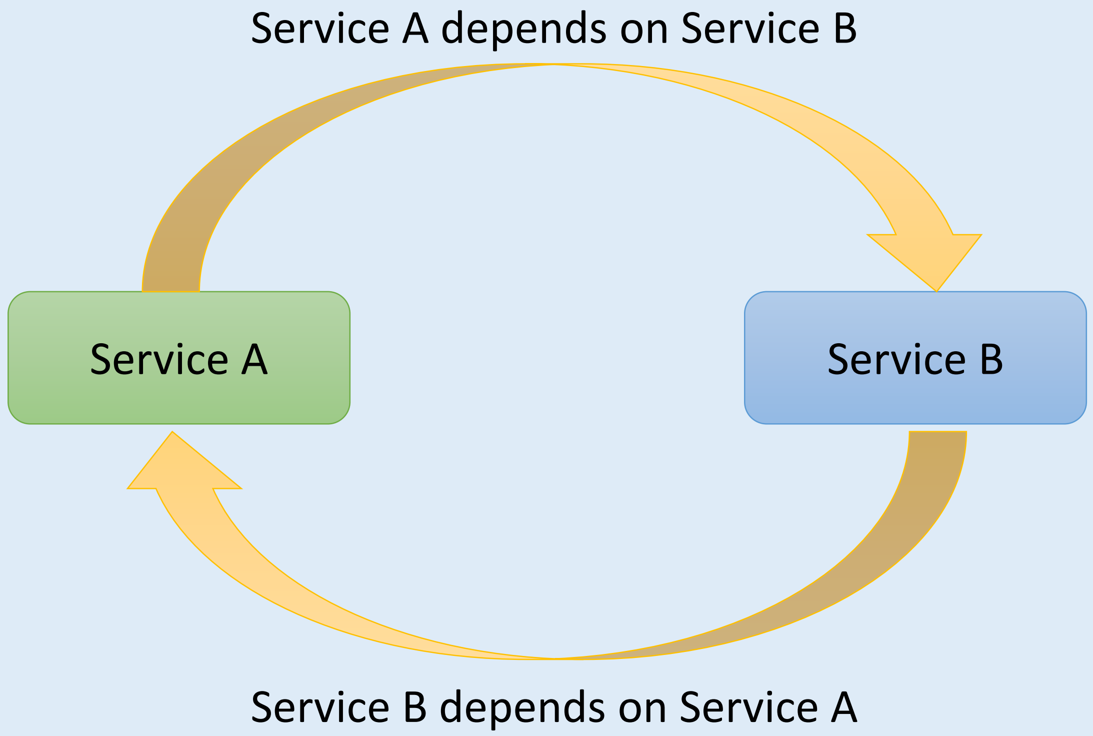




back to chapter index

# Q. What are circular dependencies? How does Spring handle it?

- When services depends on each other that is called circular dependencies.
- Constructor injection can cause circular dependency if services depends on each other (Error: BeanCurrentlyInCreationException).
- **Setter Injection** or **@Lazy Annotation** can be used to resolve circular dependency. They will optionally and lazily inject the dependencies which can avoid the circular dependency


Service A depends on Service B
Service A
Service B
Service B depends on Service A

back to chapter index

# Q. Can we have multiple Spring configuration files in one project?

- Yes, in large projects, having multiple Spring configurations is recommended to increase maintainability and modularity.

```
AnnotationConfigApplicationContext context =
        new AnnotationConfigApplicationContext(AppConfig.class,
WebConfig.class, ServiceConfig.class);
```

back to chapter index

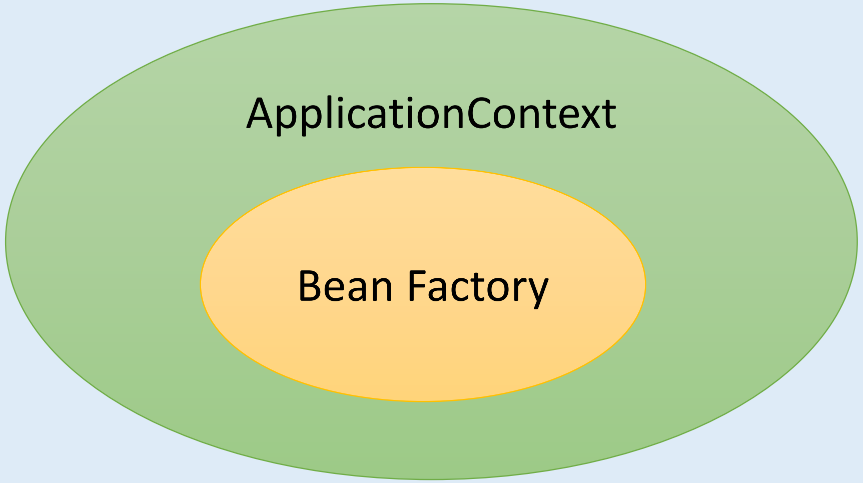
# Q. What is BeanFactory in Spring?

- BeanFactory is a basic container in Spring for **managing beans**. It provides the fundamental features for dependency injection and bean instantiation.

```java
public class BeanFactoryExample {

  public static void main(String[] args) {

    // Load the Spring configuration file using ClassPathResource
    ClassPathResource resource = new ClassPathResource(“beans.xml");

    // Create a BeanFactory instance and load the bean definitions
    BeanFactory beanFactory = new XmlBeanFactory(resource);

    // Retrieve a bean from the BeanFactory
    MyService myService = (MyService) beanFactory.getBean("myService");

    // Use the retrieved bean
    myService.doSomething();
  }
}
```

# Q. What is the difference between BeanFactory and ApplicationContext? V. IMP.

- **Similarity:** BeanFactory and ApplicationContext are used as containers for managing and accessing beans.



```java
public static void main(String[] args) {
  ClassPathResource resource = new
                ClassPathResource(“beans.xml");

  BeanFactory beanFactory = new XmlBeanFactory(resource);

  MyService myService =
       (MyService)beanFactory.getBean("myService");

  mySservice.doSomething(); // Use the retrieved bean
}
```

```java
public static void main(String[] args) {
// Create a Spring application context (IoC container)
  try (
    AnnotationConfigApplicationContext context = new
       AnnotationConfigApplicationContext(AppConfig.class)
  ) {
   // Retrieve the bean from the context
    School school = context.getBean(School.class);
    school.displayStudentCount(); // Use the bean
  }
}
```

back to chapter index

# Q. What is the difference between BeanFactory and ApplicationContext? V. IMP.

| BeanFactory | ApplicationContext |
|---|---|
| 1. Lazy initialization of beans | Eager initialization of beans |
| 2. Limited support | Full support with message sources, transaction management etc 🏆 |
| 3. Minimal integration | Integrates with Spring's AOP features 🏆 |
| 4. Simple resource loading | Good mechanism for loading resources (e.g., classpath, file system, URL) 🏆 |
| 5. Suitable for simple & small applications | Preferred for enterprise & big applications 🏆 |

```
public static void main(String[] args) {
  ClassPathResource resource = new
                 ClassPathResource(“beans.xml");

  BeanFactory beanFactory = new XmlBeanFactory(resource);

  MyService myService =
       (MyService)beanFactory.getBean("myService");

  myService.doSomething(); // Use the retrieved bean
}
```

```
public static void main(String[] args) {
// Create a Spring application context (IoC container)
  try (
    AnnotationConfigApplicationContext context = new
       AnnotationConfigApplicationContext(AppConfig.class)
  ) {
   // Retrieve the bean from the context
    School school = context.getBean(School.class);
    school.displayStudentCount(); // Use the bean
  }
}
```

back to chapter index

# Spring Boot - Basics

Q. What is Spring Boot? How does it differ from the traditional Spring Framework? **V. IMP.**

Q. What is Spring Initializr?

Q. What is Spring Boot starter?

Q. What are the types of Spring Boot starter?

Q. How Spring Boot provides auto configuration? **V. IMP.**

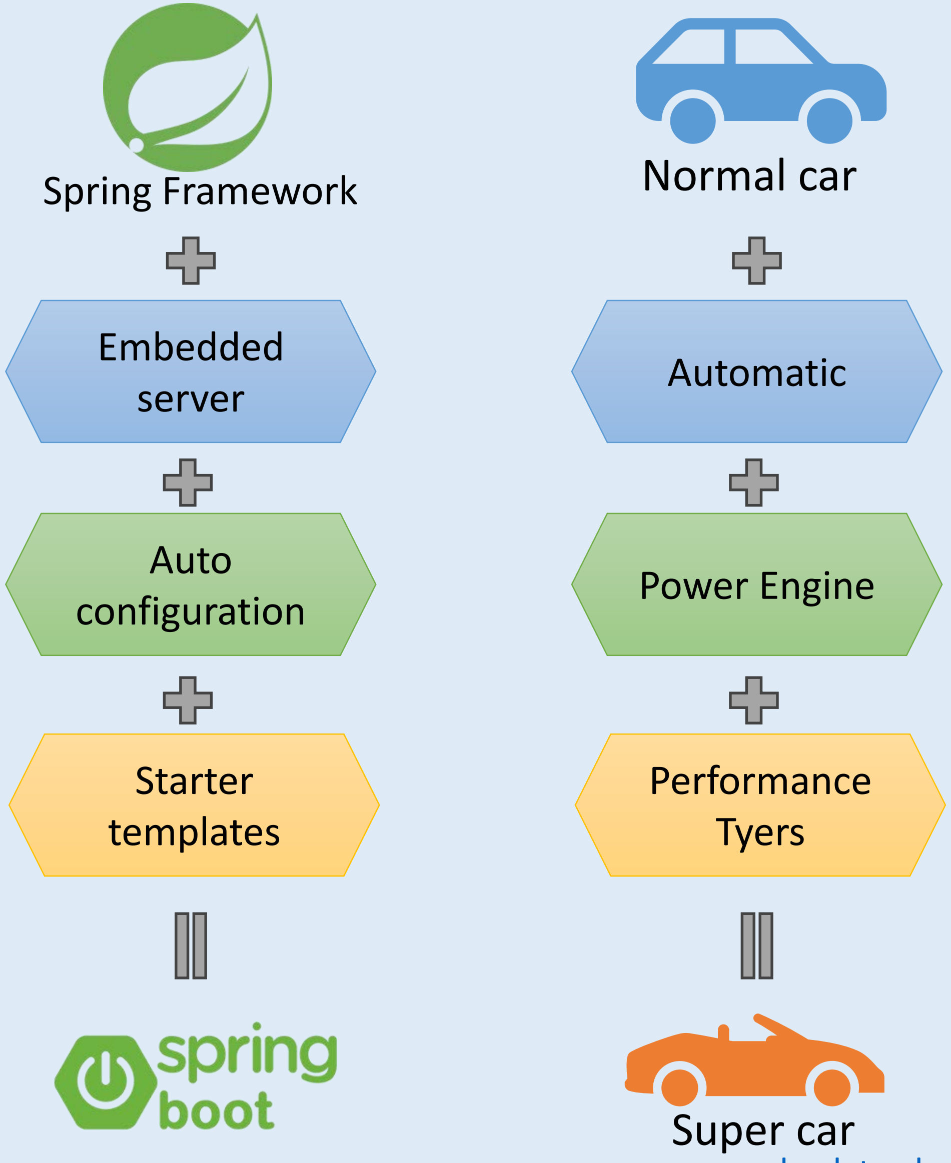
Q. What is the role of @SpringBootApplication annotation? **V. IMP.**

Q. What are embedded servers in Spring Boot? What are its benefits?

Back to main index

# Q. What is Spring Boot? How does it differ from the traditional Spring Framework? V. IMP.

- **Spring Boot** is an open-source framework that is built on top of the Spring Framework.
- Spring Boot allows developers to quickly create stand-alone, **production-ready** Spring applications with minimal setup and configuration.



Spring Framework

Embedded server

Auto configuration

Starter templates

Normal car

Automatic

Power Engine

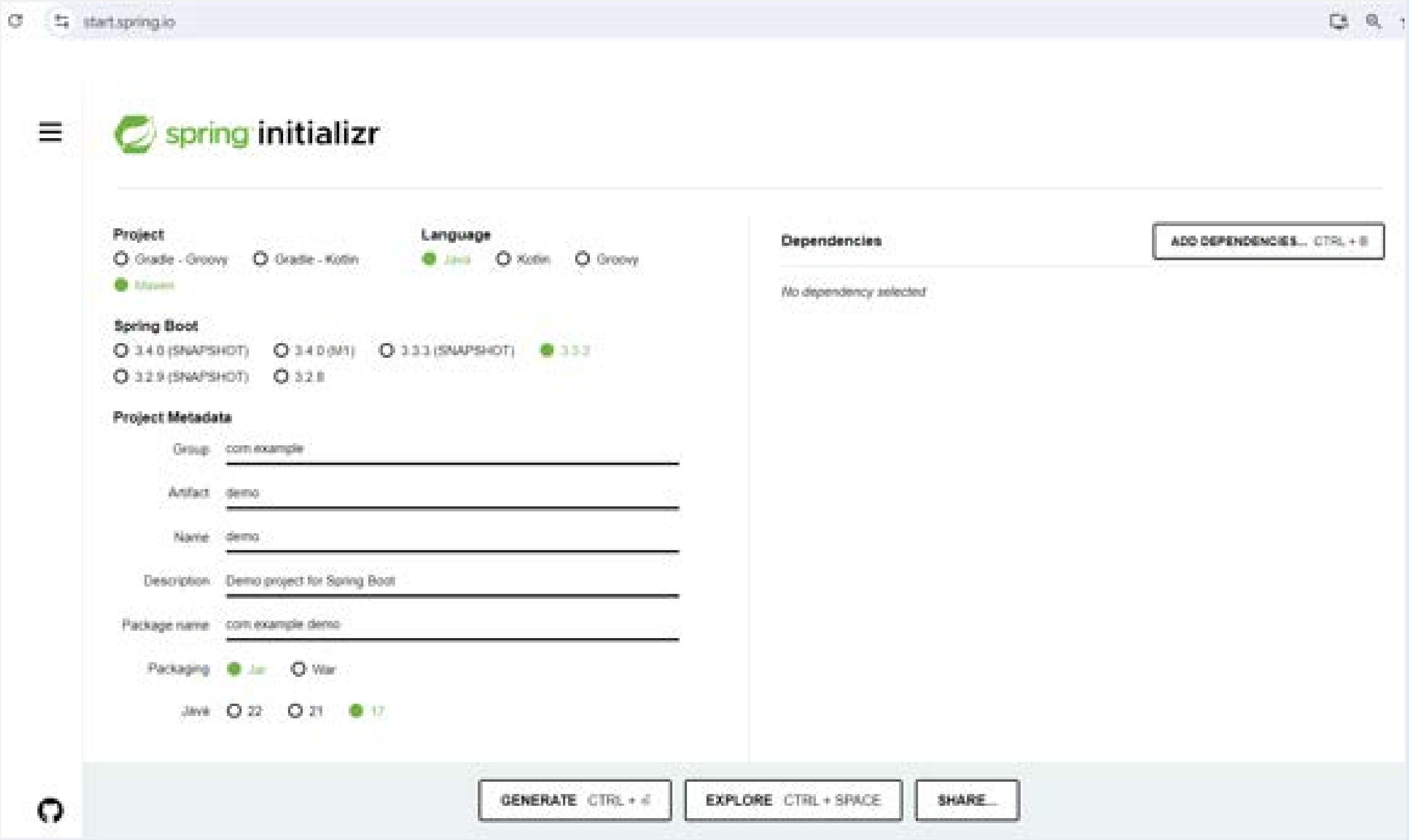
Performance Tyers

Super car

back to chapter index

# Q. What is Spring Initializr?

- Spring Initializr is a web-based tool provided by the Spring team to simplify the process of creating a new Spring Boot project.



**Project**
○ Gradle - Groovy ○ Gradle - Kotlin
● Maven

**Language**
● Java ○ Kotlin ○ Groovy

**Spring Boot**
○ 3.4.0 (SNAPSHOT) ○ 3.4.0 (M1) ○ 3.3.3 (SNAPSHOT) ● 3.3.2
○ 3.2.9 (SNAPSHOT) ○ 3.2.8

**Project Metadata**

Group: com.example

Artifact: demo

Name: demo

Description: Demo project for Spring Boot

Package name: com.example.demo

Packaging: ● Jar ○ War

Java: ○ 22 ○ 21 ● 17

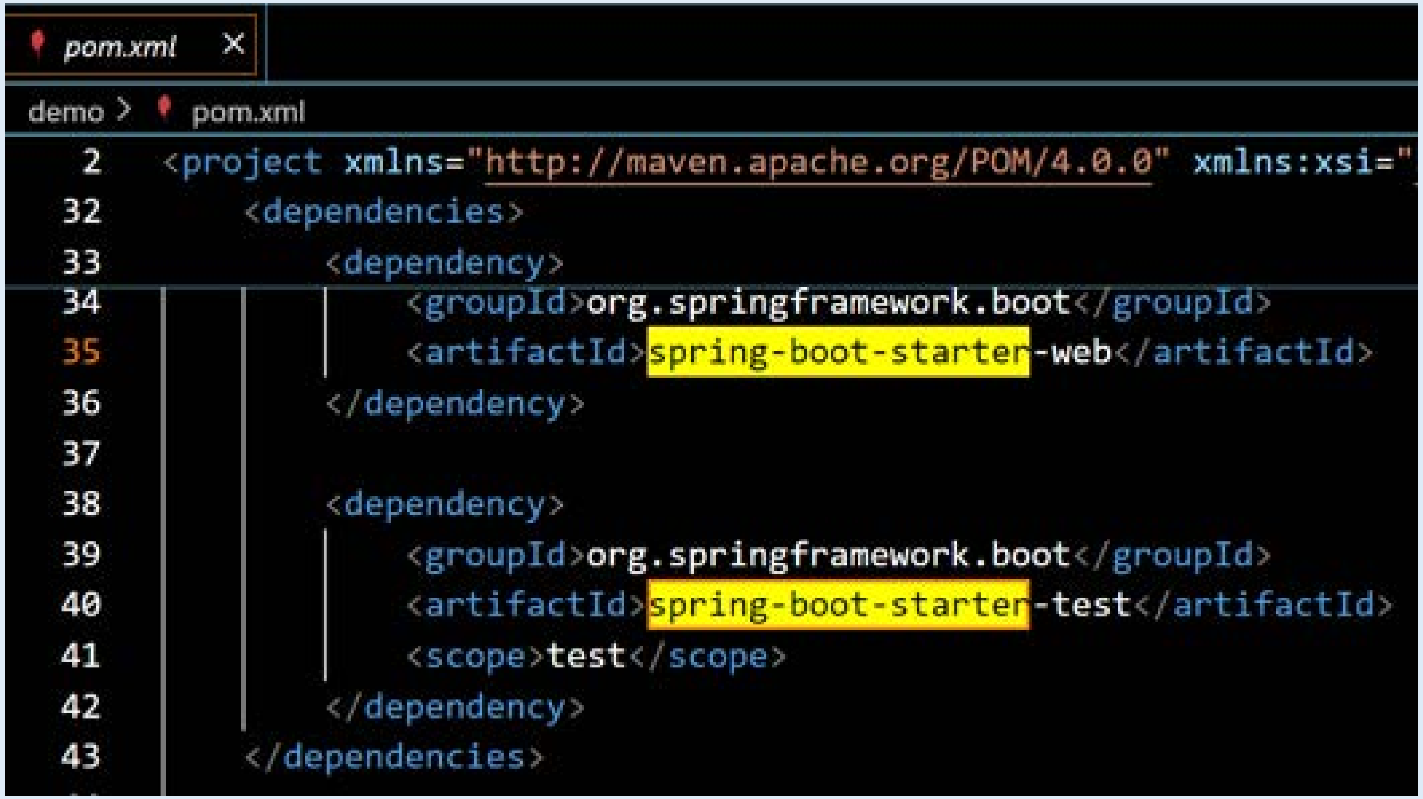
**Dependencies**

*No dependency selected*

back to chapter index

# Q. What is Spring Boot starter?

- A Spring Boot Starter is a **set of dependencies** that you can include in your Spring Boot project to quickly add specific functionalities.



```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>
```

back to chapter index

# Q. What are the types of Spring Boot starter?

**1. spring-boot-starter**

The core starter, including auto-configuration support, logging, and YAML.

**2. spring-boot-starter-web**

Starter for building web applications using Spring MVC. Includes RESTful application support.

**3. spring-boot-starter-data-jpa**

Starter for using Spring Data JPA with Hibernate.

**4. spring-boot-starter-test**

Starter for testing Spring Boot applications with libraries including JUnit, Hamcrest, and Mockito.

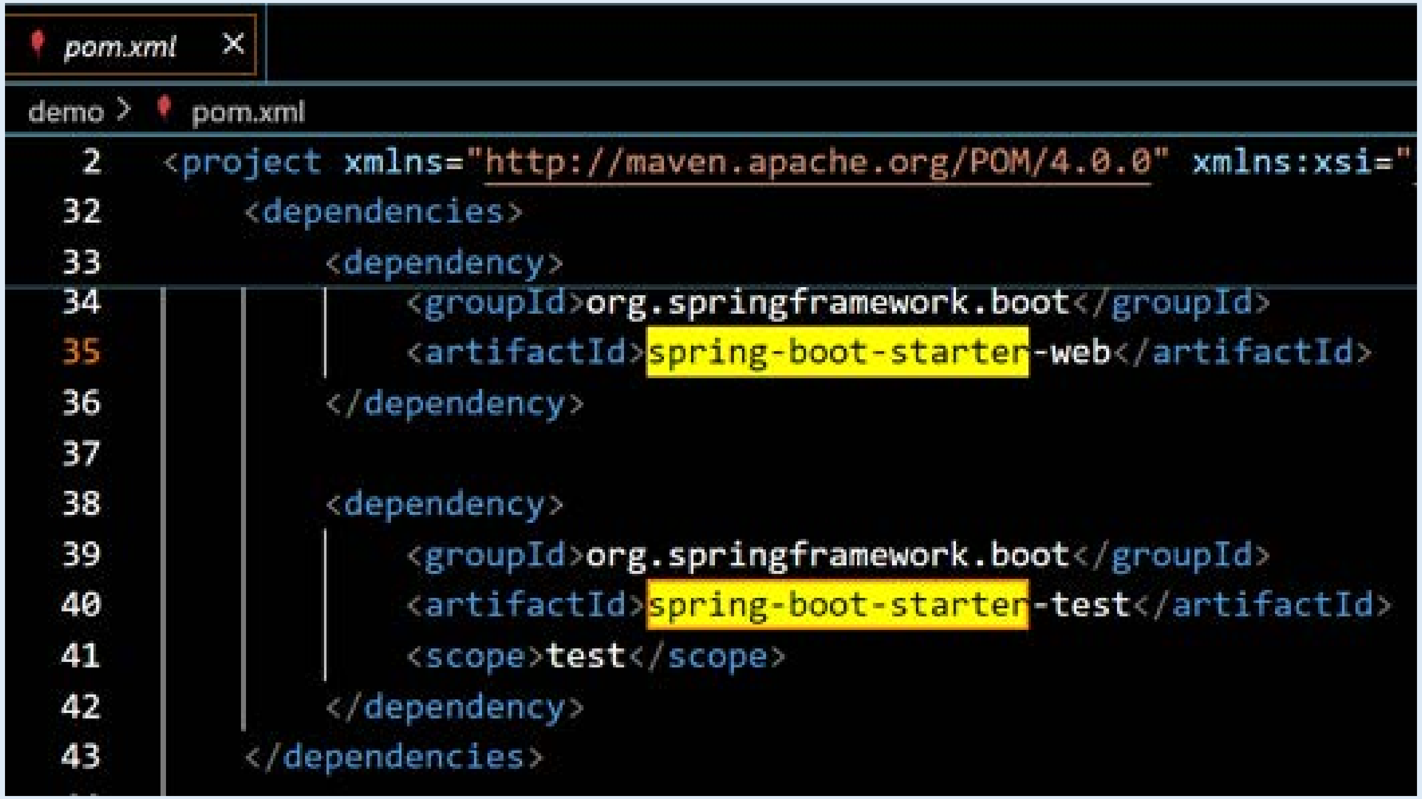
**5. spring-boot-starter-security**

Starter for using Spring Security.

back to chapter index

# Q. How Spring Boot provides auto configuration? V. IMP.

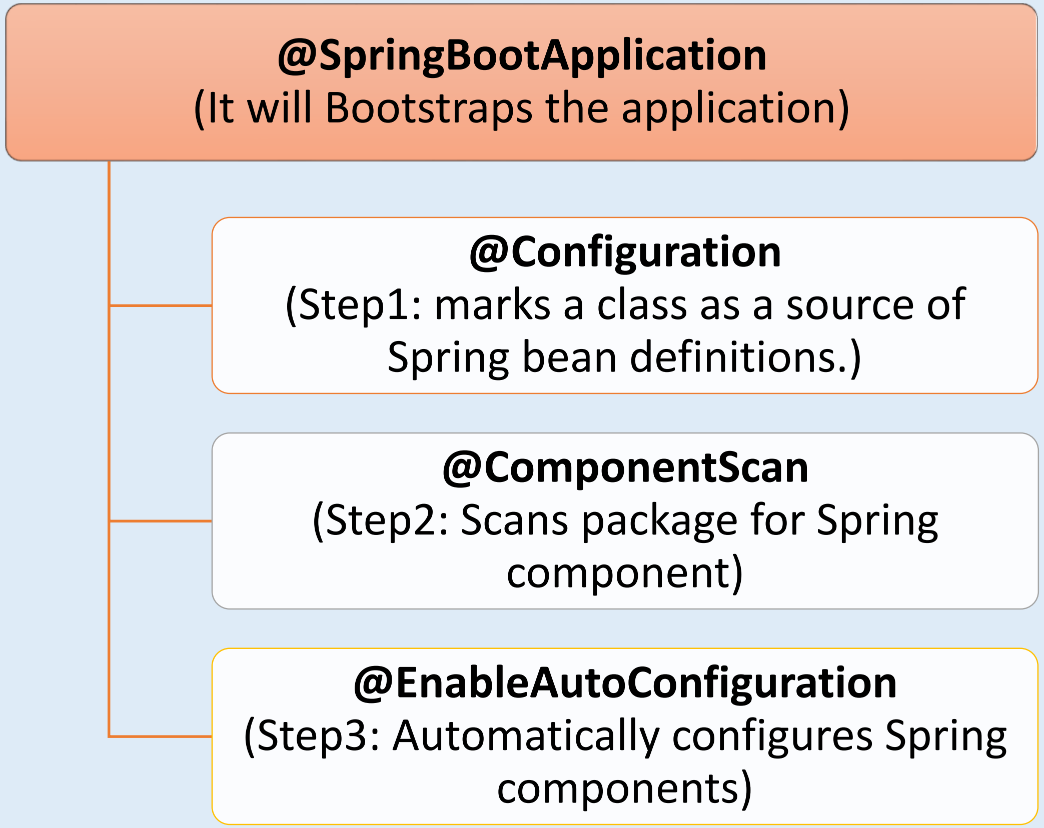
- Spring Boot provides auto-configuration by automatically configuring Spring applications based on the dependencies present on the classpath, reducing the need for manual configuration and allowing developers to focus on writing business logic.



```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>
```

back to chapter index

# Q. What is the role of @SpringBootApplication annotation? V. IMP.

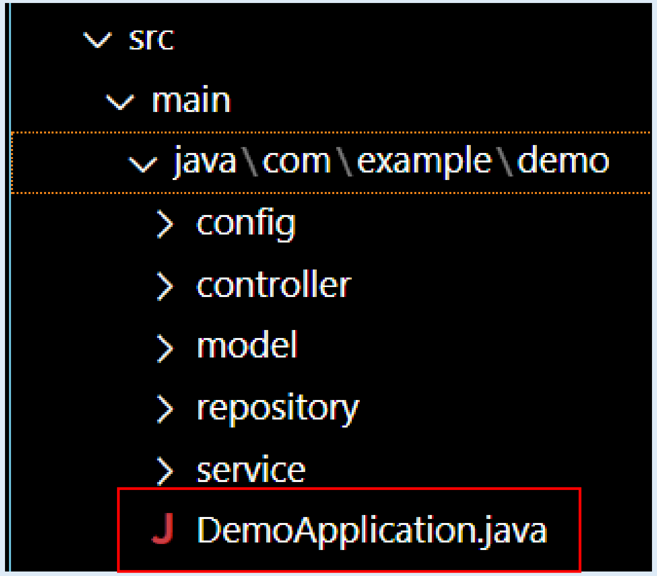


**@SpringBootApplication**
(It will Bootstraps the application)

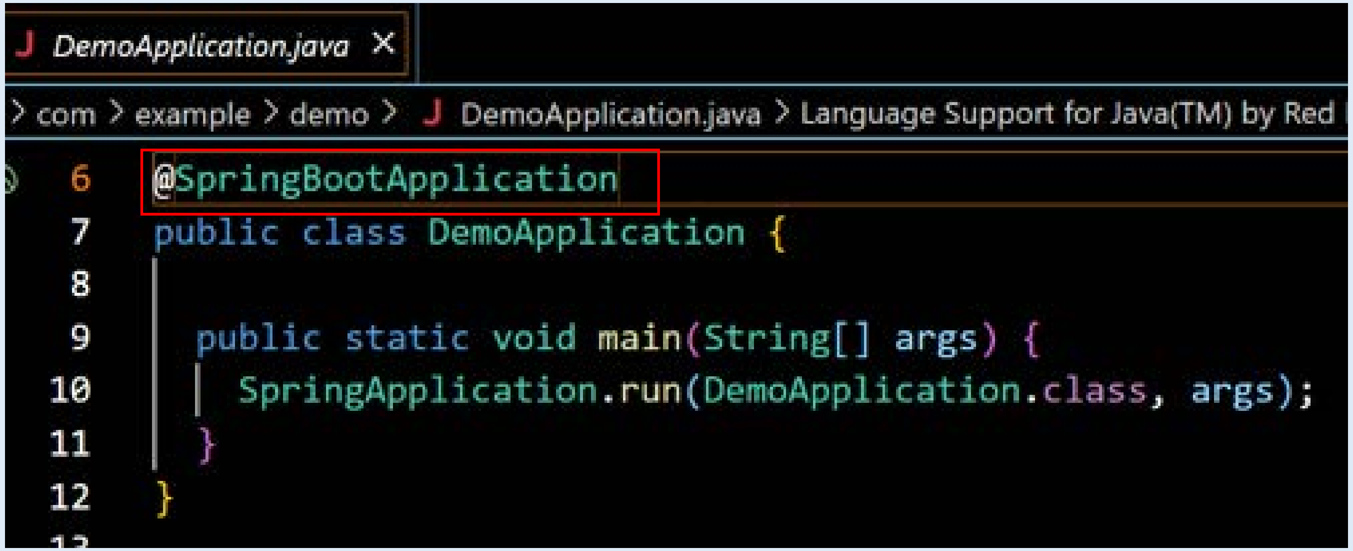
**@Configuration**
(Step1: marks a class as a source of Spring bean definitions.)

**@ComponentScan**
(Step2: Scans package for Spring component)

**@EnableAutoConfiguration**
(Step3: Automatically configures Spring components)



```
v src
  v main
    v java\com\example\demo
      > config
      > controller
      > model
      > repository
      > service
      J DemoApplication.java
```



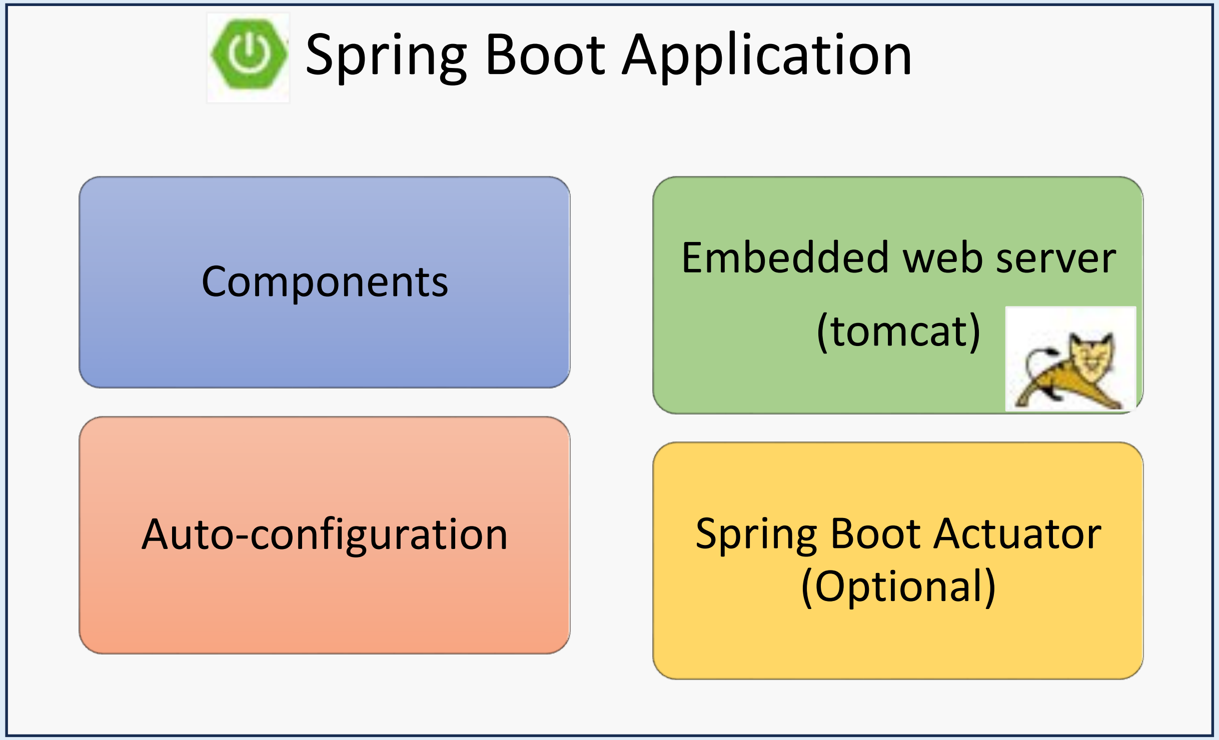
J DemoApplication.java ✕
> com > example > demo > J DemoApplication.java > Language Support for Java(TM) by Red

```java
@SpringBootApplication
public class DemoApplication {

  public static void main(String[] args) {
    SpringApplication.run(DemoApplication.class, args);
  }
}
```

back to chapter index

# Q. What are embedded servers in Spring Boot? What are its benefits?

- In Spring Boot, embedded servers refer to web servers (e.g., Tomcat, Jetty) that are **bundled within your application** as part of the build process.
- Benefits of embedded server: The application can run on any machine with a JRE without requiring a pre-installed web server, which simplified the development.



# Spring Boot - Project structure, Configuration & Actuator

Q. What is the project structure of a Spring Boot MVC application?

Q. What is application.properties used for in Spring Boot?

Q. What is application.yml file? Difference btw application.yml & application.properties? **V. IMP.**

Q. How do you configure a Spring Boot application?

Q. What is Spring Boot Actuator? **V. IMP.**

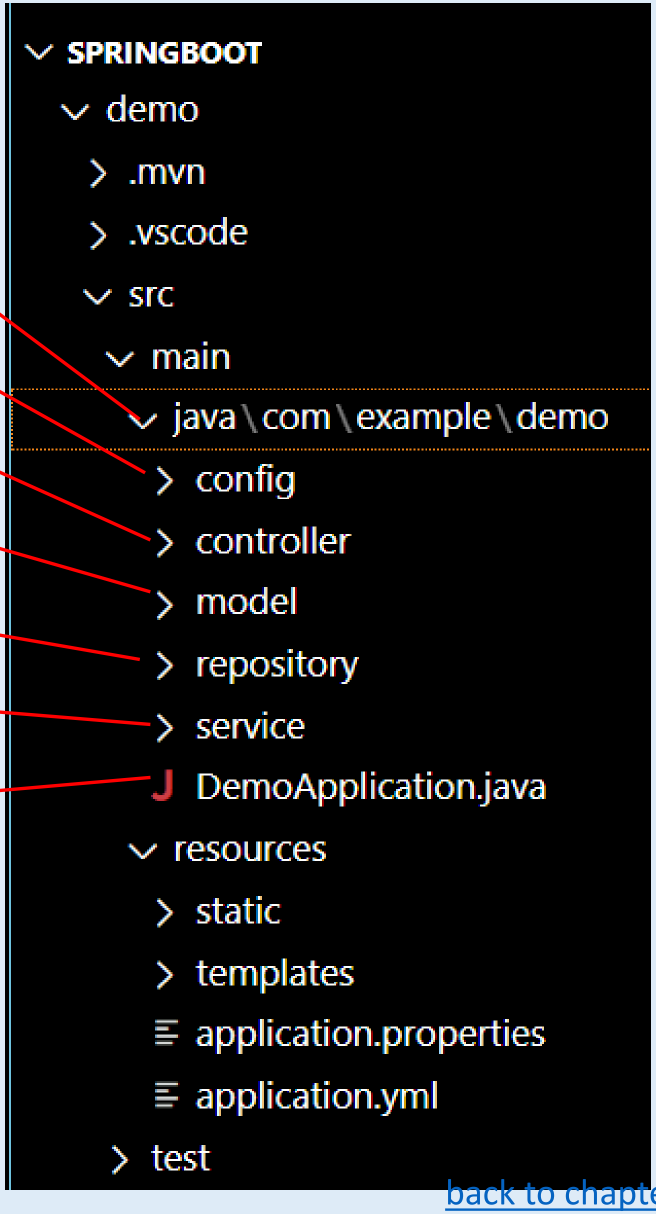
Q. What are the key features of Spring Boot Actuator?

Q. How do you package and deploy a Spring Boot application?

Back to main index

# Q. What is the project structure of a Spring Boot MVC application?

- **src/main/java/:** Contains the main source code of your application.
  - **config/(@Configuration):** Contains configuration classes.
  - **controller/(@RestController or @Controller):** Contains controllers which handle HTTP requests.
  - **model/(@Entity):** Contains entity classes that represent the data model.
  - **repository/(@Repository):** Contains repository interfaces which are used to interact with the database.
  - **service/(@Service):** Contains service classes which contains business logic.
  - **Main class(@SpringBootApplication):** The main class is the entry point for the Spring Boot application.

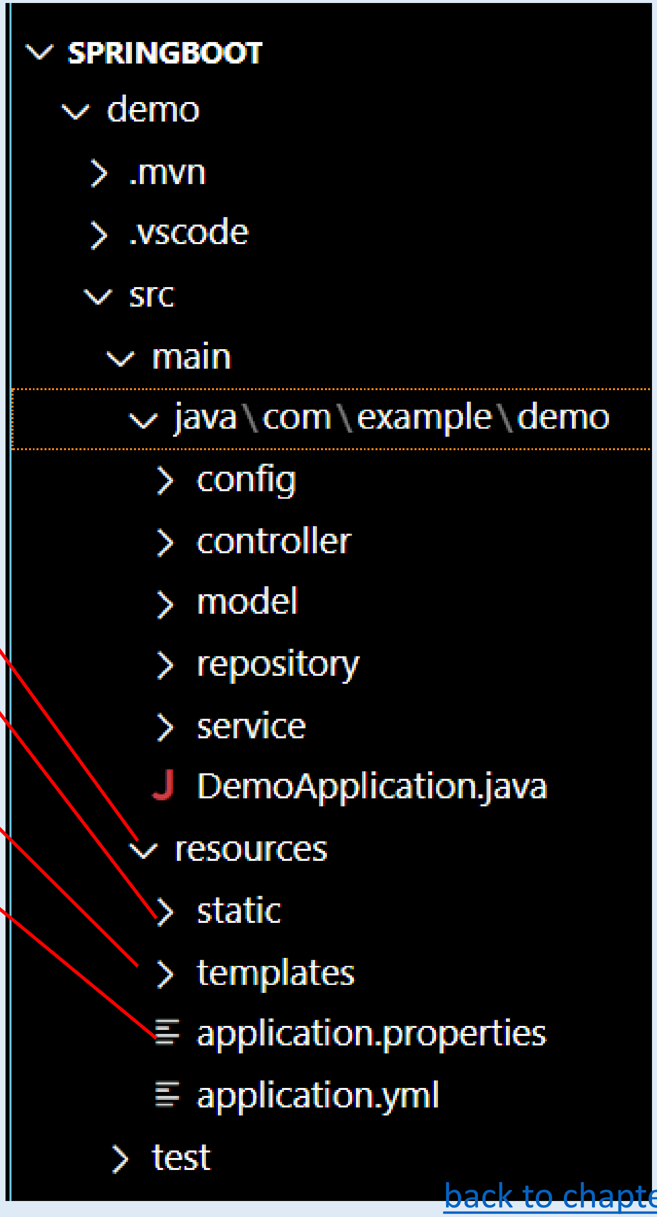


```
∨ SPRINGBOOT
  ∨ demo
    > .mvn
    > .vscode
    ∨ src
      ∨ main
        ∨ java \ com \ example \ demo
          > config
          > controller
          > model
          > repository
          > service
          J DemoApplication.java
        ∨ resources
          > static
          > templates
          ≡ application.properties
          ≡ application.yml
      > test
```

back to chapter index

# Q. What is the project structure of a Spring Boot MVC application?

- **src/main/resources/**: Contains static resources, templates, and configuration files.
  - **static/**: Used for static assets like CSS, JavaScript, and images. These files are served directly by the web server.
  - **templates/:** Contains view templates (e.g., Thymeleaf, FreeMarker) used for server-side rendering of web pages.
  - **application.properties and application.yml:** Configuration files for the application.

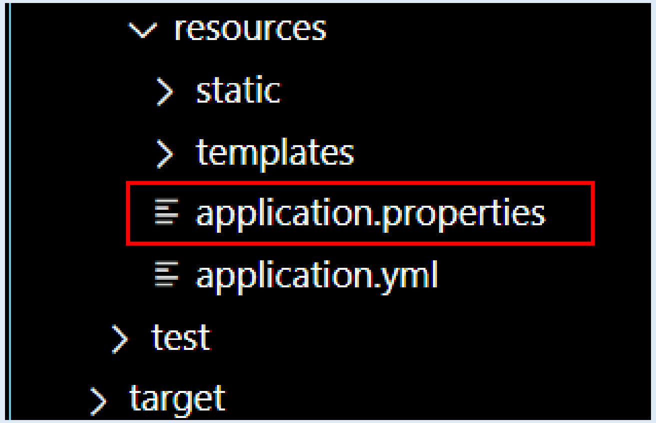


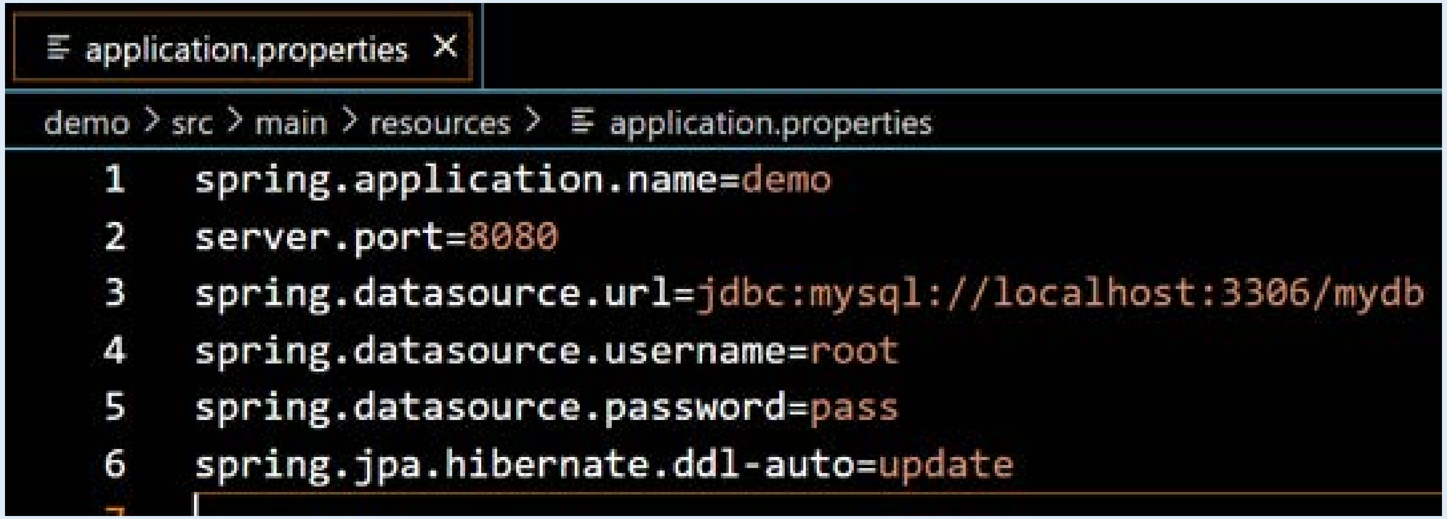
```
∨ SPRINGBOOT
  ∨ demo
    > .mvn
    > .vscode
    ∨ src
      ∨ main
        ∨ java\com\example\demo
          > config
          > controller
          > model
          > repository
          > service
          J DemoApplication.java
        ∨ resources
          > static
          > templates
          application.properties
          application.yml
      > test
```

back to chapter index

# Q. What is application.properties used for in Spring Boot?

- application.properties is a configuration files used to define various **settings** in the form of **key value pairs.**
- Used for defining database connections, application server settings, port etc.
- This is ideal for simpler applications with straightforward configurations.





```
spring.application.name=demo
server.port=8080
spring.datasource.url=jdbc:mysql://localhost:3306/mydb
spring.datasource.username=root
spring.datasource.password=pass
spring.jpa.hibernate.ddl-auto=update
```

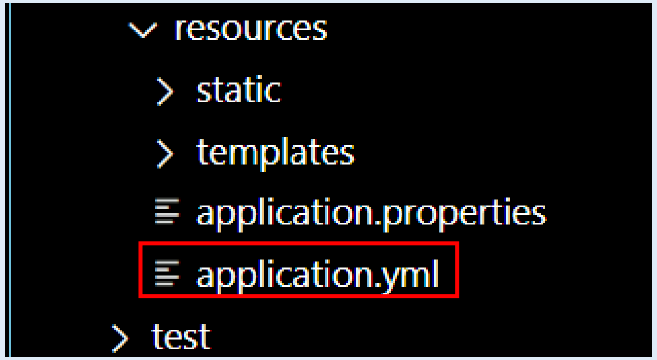
back to chapter index

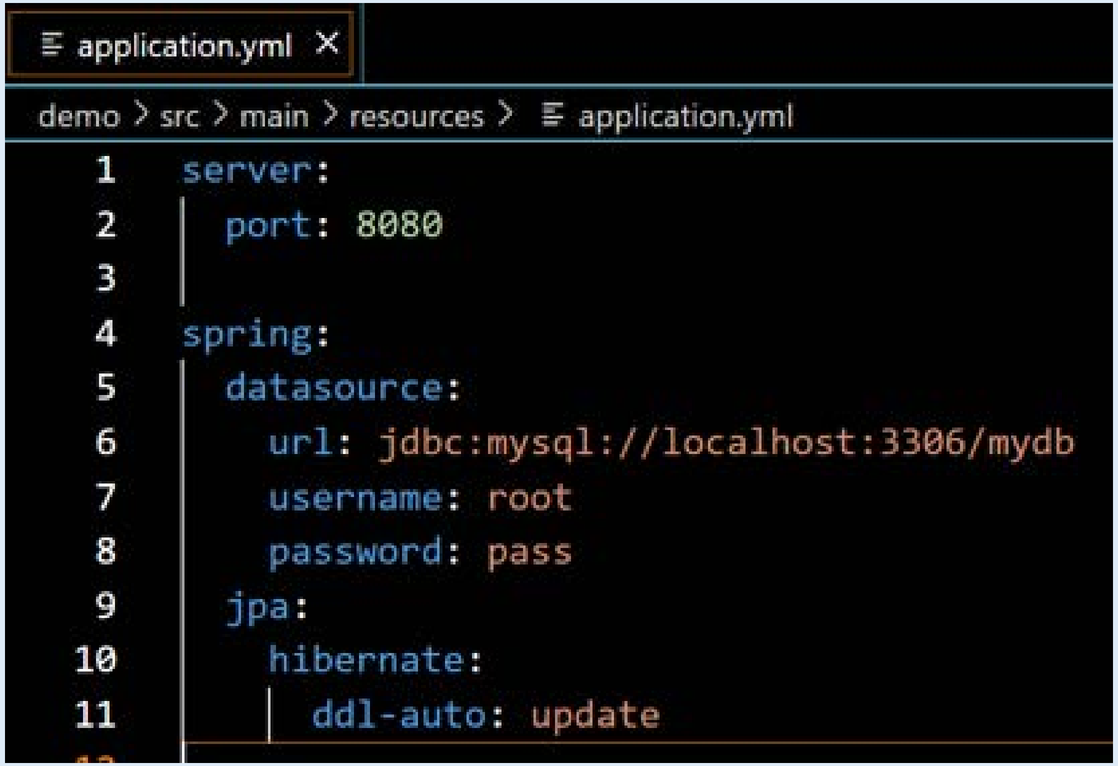
# Q. What is application.yml file? Difference btw application.yml & application.properties? V. IMP.

- application.yml is a configuration files used to define various **settings** in the form of **hierarchical and nested structure.**

- Used for defining database connections, application server settings, port etc.

- This is ideal for more complex applications with many nested configurations, as the YAML format is more readable then.





```yaml
server:
  port: 8080

spring:
  datasource:
    url: jdbc:mysql://localhost:3306/mydb
    username: root
    password: pass
  jpa:
    hibernate:
      ddl-auto: update
```

[back to chapter index](#)

# Q. How do you configure a Spring Boot application?

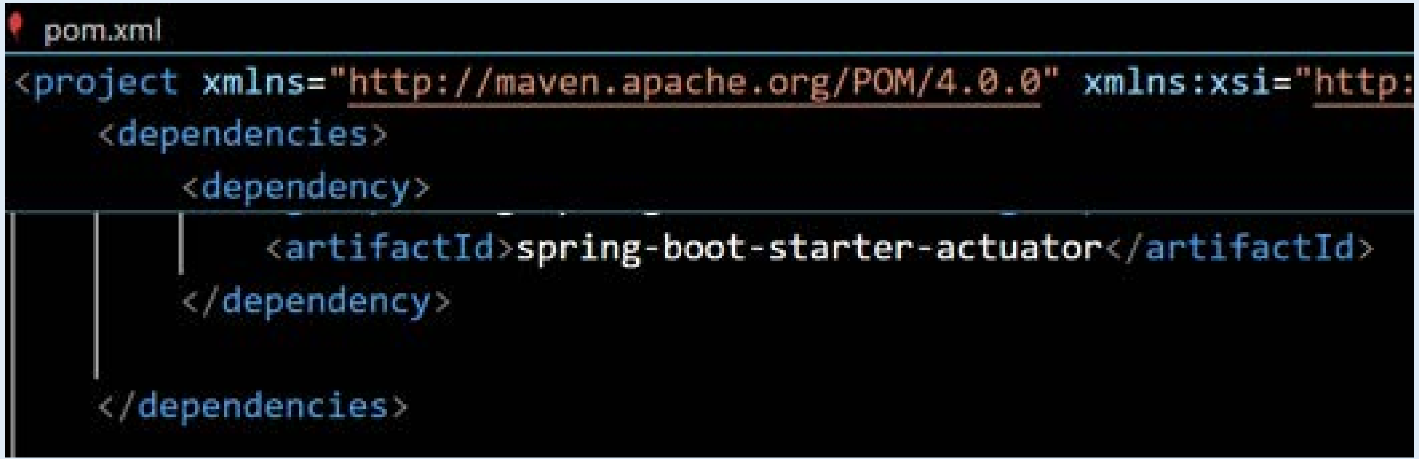
❖ Two ways to configure a Spring Boot application:

1. application.properties
2. application.yml

back to chapter index

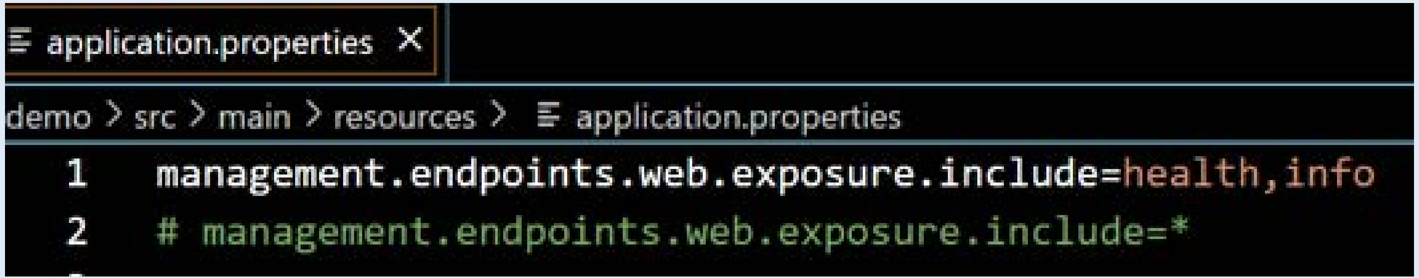
# Q. What is Spring Boot Actuator? V. IMP.

- Spring Boot Actuator is a subproject of Spring boot that provides production-ready features for **monitoring and managing** Spring Boot applications through various endpoints.



pom.xml

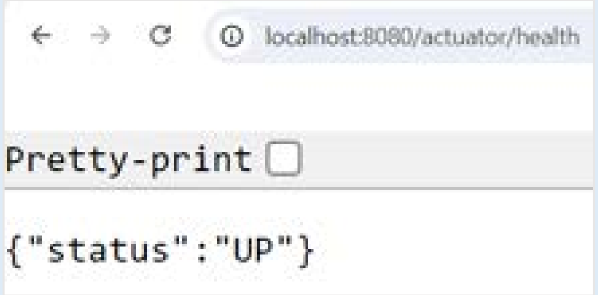
```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http:
    <dependencies>
        <dependency>
            <artifactId>spring-boot-starter-actuator</artifactId>
        </dependency>

    </dependencies>
```



application.properties

demo > src > main > resources > application.properties

```
management.endpoints.web.exposure.include=health,info
# management.endpoints.web.exposure.include=*
```



localhost:8080/actuator/health

Pretty-print □

```
{"status":"UP"}
```

back to chapter index

# Q. What are the key features of Spring Boot Actuator?

**1. System Metrics:**
JVM Metrics: Memory usage, garbage collection, threads, class loading.

**2. Application Metrics:**
HTTP Metrics: Request count, response times, status codes.

**3. Health Metrics:**
Health Checks: Application health status, custom health indicators (e.g., database connectivity, disk space).

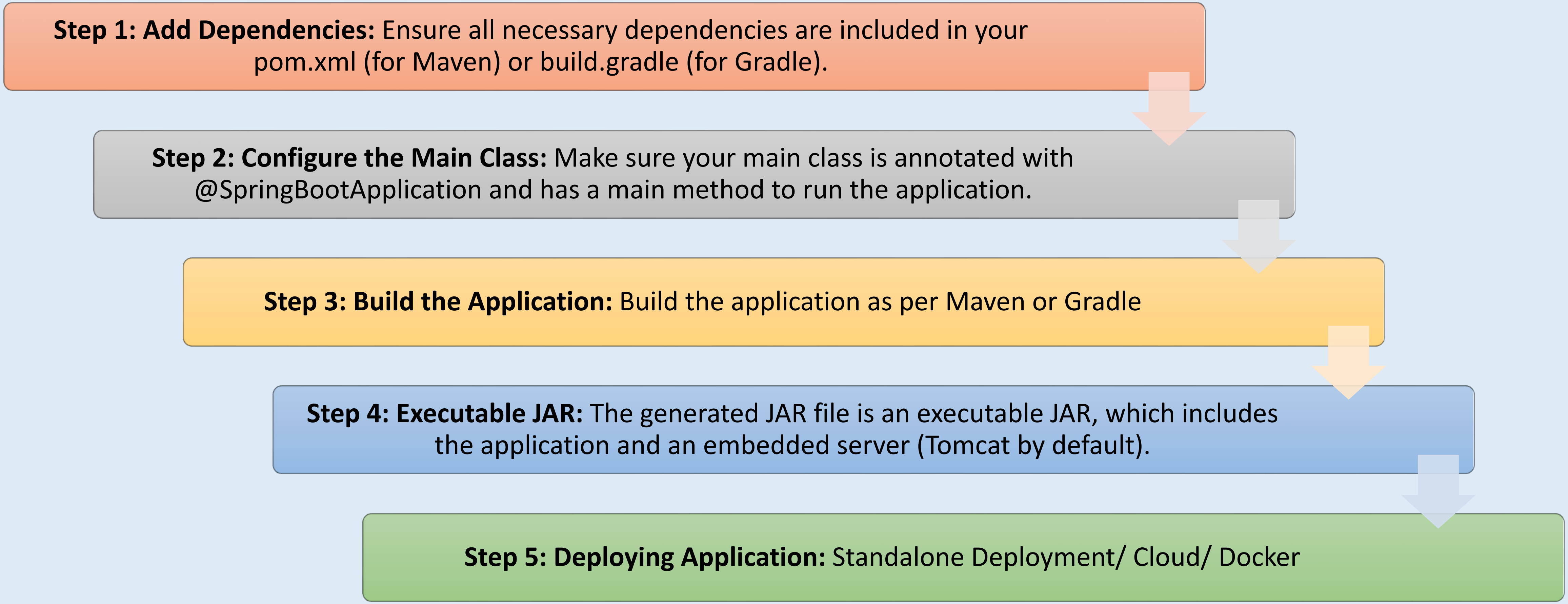
**4. Environment Metrics:**
Configuration Properties: Active profiles, configuration properties.

**5. Logging Metrics:**
Loggers: Logging levels and changes.

**6. Application Info:**
Build Information: Application version, build time.

back to chapter index

# Q. How do you package and deploy a Spring Boot application?



**Step 1: Add Dependencies:** Ensure all necessary dependencies are included in your pom.xml (for Maven) or build.gradle (for Gradle).

**Step 2: Configure the Main Class:** Make sure your main class is annotated with @SpringBootApplication and has a main method to run the application.

**Step 3: Build the Application:** Build the application as per Maven or Gradle

**Step 4: Executable JAR:** The generated JAR file is an executable JAR, which includes the application and an embedded server (Tomcat by default).

**Step 5: Deploying Application:** Standalone Deployment/ Cloud/ Docker

back to chapter index

# Spring MVC - Basics

Q. What is Spring MVC? Explain the architecture of Spring MVC? **V. IMP.**

Q. What does MVC stands for?

Q. What is the role of Dispatcher Servlet in Spring MVC?

Q. What is Controller and @Controller annotation in Spring MVC?

Q. What is the role of @RequestMapping annotation in Spring MVC? **V. IMP.**

Q. What is the role of @GetMapping annotation in Spring MVC?

Q. What is the difference between @PostMapping & @PutMapping annotation? **V. IMP.**

Q. What is the role of @DeleteMapping annotation in Spring MVC?

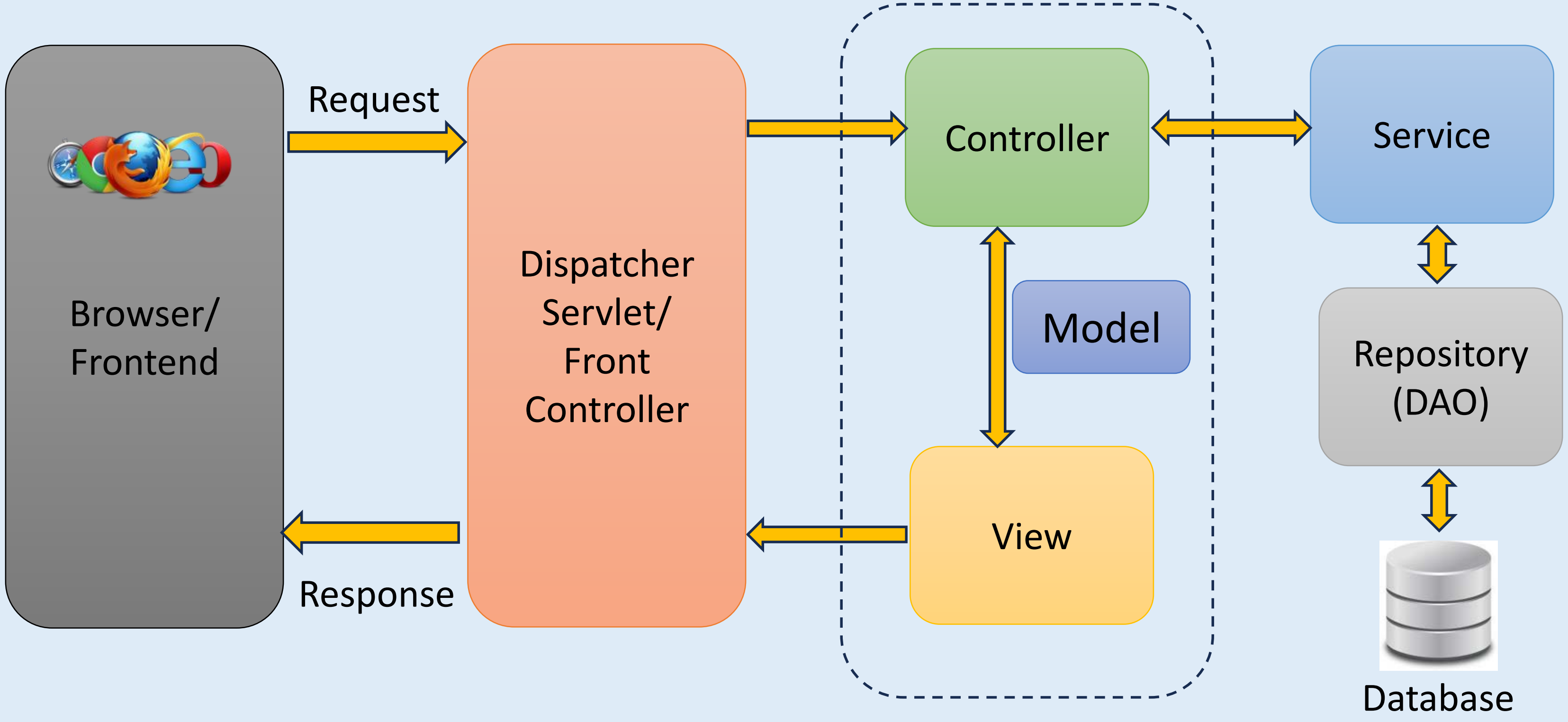
Q. What is the difference between @Controller and @RestController annotations?

Q. What is the role of Model in Spring MVC?

Back to main index

# Q. What is Spring MVC? Explain the architecture of Spring MVC? V. IMP.

- Spring MVC is a module within the Spring framework that simplifies web application development by using MVC(Model View Controller) pattern.



Spring MVC Architecture

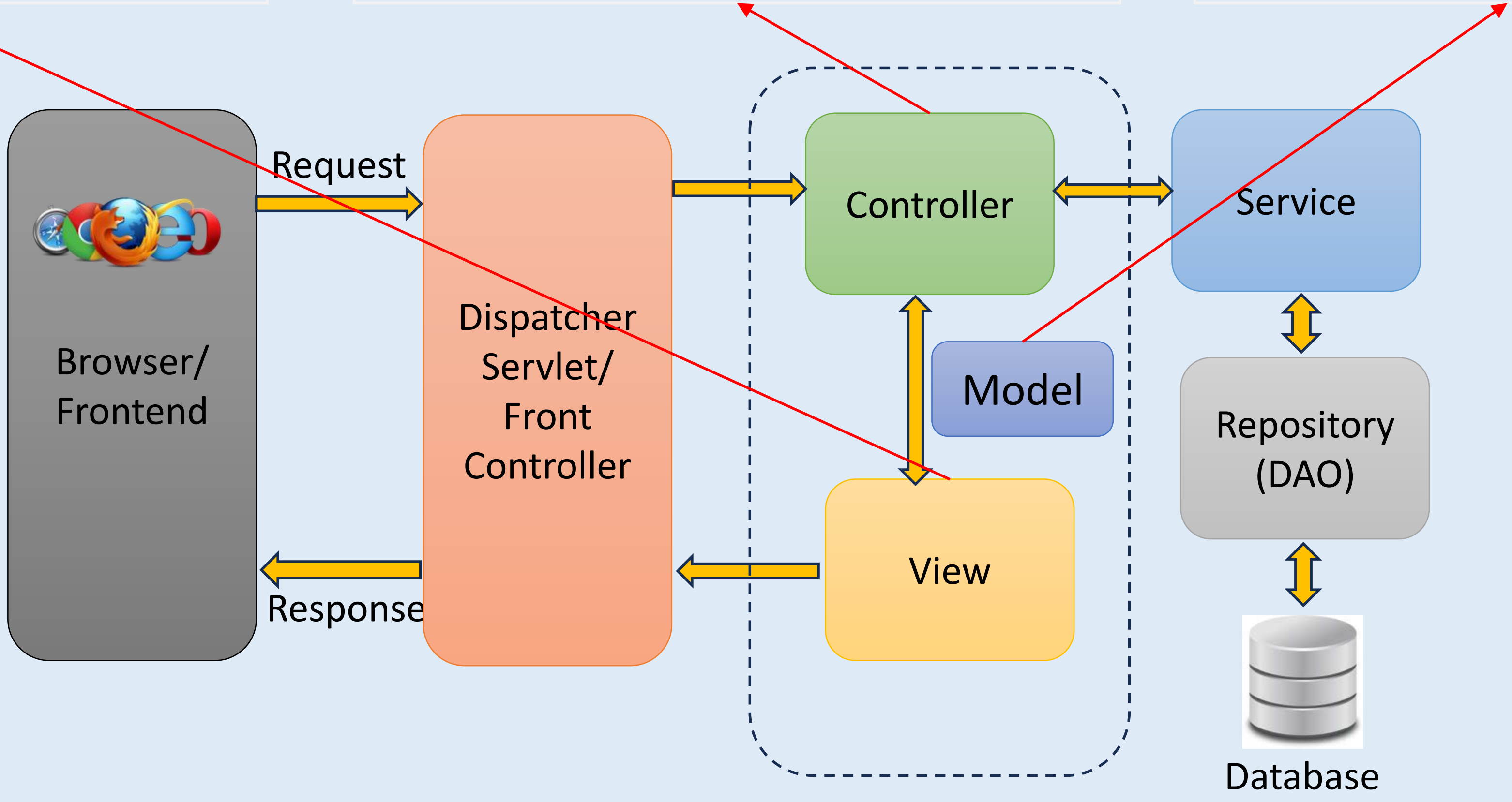
back to chapter index

# Q. What does MVC stands for?

- The **View** renders the model(data) and generates the response to be sent to the client.

- The **Controllers** handles incoming requests and invokes appropriate service(business logic) to prepare data(model) for rendering.

- The **Model** represents the data. It is used to hold data and transfer it between View and Controller.

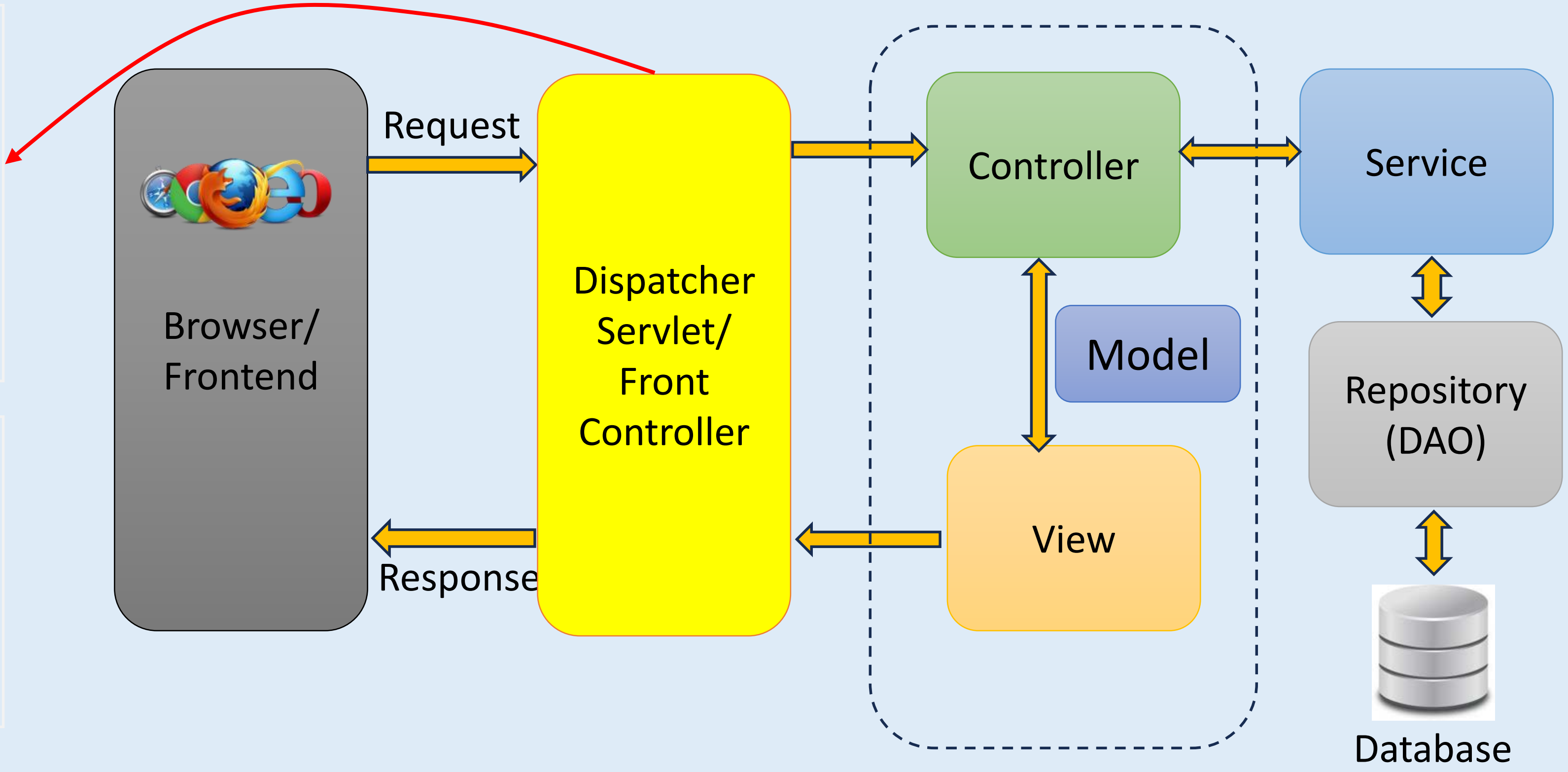


back to chapter index

# Q. What is the role of Dispatcher Servlet in Spring MVC?

- Dispatcher Servlet acts as the front controller responsible for receiving incoming requests and dispatching them to the appropriate controllers for processing based on the request URL match.

- In Spring we configure Dispatcher Servlet but because of auto configuration, Spring Boot automatically sets up the DispatcherServlet.



back to chapter index

# Q. What is the role of Dispatcher Servlet in Spring MVC?

- AbstractAnnotationConfigDispatcherServletInitializer is an abstract base class provided by the Spring Framework that simplifies the configuration of the DispatcherServlet and other web-related settings using Java-based configuration instead of traditional web.xml configuration.

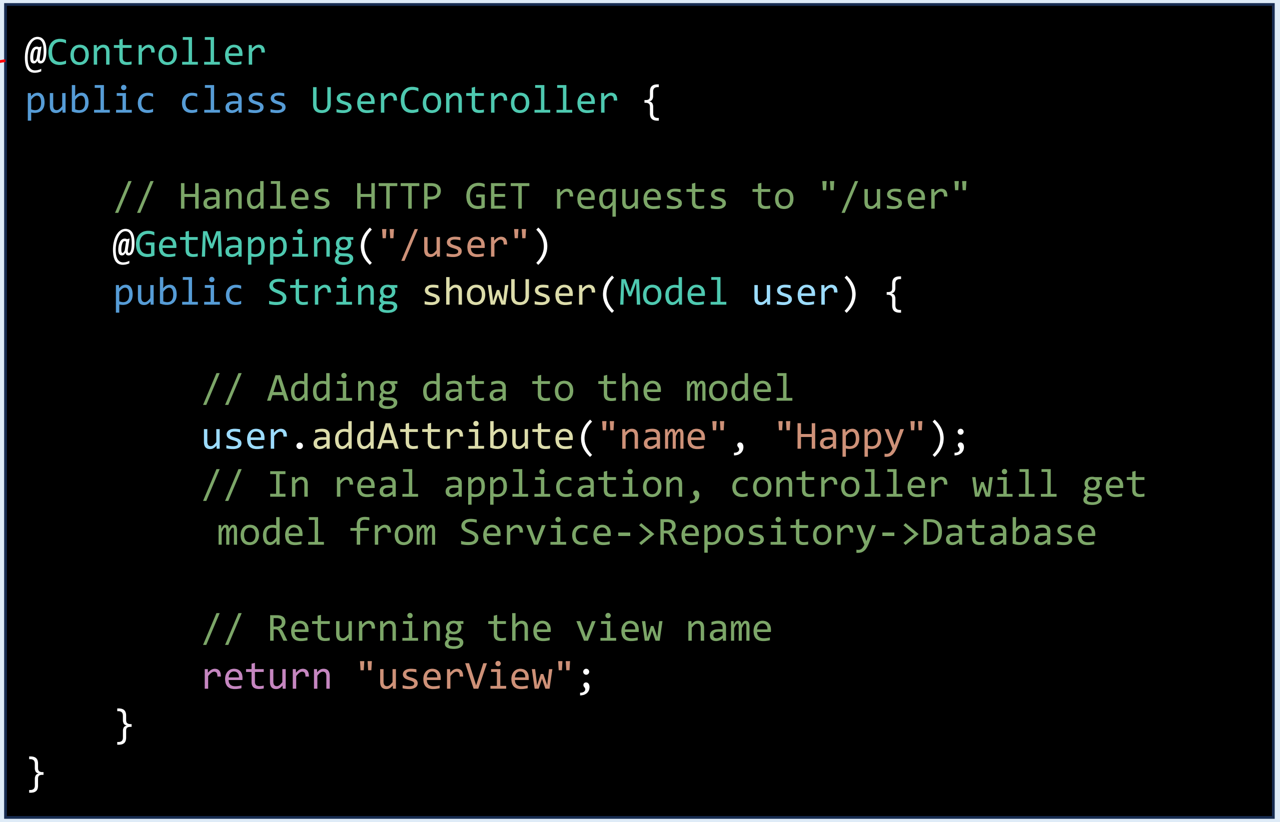
```java
public class MyWebAppInitializer extends
AbstractAnnotationConfigDispatcherServletInitializer {

    @Override
    protected Class<?>[] getRootConfigClasses() {
    // Root context configuration (e.g., services, repositories)
        return new Class[] { RootConfig.class };
    }

    @Override
    protected Class<?>[] getServletConfigClasses() {
    // Servlet context configuration (e.g., controllers, view
resolvers)
        return new Class[] { WebConfig.class };
    }

    @Override
    protected String[] getServletMappings() {
    // Map the DispatcherServlet to the root URL pattern ("/")
        return new String[] { "/" };
    }
}
```

back to chapter index

# Q. What is Controller and @Controller annotation in Spring MVC?

- @Controller annotation marks a class as a **web controller.**

- A controller is a key component that handles user requests and processes them to generate appropriate responses.



```
@Controller
public class UserController {

    // Handles HTTP GET requests to "/user"
    @GetMapping("/user")
    public String showUser(Model user) {

        // Adding data to the model
        user.addAttribute("name", "Happy");
        // In real application, controller will get
         model from Service->Repository->Database

        // Returning the view name
        return "userView";
    }
}
```

back to chapter index

# Q. What is the role of @RequestMapping annotation in Spring MVC? V. IMP.

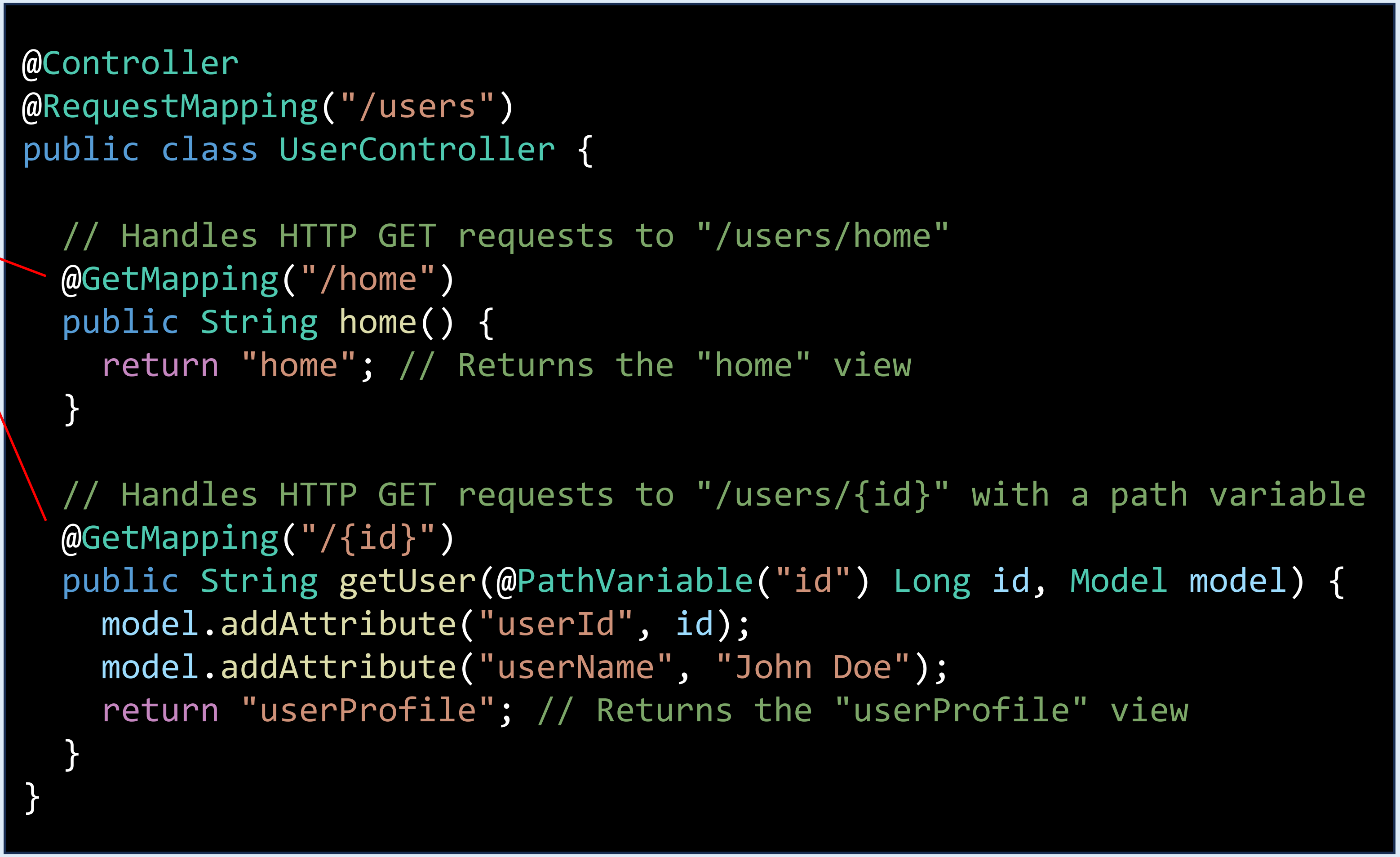
- @RequestMapping("/users") **defines a base URL** for all handler methods within the class.
- For example, in code base URL(“/users”) is applied to all methods in the controller.

```
@Controller
@RequestMapping("/users") // Base URL for all methods in this controller
public class UserController {

  // Handles HTTP GET requests to "/users/home"
  @GetMapping("/home")
  public String home() {
    return "home"; // Returns the "home" view
  }

  // Handles HTTP GET requests to "/users/{id}" with a path variable
  @GetMapping("/{id}")
  public String getUser(@PathVariable("id") Long id, Model model) {
    model.addAttribute("userId", id);
    model.addAttribute("userName", "John Doe");
    return "userProfile"; // Returns the "userProfile" view
  }
}
```

back to chapter index

# Q. What is the role of @GetMapping annotation in Spring MVC?

- @GetMapping annotation maps HTTP GET requests to specific handler methods, simplifying the process of **retrieving data** or rendering views.
- It’s a concise alternative to @RequestMapping(method = RequestMethod.GET).



```java
@Controller
@RequestMapping("/users")
public class UserController {

  // Handles HTTP GET requests to "/users/home"
  @GetMapping("/home")
  public String home() {
    return "home"; // Returns the "home" view
  }

  // Handles HTTP GET requests to "/users/{id}" with a path variable
  @GetMapping("/{id}")
  public String getUser(@PathVariable("id") Long id, Model model) {
    model.addAttribute("userId", id);
    model.addAttribute("userName", "John Doe");
    return "userProfile"; // Returns the "userProfile" view
  }
}
```

back to chapter index

# Q. What is the difference between @PostMapping & @PutMapping annotation? V. IMP.

- **@PostMapping("/submit"):** Handles POST requests for creating user data.

- **@PutMapping("/update/{id}"):** Handles PUT requests for updating user data.

```
@Controller
@RequestMapping("/users")
public class UserController {

  // Handles HTTP POST requests to "/users/submit" for form submissions
  @PostMapping("/submit")
  public String submitForm(@RequestParam("name") String name, Model
model) {
    model.addAttribute("name", name);
    // Handle form submission logic here
    return "result"; // Returns the "result" view
  }

  // Handles HTTP PUT requests to "/users/update/{id}" with a path
variable
  @PutMapping("/update/{id}")
  public String updateUser(
    @PathVariable("id") Long id,
    @RequestBody User user
  ) {
    // Update user logic here using the user object and id
    return "userUpdated"; // Returns the "userUpdated" view
  }
}
```

back to chapter index

# Q. What is the role of @DeleteMapping annotation in Spring MVC?

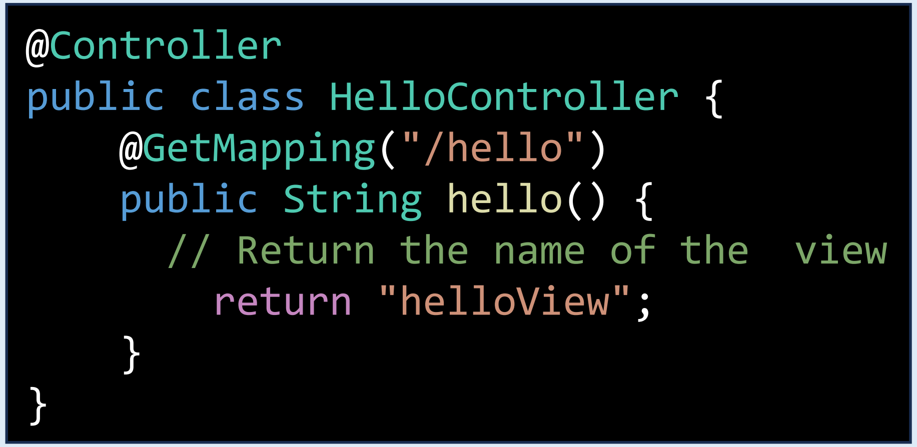
- **@DeleteMapping("/delete/{id}"):** Handles DELETE requests for deleting a user.

```java
@Controller
@RequestMapping("/users")
public class UserController {

  // Handles HTTP DELETE requests to "/users/delete/{id}"
  @DeleteMapping("/delete/{id}")
  public String deleteUser(@PathVariable("id") Long id) {

    // Delete user logic here using the id
    return "userDeleted"; // Returns the "userDeleted" view
  }
}
```

back to chapter index

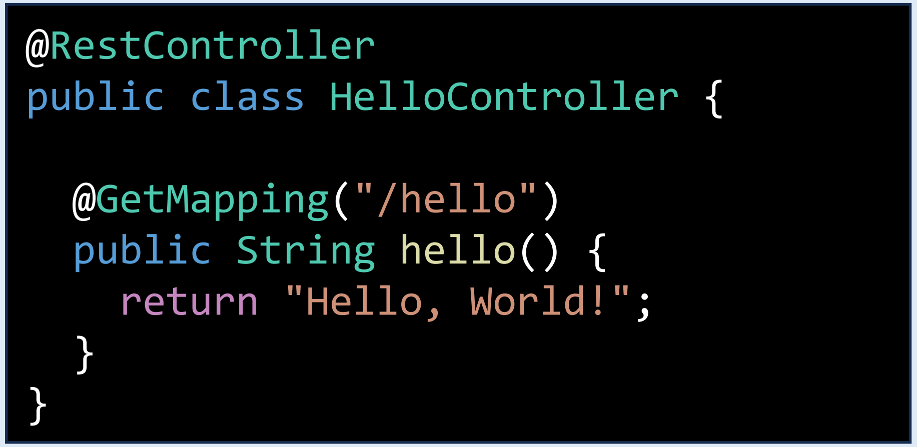
# Q. What is the difference between @Controller and @RestController annotations? V. IMP.

- @Controller is a general-purpose annotation used to define a controller that handles web requests. It is typically used in web applications where the response is HTML, JSP, or other types of views.



```java
@Controller
public class HelloController {
    @GetMapping("/hello")
    public String hello() {
      // Return the name of the  view
        return "helloView";
    }
}
```

- @RestController is a specialized version of the @Controller annotation which is used to create RESTful web services where the response is typically in JSON or XML format.

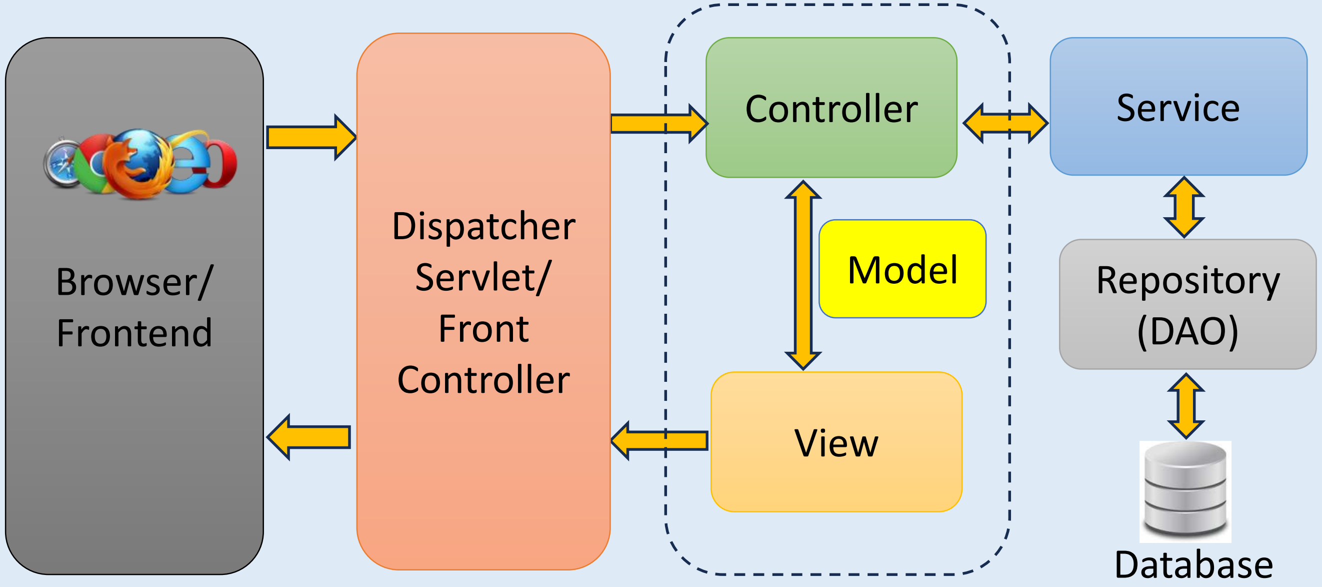


```java
@RestController
public class HelloController {

  @GetMapping("/hello")
  public String hello() {
    return "Hello, World!";
  }
}
```

back to chapter index

# Q. What is the role of Model in Spring MVC?

- In Spring MVC, the Model represents the data that is transferred between the controller and the view.



```java
// user model
public class user {
  private Long id;
  private String name;

  public user(Long id, String name) {
    this.id = id;
    this.name = name;
  }

  // Getters and Setters
  public Long getId() {
    return id;
  }
  public void setId(Long id) {
    this.id = id;
  }

  public String getName() {
    return name;
  }

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

back to chapter index

# Spring MVC - Important Annotations

Q. What is the difference between @RequestParam and @PathVariable? **V. IMP.**

Q. What is @ModelAttribute annotation in Spring MVC? **V. IMP.**

Q. What is @ModelAttribute annotation at method level? When to use it? **V. IMP.**

Q. What is the role of @ModelAttribute annotation with parameters?

Q. How to validate the data in Spring MVC? **V. IMP.**

Q. What is the purpose of ModelAndView in Spring MVC?

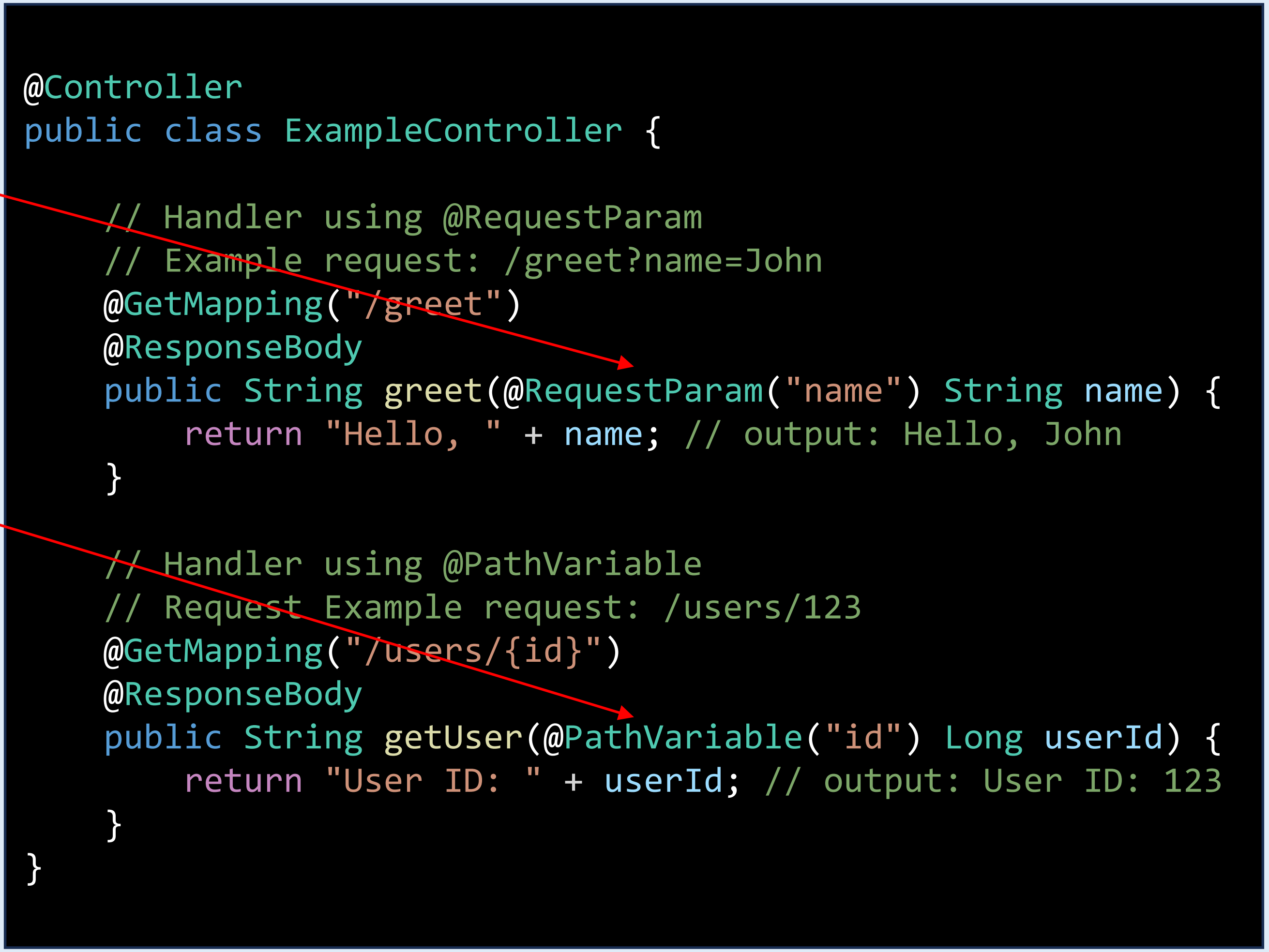
Q. How does Spring MVC handle form submissions?

Back to main index

# Q. What is the difference between @RequestParam and @PathVariable? V. IMP.

- @RequestParam extracts **query parameters**, form data, or any parameters from the URL.

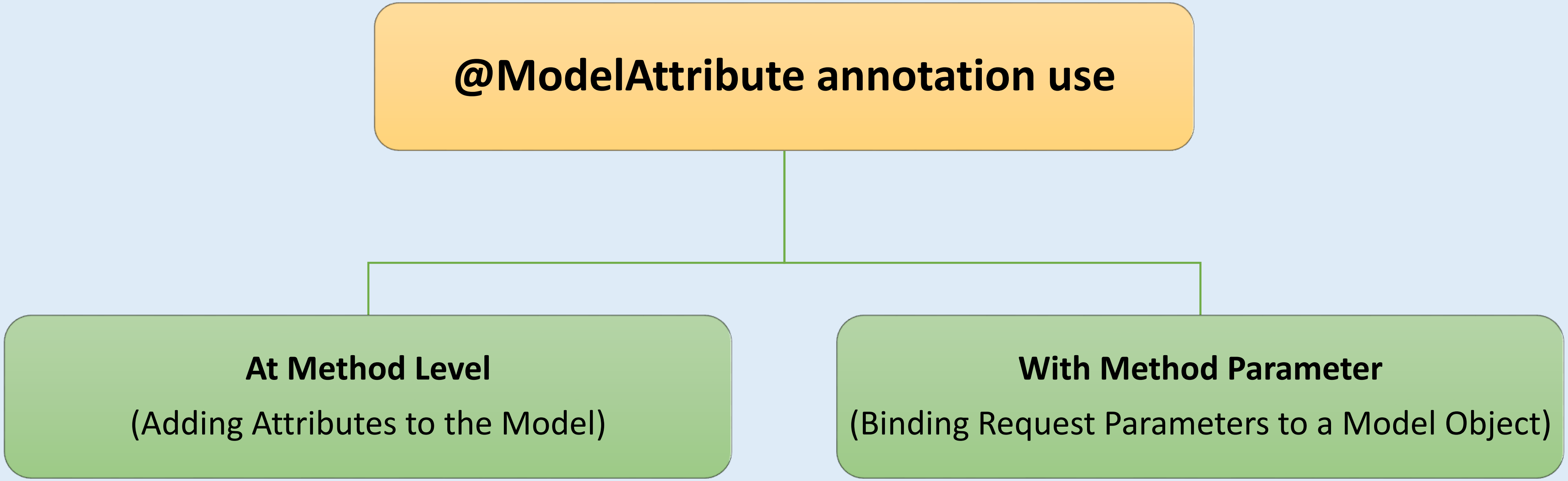
- @PathVariable extracts values from the URI path itself.



```
@Controller
public class ExampleController {

    // Handler using @RequestParam
    // Example request: /greet?name=John
    @GetMapping("/greet")
    @ResponseBody
    public String greet(@RequestParam("name") String name) {
        return "Hello, " + name; // output: Hello, John
    }

    // Handler using @PathVariable
    // Request Example request: /users/123
    @GetMapping("/users/{id}")
    @ResponseBody
    public String getUser(@PathVariable("id") Long userId) {
        return "User ID: " + userId; // output: User ID: 123
    }
}
```

back to chapter index

# Q. What is @ModelAttribute annotation in Spring MVC? V. IMP.



@ModelAttribute annotation use

**At Method Level**

(Adding Attributes to the Model)

**With Method Parameter**

(Binding Request Parameters to a Model Object)

back to chapter index

# Q. What is @ModelAttribute annotation at method level? When to use it? V. IMP.

- **Definition:** The @ModelAttribute annotation at the method level ensures that the method runs before any request-handling method in the controller.

- **When to use?** For common reusable code: It is used to automatically add common data to the model, making it available to all methods in that controller for every request.

```
@Controller
public class UserController {

    // Method-level @ModelAttribute to add
    // common data to the model
    @ModelAttribute
    public void addCommonAttributes(Model model) {
        model.addAttribute("appName", "My App");
    }

    @GetMapping("/user")
    public String showUser(Model model) {
        model.addAttribute("name", "Happy");
        return "userView";
    }

    @GetMapping("/profile")
    public String showProfile(Model model) {
        model.addAttribute("username", "Happy");
        return "profileView";
    }
}
```

back to chapter index

# Q. What is the role of @ModelAttribute annotation with parameters?

- **Definition:** The @ModelAttribute annotation with a parameter binds request data to a method parameter object.
- **When to use?** Use @ModelAttribute with parameters to customize names of the model attributes.

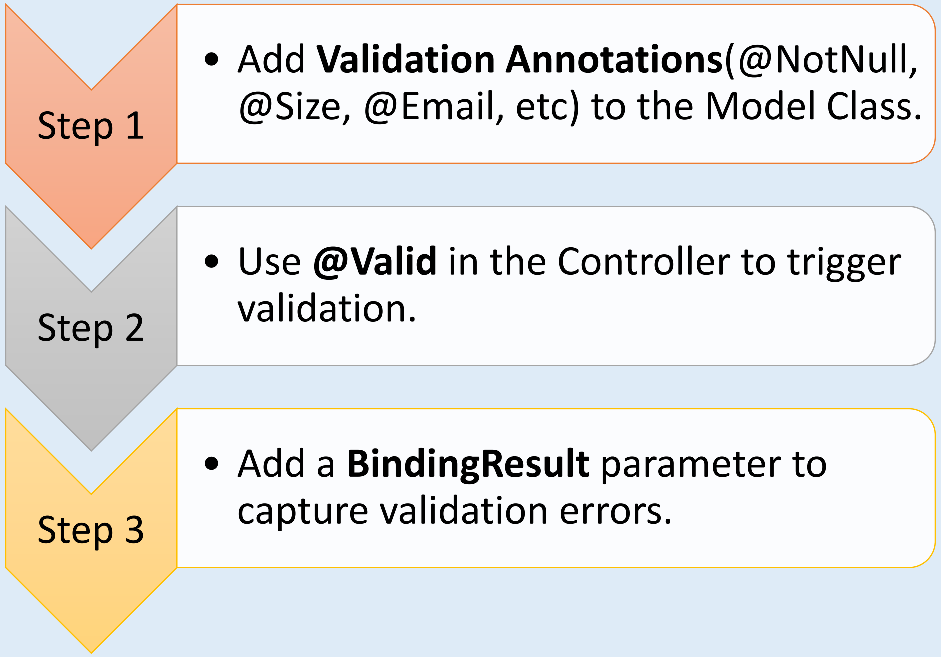
```
@Controller // with @ModelAttribute
public class UserController {

  @PostMapping("/register")
  public String submitForm(@ModelAttribute("customer") User user) {

    // 'user' object is stored in the model as "customer"
    return "registrationSuccess";
  }
}
```

```
@Controller // without @ModelAttribute
public class UserController {

 @PostMapping("/register")
  public String submitForm(User user) {

    // 'user' object is stored in the model as "user“
    return "registrationSuccess";
  }
}
```

back to chapter index

# Q. How to validate the data in Spring MVC? V. IMP.

Steps to Validate Data in Spring MVC:



Step 1

- Add **Validation Annotations**(@NotNull, @Size, @Email, etc) to the Model Class.

Step 2

- Use **@Valid** in the Controller to trigger validation.

Step 3

- Add a **BindingResult** parameter to capture validation errors.

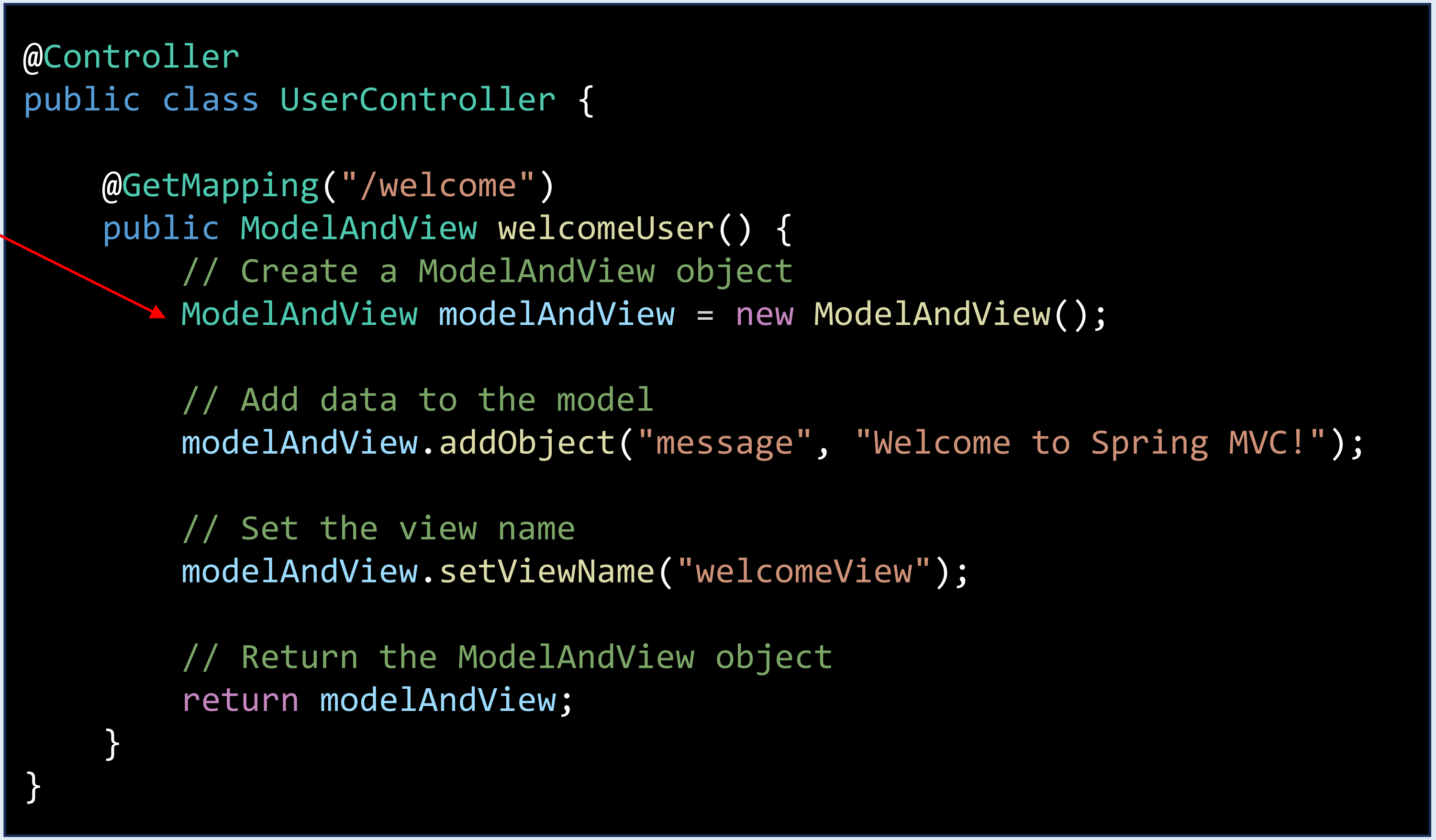
```
// User Model
public class User {
  @NotEmpty(message = "Username is required")
  private String username;

  @NotEmpty(message = "Email is required")
  @Email(message = "Email should be valid")
  private String email;

  // Getters and setters
}
```

```
@Controller
public class UserController {

    @PostMapping("/register")
    public String submitForm(@Valid @ModelAttribute("user") User
user, BindingResult result) {
        if (result.hasErrors()) {
       // Return to the form view if there are validation errors
            return "registrationForm";
        }
        // Process the valid form data
        return "registrationSuccess";
    }
}
```

back to chapter index

# Q. What is the purpose of ModelAndView in Spring MVC?

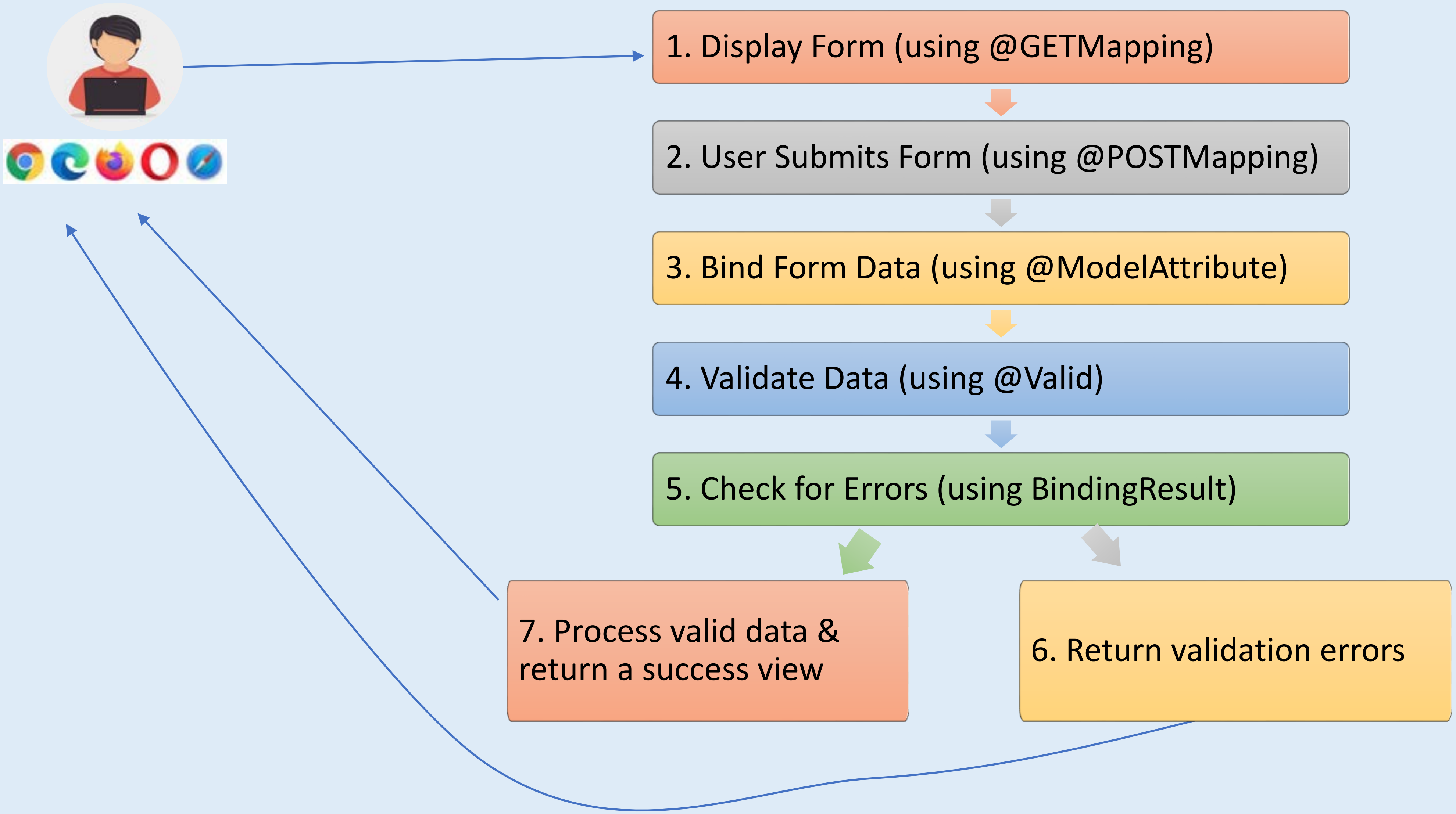
- ModelAndView class combines model data and view name in a single object.
- Then controllers return both together for view rendering.



```java
@Controller
public class UserController {

    @GetMapping("/welcome")
    public ModelAndView welcomeUser() {
        // Create a ModelAndView object
        ModelAndView modelAndView = new ModelAndView();

        // Add data to the model
        modelAndView.addObject("message", "Welcome to Spring MVC!");

        // Set the view name
        modelAndView.setViewName("welcomeView");

        // Return the ModelAndView object
        return modelAndView;
    }
}
```

back to chapter index

# Q. How does Spring MVC handle form submissions?



1. Display Form (using @GETMapping)

2. User Submits Form (using @POSTMapping)

3. Bind Form Data (using @ModelAttribute)

4. Validate Data (using @Valid)

5. Check for Errors (using BindingResult)

7. Process valid data & return a success view

6. Return validation errors

back to chapter index

# Q. How does Spring MVC handle form submissions?

1. Display Form: Controller serves the form with an empty model object.

2. User Submits Form: Form data is sent to the server.

3. Bind Form Data: Spring binds form data to a model object using @ModelAttribute.

4. Validate Data: Spring validates the model object using @Valid and captures errors in BindingResult.

5. Check for Errors: Controller checks BindingResult for validation errors.

6. Process Valid Data: If no errors, the controller processes the valid data.

7. Return View: Return a success view or redisplay form with errors.

8. Show Validation Errors: Errors are displayed next to form fields in the view.

back to chapter index

# Java Chapters

## REST WEBSERVICES/ REST API

Back to main index

# RESTful Services/ REST API - Basics

Q. What is REST & RESTful API? **V. IMP.**

Q. What are HTTP Request and Response structures in UI and REST API?

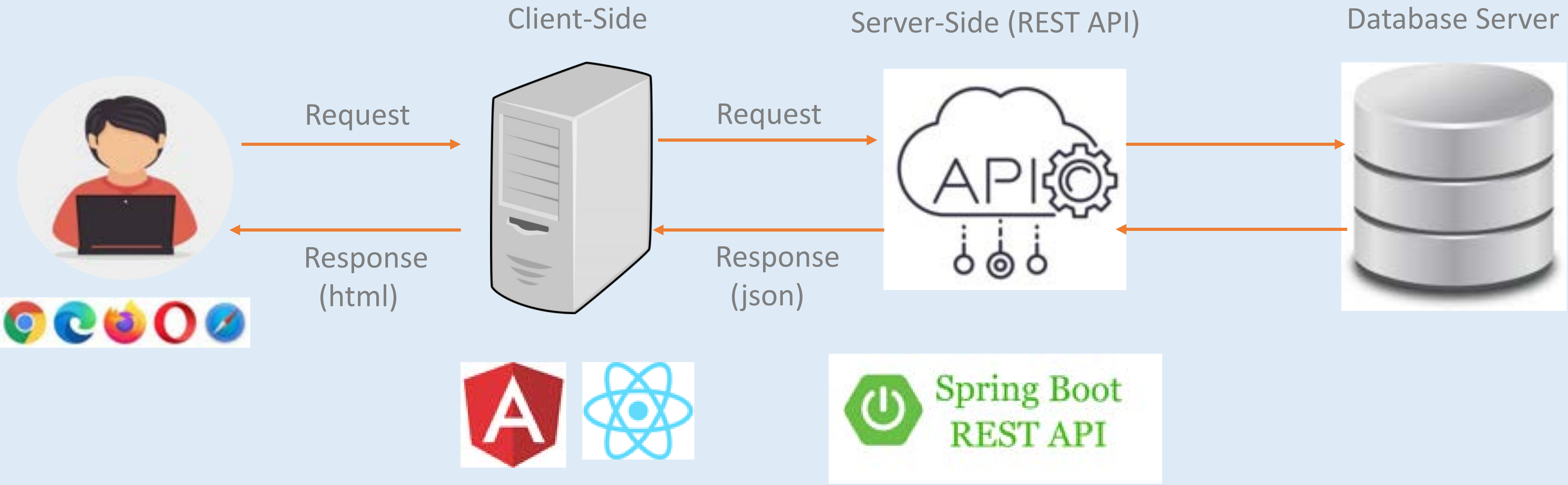
Q. What are Top 5 REST guidelines and the advantages of them? **V. IMP.**

Q. What is the difference between REST API and SOAP API?

Back to main index

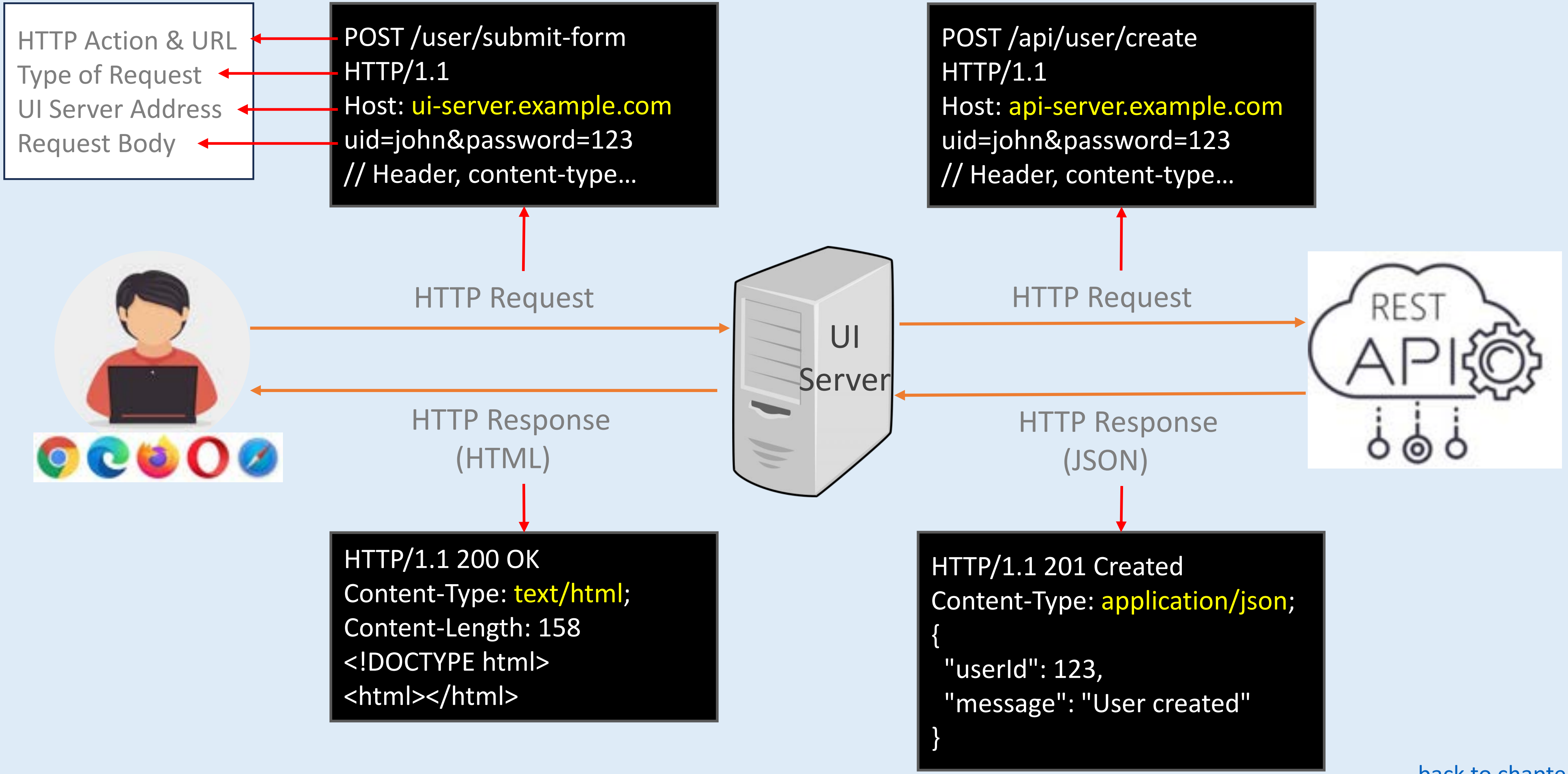
# Q. What is REST & RESTful API? V. IMP.

- REST (Representational State Transfer) is an **architectural style** for designing networked applications (REST is a set of **guidelines** for creating API’s).
- RESTful API is a service which follow REST principles/ guidelines.



back to chapter index

# Q. What are HTTP Request and Response structures in UI and REST API?



```
POST /user/submit-form
HTTP/1.1
Host: ui-server.example.com
uid=john&password=123
// Header, content-type...
```

```
HTTP/1.1 200 OK
Content-Type: text/html;
Content-Length: 158
<!DOCTYPE html>
<html></html>
```

```
POST /api/user/create
HTTP/1.1
Host: api-server.example.com
uid=john&password=123
// Header, content-type...
```

```
HTTP/1.1 201 Created
Content-Type: application/json;
{
  "userId": 123,
  "message": "User created"
}
```

back to chapter index

# Q. What are HTTP Request and Response structures in UI and REST API?

## HTTP Request

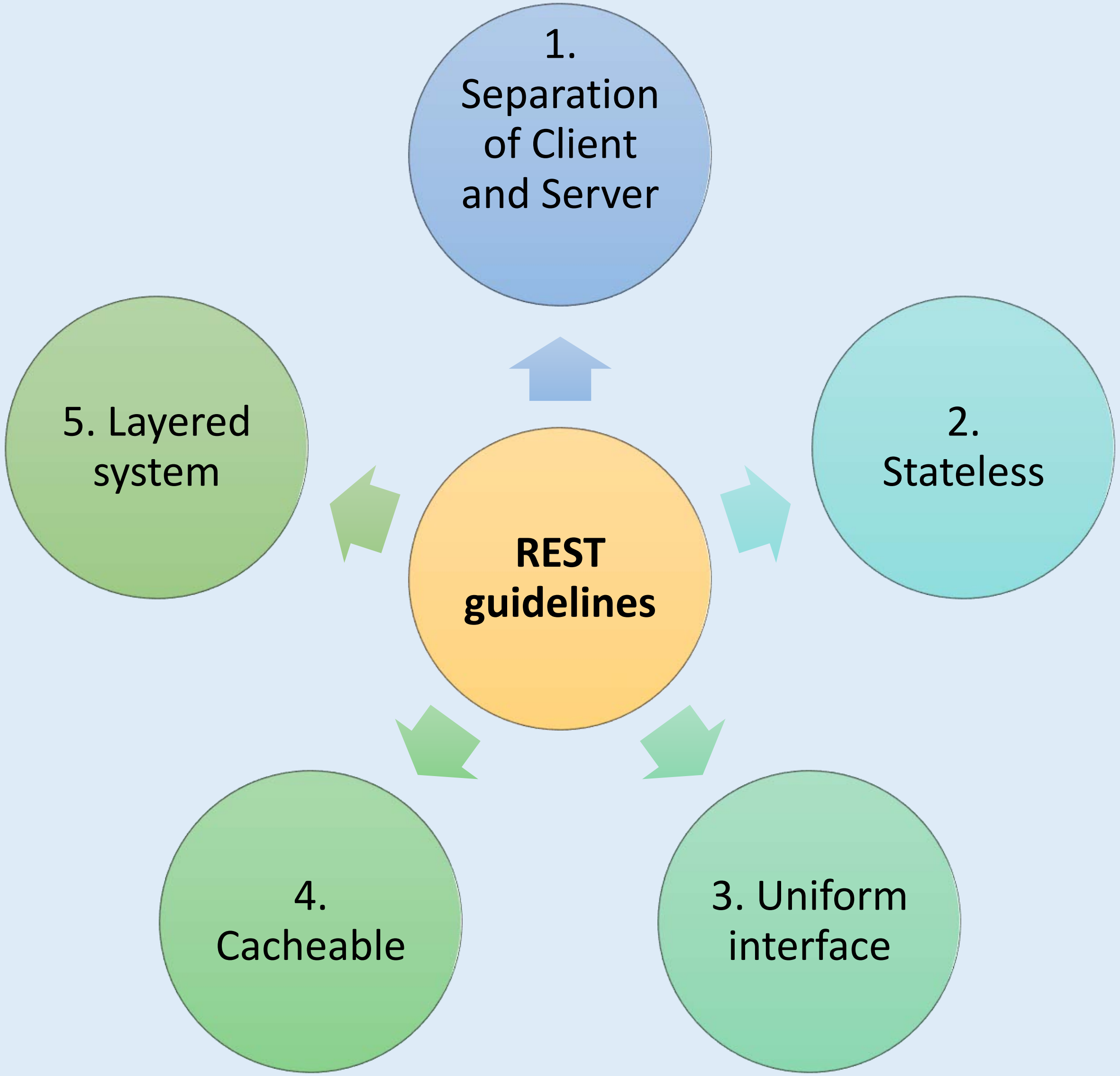
- An HTTP (Hypertext Transfer Protocol) request is a message sent by a client (such as a web browser or a mobile app) to a server, requesting a particular action or resource.
- It contains HTTP Action(GET, POST…), URL, Request Body, Request Header.

## HTTP Response

- An HTTP response is a message sent by a server back to the client in response to an HTTP request.
- It includes status code, content type, content.

back to chapter index

# Q. What are Top 5 REST guidelines and the advantages of them? V. IMP.



back to chapter index

# Q. What are Top 5 REST guidelines and the advantages of them? V. IMP.

**1. Separation of Client & Server** The implementation of the client and the server must be done independently.

- Advantage: Independence allows **easier maintenance, scalability, and evolution**.

**2. Stateless** The server will not store anything about the latest HTTP request the client made.

- Advantage: It will treat every request as new request. It **simplifies** server implementation as it is not overloading it with state management.

**3. Uniform interface** Identify the resources by URL (e.g., www.abc.com/api/questions).

- Advantage: standardized URLs, making it **easy to understand** and use the API.

**4. Cacheable** The API response should be cacheable to improve the performance.

- Advantage: Caching API responses **improves performance** by reducing the need for repeated requests to the server.

**5. Layered system** The system should follow layered pattern.

- Advantage: A layered system, such as the Model-View-Controller (MVC) pattern, promotes **modular design** and separation of concerns.

back to chapter index

# Q. What is the difference between REST API and SOAP API?

| Feature | REST API | SOAP API |
|---|---|---|
| **Architecture** | REST is an architectural style. | SOAP(Simple Object Access Protocol) is a protocol. |
| **Protocol** | Uses HTTP or HTTPS. | Can use various protocols (HTTP, SMTP, etc.). |
| **Message Format** | Uses lightweight formats like JSON, XML. | Typically uses XML. |
| **State** | Stateless. | Can be stateful or stateless. |
| **Error Handling** | Relies on HTTP status codes. | Defines its own fault mechanism. |
| **Performance** | Generally lightweight and faster. | Can be slower due to XML processing. |

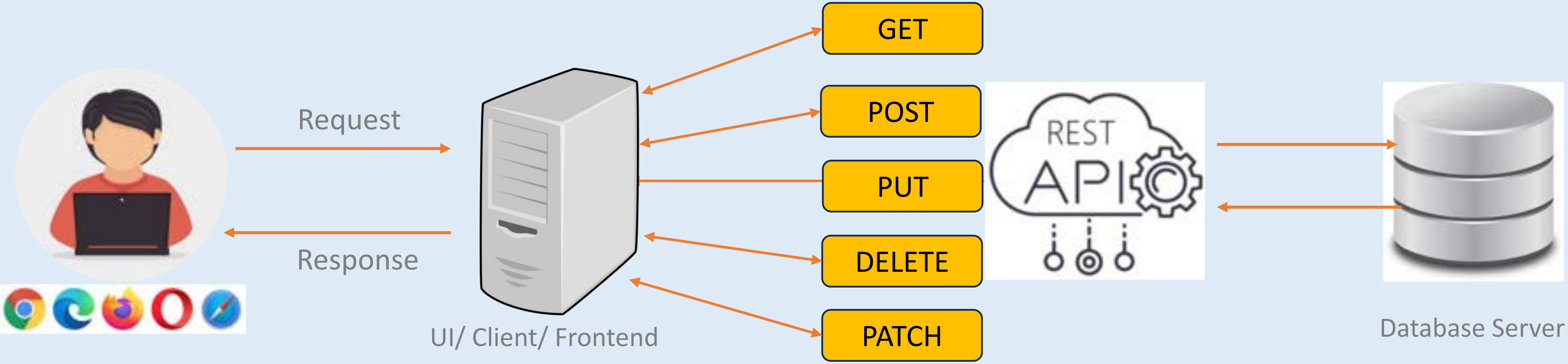
back to chapter index

# REST - HTTP Methods & Status Codes

Back to main index

# Q. What are HTTP Verbs and HTTP methods? V. IMP.

- HTTP methods, also known as HTTP verbs, are a set of actions that a client can take on a resource.



back to chapter index

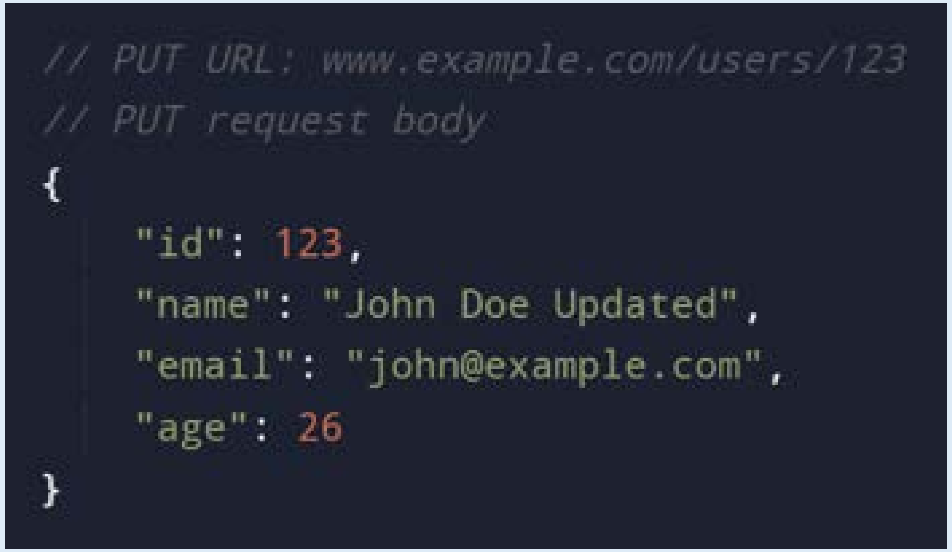
# Q. What are GET, POST, PUT & DELETE HTTP methods?

| HTTP Method | Action | Example |
|---|---|---|
| GET | Retrieve data from a specified resource | www.example.com/users<br>(retrieve users list)<br>www.example.com/users/123<br>(retrieve single user of id - 123) |
| POST | Submit data to be processed | www.example.com/users<br>(submit and create a new user from data provided in request) |
| PUT | Update a resource or create a new resource if it does not exist | www.example.com/users/123<br>(update user 123 details from data provided in request) |
| DELETE | Request removal of a resource | www.example.com/users/123<br>(delete user 123) |

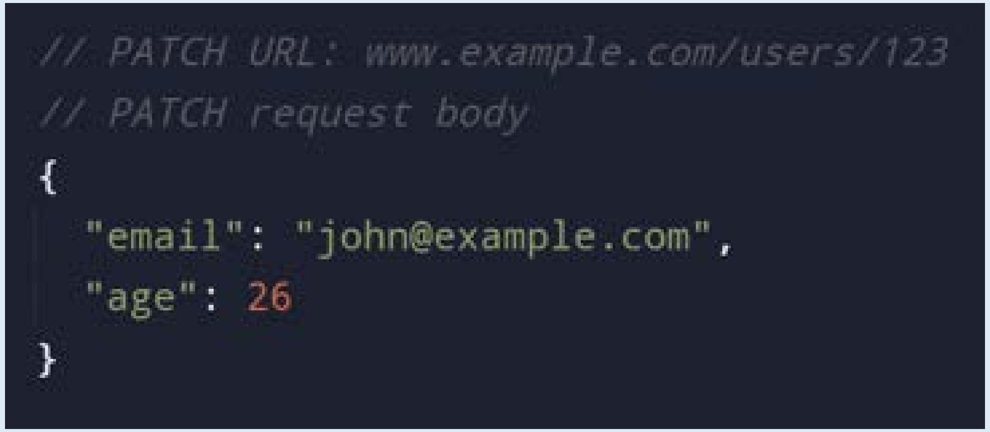
back to chapter index

# Q. What is the difference between PUT & PATCH methods?

| PUT | PATCH |
|---|---|
| Both PUT and PATCH method are used to **update a resource** by replacing the resource with the new data provided in the request. | |
| **Full Resource Replacement:** In a PUT request, the client sends the full updated resource in the request body, replacing the existing resource on the server. | **Partial Updates:** In a PATCH request, the client sends specific changes or instructions for modifying the resource, updating only certain fields without resending the entire resource. |



```
// PUT URL: www.example.com/users/123
// PUT request body
{
    "id": 123,
    "name": "John Doe Updated",
    "email": "john@example.com",
    "age": 26
}
```

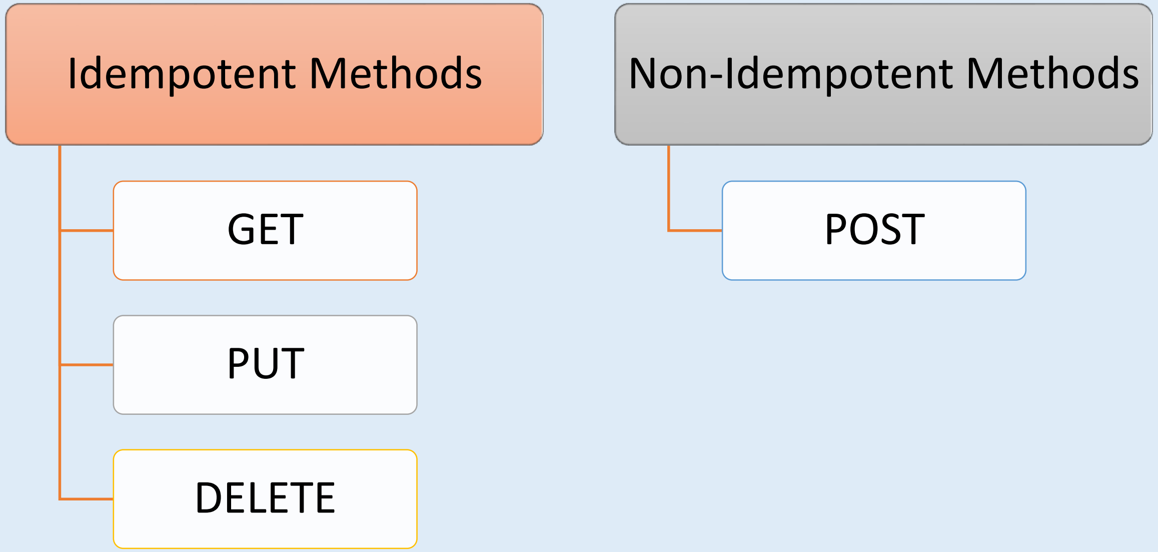


```
// PATCH URL: www.example.com/users/123
// PATCH request body
{
  "email": "john@example.com",
  "age": 26
}
```

back to chapter index

# Q. Explain the concept of Idempotence in RESTful APIs.

- Idempotence meaning performing an operation multiple times should have the same outcome as performing it once. For example, Sending multiple identical GET requests will always return the same response



Idempotent Methods

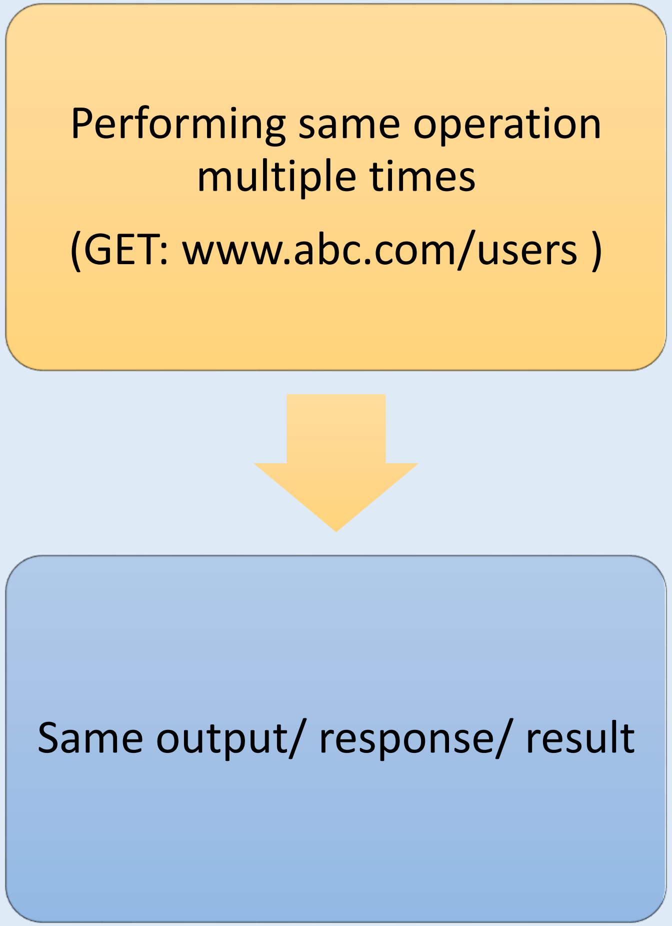
GET

PUT

DELETE

Non-Idempotent Methods

POST



Performing same operation multiple times

(GET: www.abc.com/users )

Same output/ response/ result

back to chapter index

# Q. What are the role of status codes in RESTful APIs?

- Status codes are used to convey the results of a client’s request.

### 1XX (Info)

- 100 Continue

### 2XX (Success)

- 200: OK
- 201: Created
- 202: Accepted
- 204: No Content

### 3XX (Redirection)

- 300: Multiple Choices

### 4XX (Client Error)

- 400: Bad Request
- 401 Unauthorized
- 403 Forbidden
- 404 Not Found

### 5XX (Server Error)

- 500: Internal Server Error
- 501: Not Implemented
- 502: Bad Gateway
- 503: Service Unavailable

back to chapter index

# REST - CORS, Serialization, Deserialization, Others

Q. What is CORS in RESTful APIs? **V. IMP.**

Q. What are Serialization & Deserialization? **V. IMP.**

Q. What are the types of serialization?

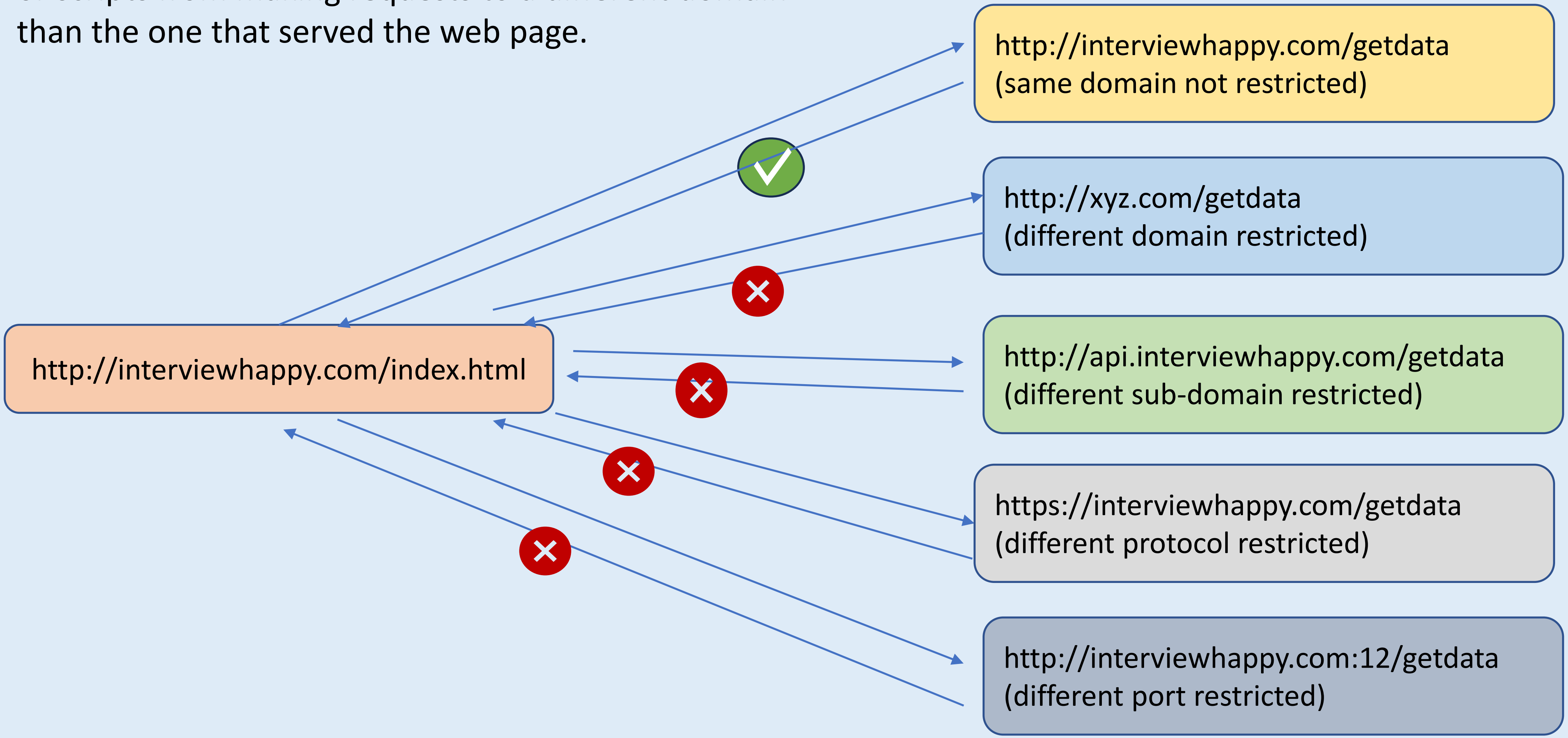
Q. Explain the concept of versioning in RESTful APIs.

Q. What is an API document? What are the popular documentation formats?

Back to main index

# Q. What is CORS in RESTful APIs? V. IMP.

- CORS(Cross-Origin Resource Sharing) is a security feature implemented in web browsers that restricts web pages or scripts from making requests to a different domain than the one that served the web page.



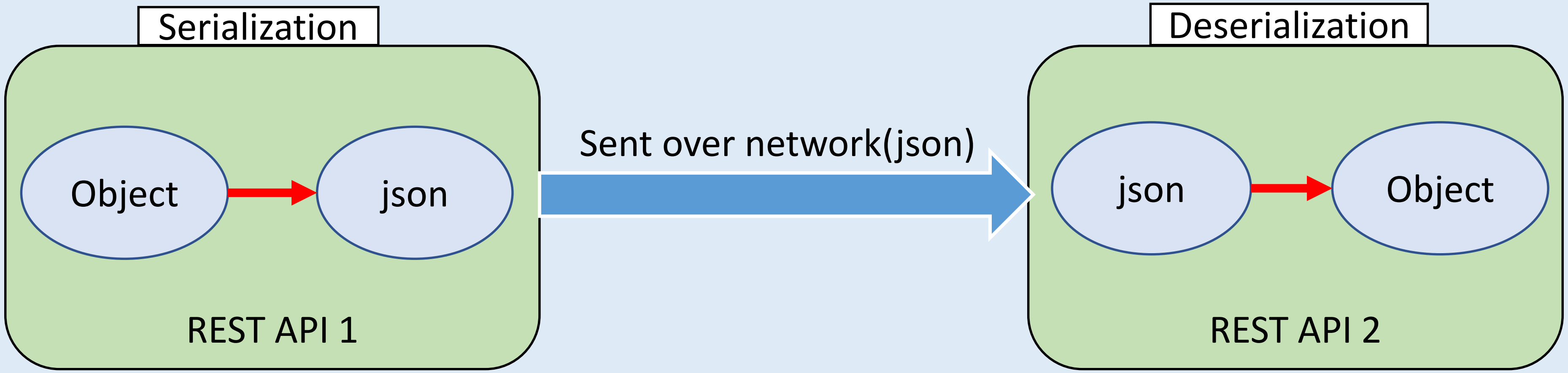
http://interviewhappy.com/index.html
http://interviewhappy.com/getdata
(same domain not restricted)
✓
http://xyz.com/getdata
(different domain restricted)
✗
http://api.interviewhappy.com/getdata
(different sub-domain restricted)
✗
https://interviewhappy.com/getdata
(different protocol restricted)
✗
http://interviewhappy.com:12/getdata
(different port restricted)
✗

back to chapter index

# Q. What are Serialization & Deserialization? V. IMP.

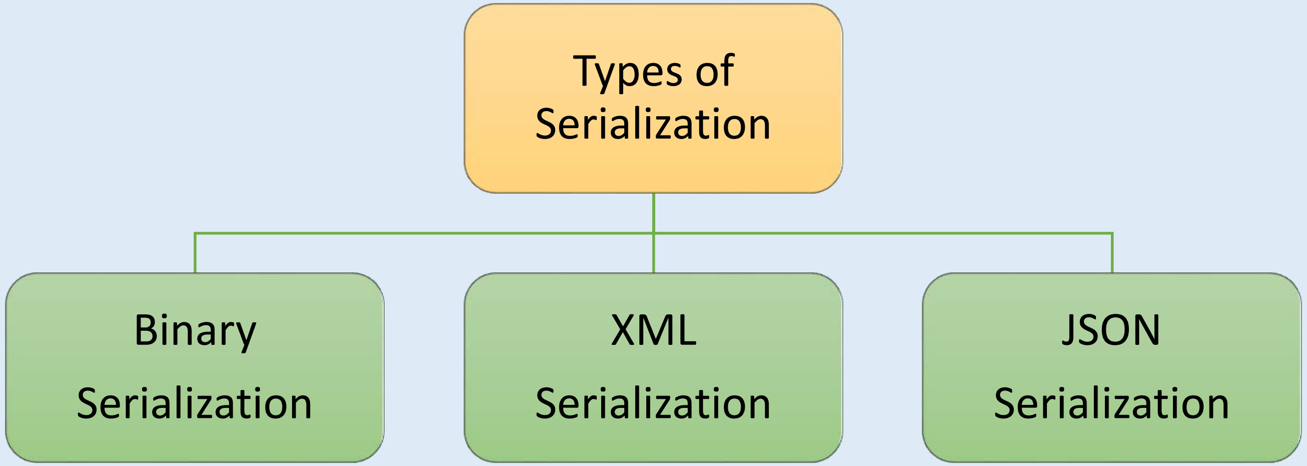
- Serialization is the process of **converting an object** into a format that can be stored, transmitted, or reconstructed later.

- Deserialization is the process of converting serialized data, such as binary/ XML/ json data, back into an object.



back to chapter index

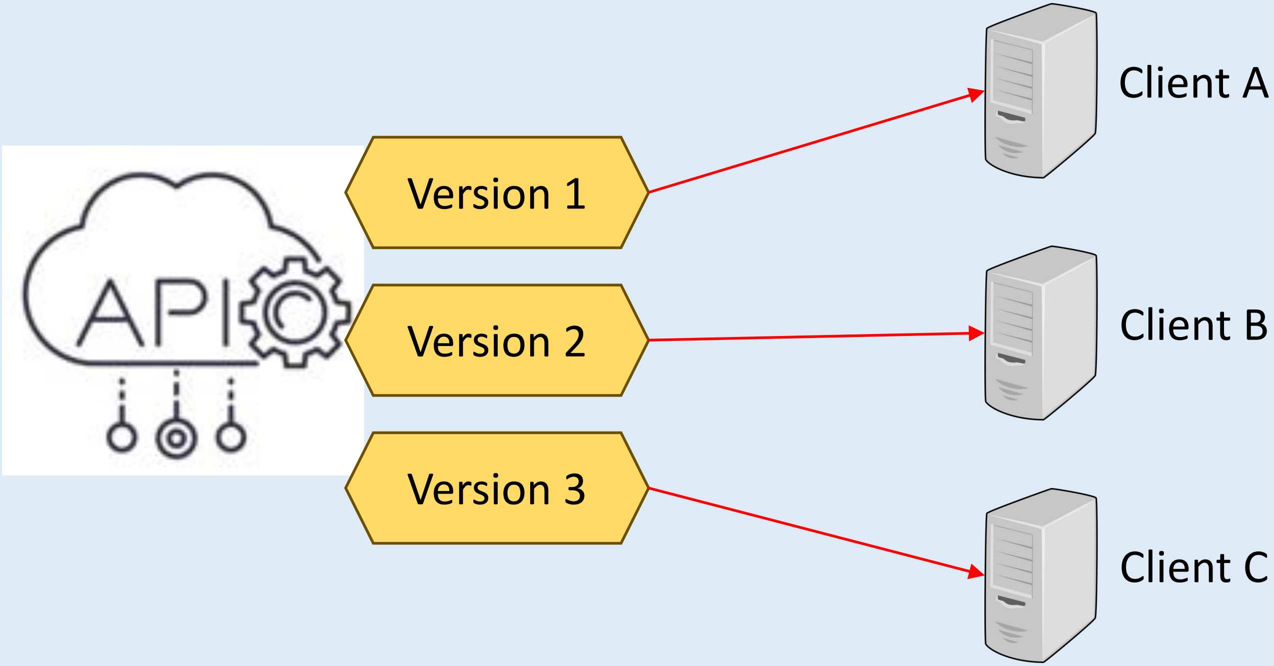
# Q. What are the types of serialization?



back to chapter index

# Q. Explain the concept of versioning in RESTful APIs.

- Versioning in RESTful APIs refers to the practice of **maintaining multiple versions** of an API to support backward compatibility.



https://api.example.com/v1/resource

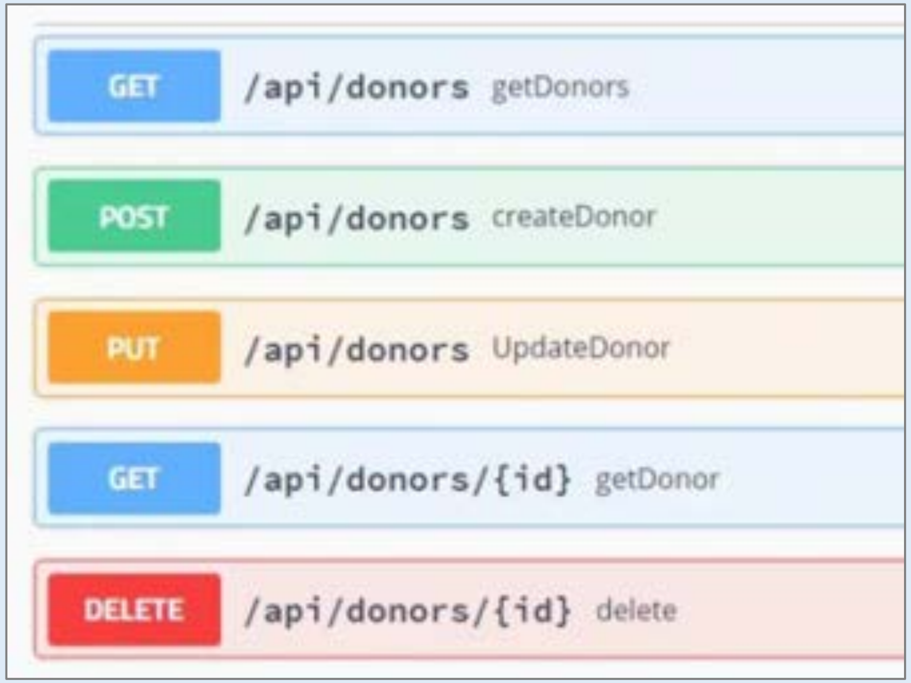
https://api.example.com/v2/resource

https://api.example.com/v3/resource

back to chapter index

# Q. What is an API document? What are the popular documentation formats?

- An API document, describe the functionality, features, and usage of a REST API.



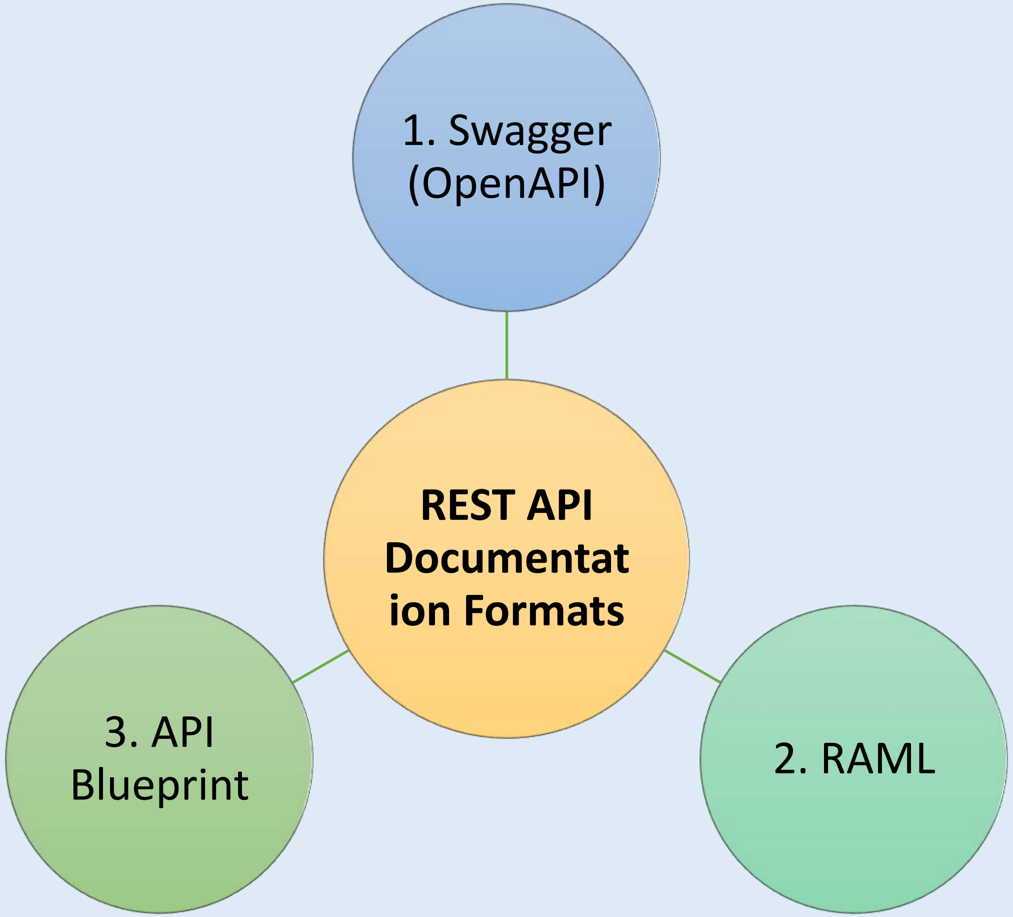
GET /api/donors getDonors

POST /api/donors createDonor

PUT /api/donors UpdateDonor

GET /api/donors/{id} getDonor

DELETE /api/donors/{id} delete



**REST API Documentation Formats**

1. Swagger (OpenAPI)

2. RAML

3. API Blueprint

back to chapter index

# REST - Authentication & Authorization

Q. What are Authentication and Authorization? **V. IMP.**

Q. What is Token based and JWT authentication? **V. IMP.**

Q. What are the parts of JWT token?

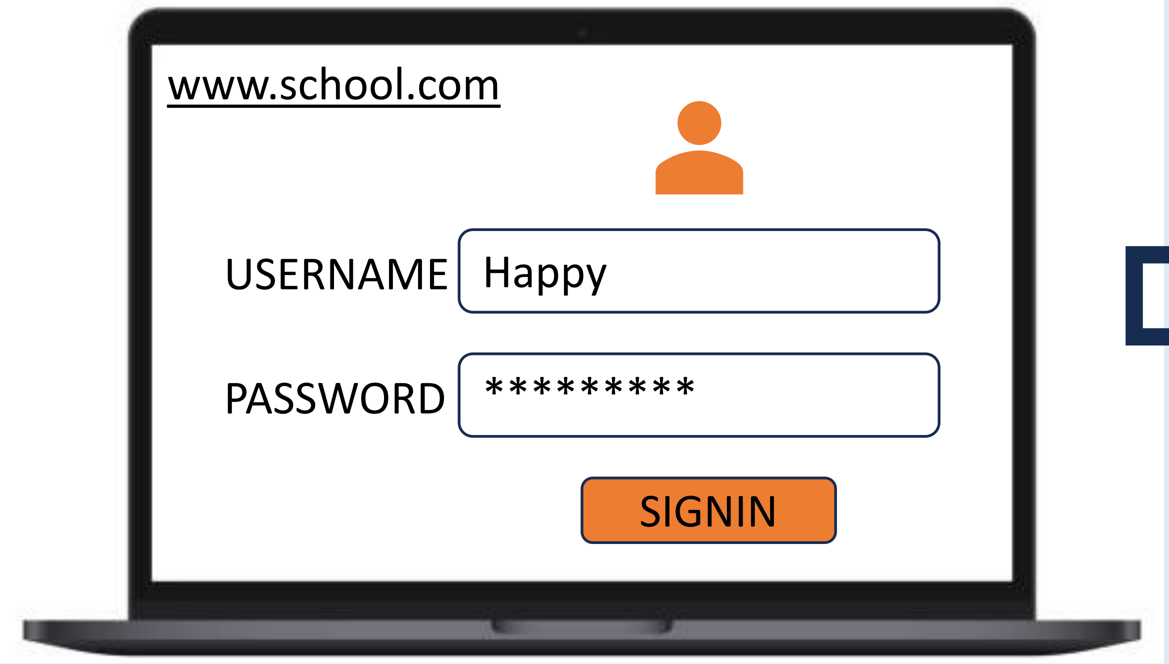
Q. Where JWT token reside in the request?

Back to main index

# Q. What are Authentication and Authorization? V. IMP.

- Authentication is the process of **verifying the identity** of a user by validating their credentials such as username and password.

- Authorization is the process of allowing an authenticated user **access to resources**. Authentication is always precedes to Authorization.

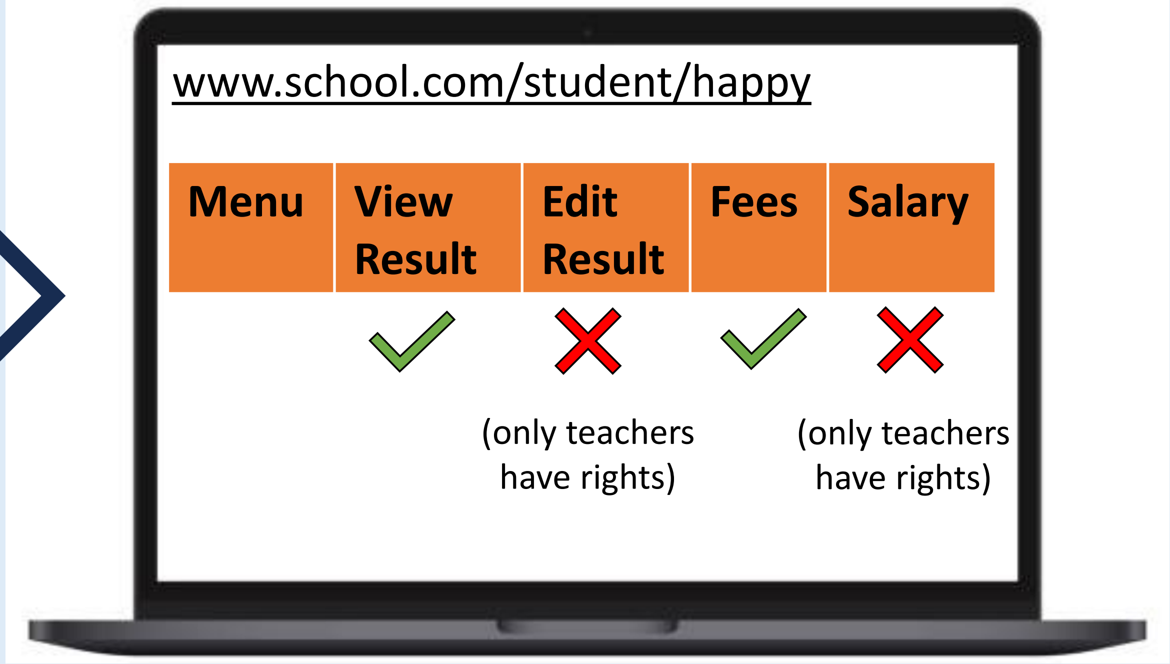


www.school.com

USERNAME Happy

PASSWORD *********

**Authentication(Who are you?)**



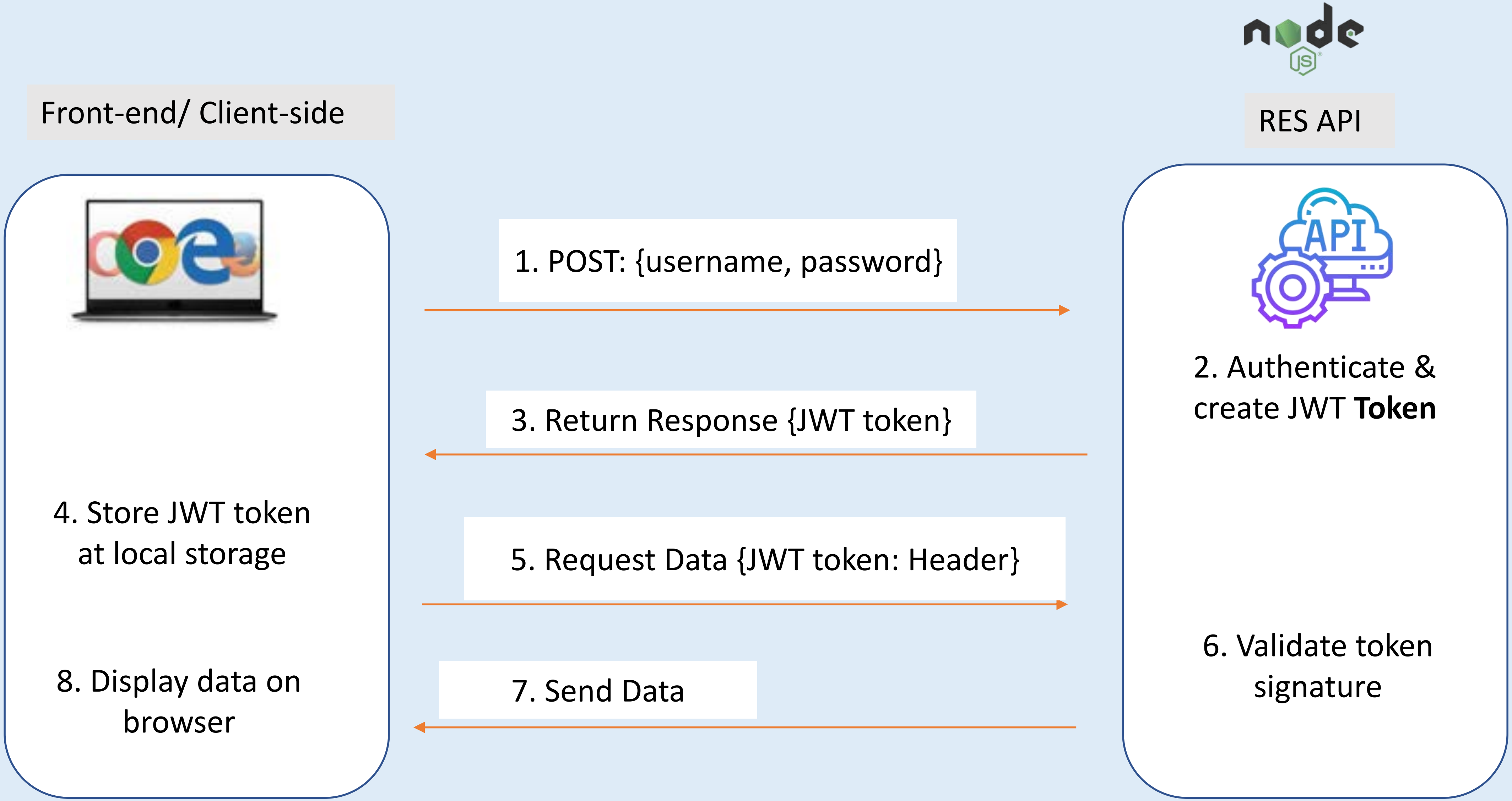
www.school.com/student/happy

| Menu | View Result | Edit Result | Fees | Salary |
|---|---|---|---|---|
|  | ✓ | ✗ | ✓ | ✗ |
|  |  | (only teachers have rights) |  | (only teachers have rights) |

**Authorization(Your rights/ access?)**

back to chapter index

# Q. What is Token based and JWT authentication? V. IMP.



Front-end/ Client-side

RES API

1. POST: {username, password}

2. Authenticate & create JWT **Token**

3. Return Response {JWT token}

4. Store JWT token at local storage

5. Request Data {JWT token: Header}

6. Validate token signature

7. Send Data

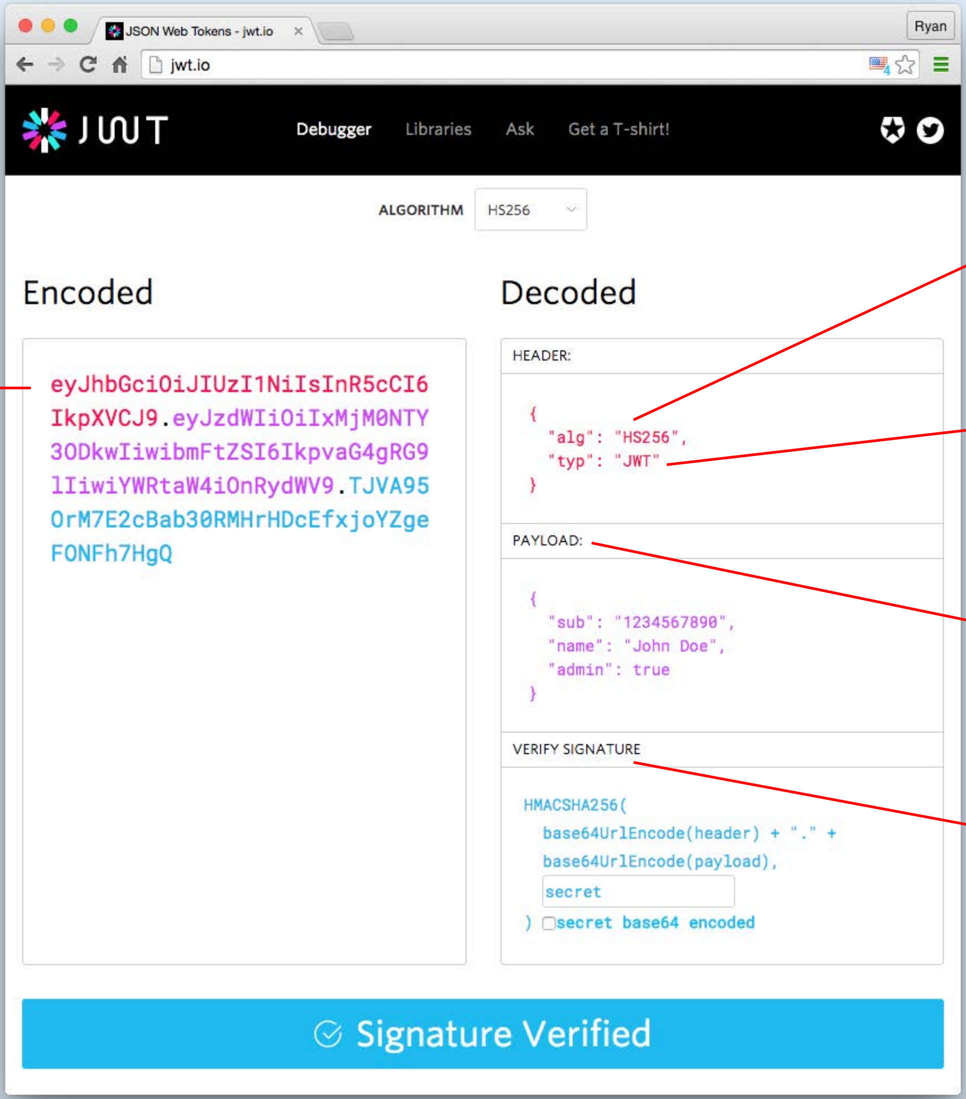
8. Display data on browser

back to chapter index

# Q. What are the parts of JWT token?

- JWT token has 3 parts:

1. Header
2. Payload
3. Signature



ALGORITHM HS256

Encoded

```
eyJhbGciOiJIUzI1NiIsInR5cCI6
IkpXVCJ9.eyJzdWIiOiIxMjM0NTY
3ODkwIiwibmFtZSI6IkpvaG4gRG9
lIiwiYWRtaW4iOnRydWV9.TJVA95
OrM7E2cBab30RMHrHDcEfxjoYZge
FONFh7HgQ
```

Decoded

HEADER:

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

PAYLOAD:

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}
```

VERIFY SIGNATURE

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secret
) □secret base64 encoded
```

Signature Verified

JWT Token

Algorithm used to generate the token.

The type of the token, which is JWT here.

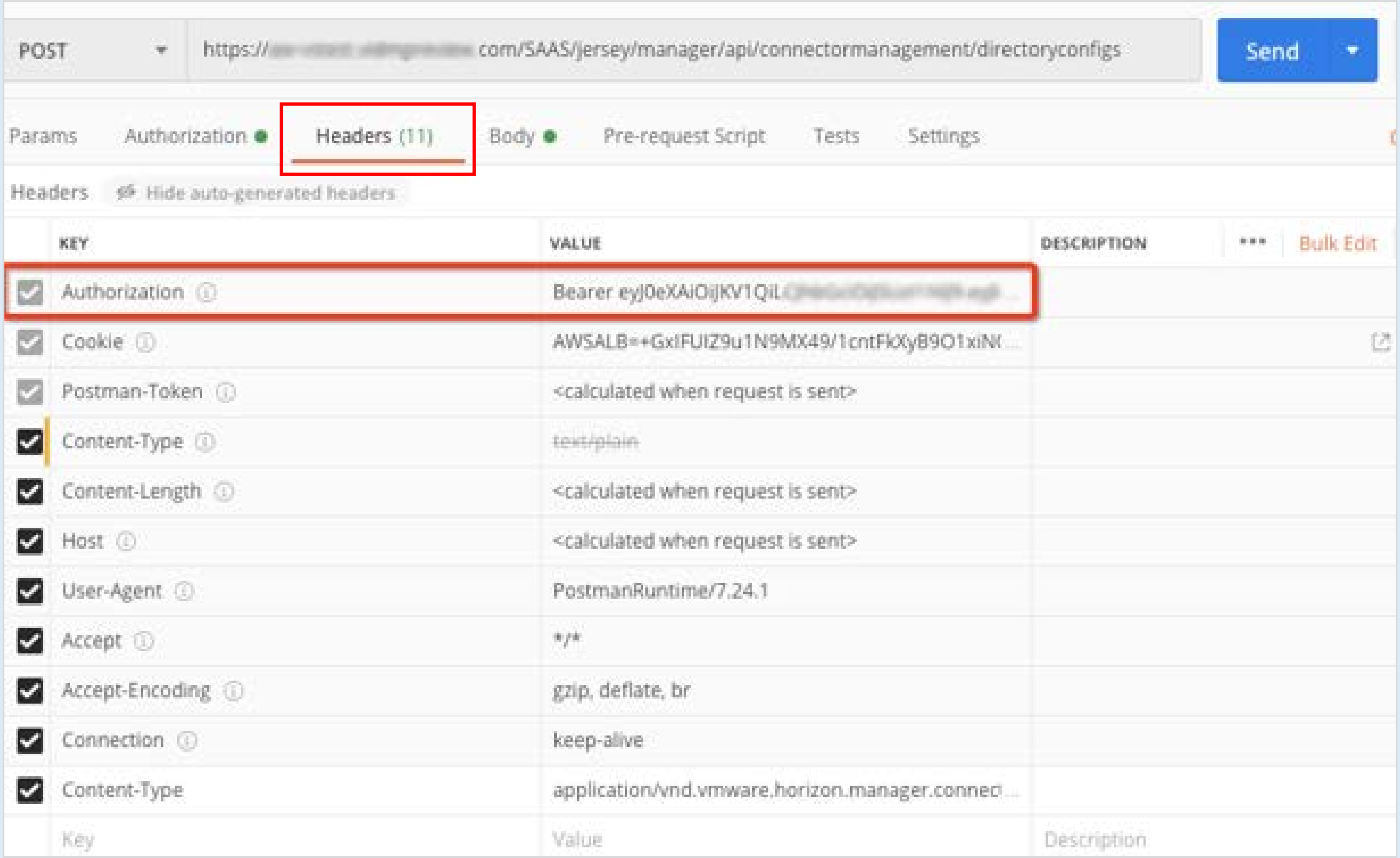
The payload contains the CLAIMS.

The signature is a string that is used to ensure the INTEGRITY of the token and verify that it has not been tampered with.

back to chapter index

# Q. Where JWT token reside in the request?

- In REQUEST HEADER.



https://[redacted].com/SAAS/jersey/manager/api/connectormanagement/directoryconfigs

| | KEY | VALUE | DESCRIPTION |
|---|---|---|---|
| ☑ | Authorization | Bearer eyJ0eXAiOiJKV1QiL[redacted] | |
| ☑ | Cookie | AWSALB=+GxIFUIZ9u1N9MX49/1cntFkXyB9O1xiN… | |
| ☑ | Postman-Token | <calculated when request is sent> | |
| ☑ | Content-Type | ~~text/plain~~ | |
| ☑ | Content-Length | <calculated when request is sent> | |
| ☑ | Host | <calculated when request is sent> | |
| ☑ | User-Agent | PostmanRuntime/7.24.1 | |
| ☑ | Accept | */* | |
| ☑ | Accept-Encoding | gzip, deflate, br | |
| ☑ | Connection | keep-alive | |
| ☑ | Content-Type | application/vnd.vmware.horizon.manager.connec… | |
| | Key | Value | Description |

back to chapter index

# Mock Interviews

1. Java-Basics, Variables & Data types, Operators, Control statements

2. OOPS - Classes, Objects, Access Specifiers, Getter-Setter & this keyword

3. OOPS - Inheritance, Polymorphism, Encapsulation & Abstraction

4. Abstract class & Interface, Constructors

5. Exception Handling

6. Collections

7. Multithreading

8. Generics

9. Types of Classes

Back to main index

# Mock Interview

## Java-Basics, Variables & Data types, Operators, Control statements

Q. What happens if a break statement is not used with case of a switch statement?

Q. What is the use of the default case in a switch statement?

Q. Can the body of an if statement be empty in Java?

Q. Can you nest if statements within switch statements and vice versa in Java?

Q. Can you have an else statement without an if statement in Java?

Q. Can a Java program run without JDK or JRE or JVM installed on the system?

Q. What data type would you use to store a single character?

Q. What is the default value of a local variable in Java?

Q. What is the default value of instance variables in Java?

Q. What keyword is used to declare a constant variable?

Back to main index

# Q. What happens if a break statement is not used with case of a switch statement?

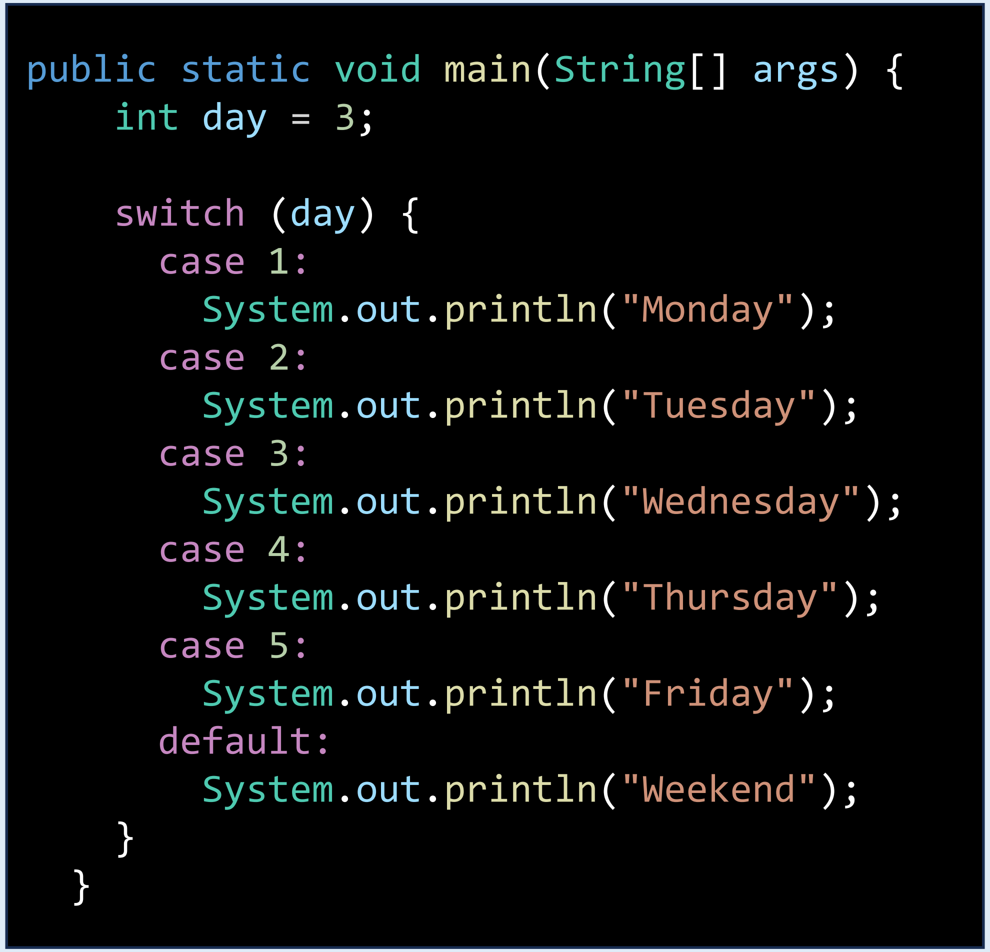
- If a break is missing in a switch case, the code runs into the **next cases until a break or the switch ends**. This is called "fall-through."

```java
public static void main(String[] args) {
    int day = 3;

    switch (day) {
      case 1:
        System.out.println("Monday");
      case 2:
        System.out.println("Tuesday");
      case 3:
        System.out.println("Wednesday");
      case 4:
        System.out.println("Thursday");
      case 5:
        System.out.println("Friday");
      default:
        System.out.println("Weekend");
    }
  }
// Output: Wednesday Thursday Friday Weekend
```

back to chapter index

# Q. What is the use of the default case in a switch statement?

- The default case runs when **none of the other cases match**, ensuring something happens for unmatched values.

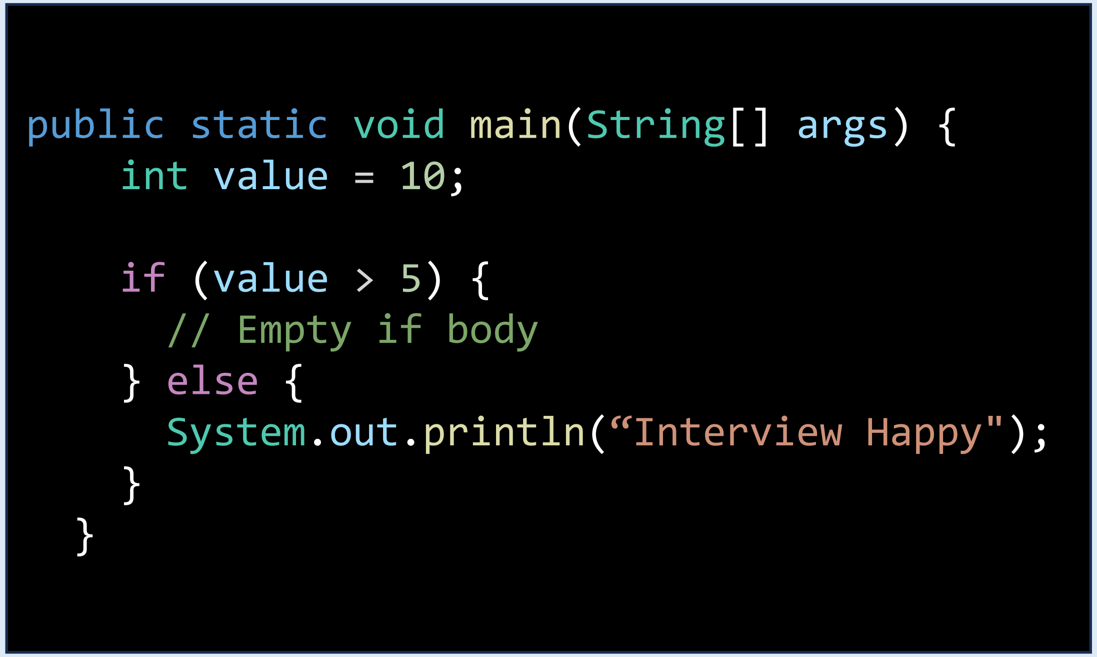


```java
public static void main(String[] args) {
    int day = 3;

    switch (day) {
      case 1:
        System.out.println("Monday");
      case 2:
        System.out.println("Tuesday");
      case 3:
        System.out.println("Wednesday");
      case 4:
        System.out.println("Thursday");
      case 5:
        System.out.println("Friday");
      default:
        System.out.println("Weekend");
    }
  }
```

back to chapter index

# Q. Can the body of an if statement be empty in Java?

- **Yes**, an if statement can have an empty body. It will simply do nothing if the condition is true.

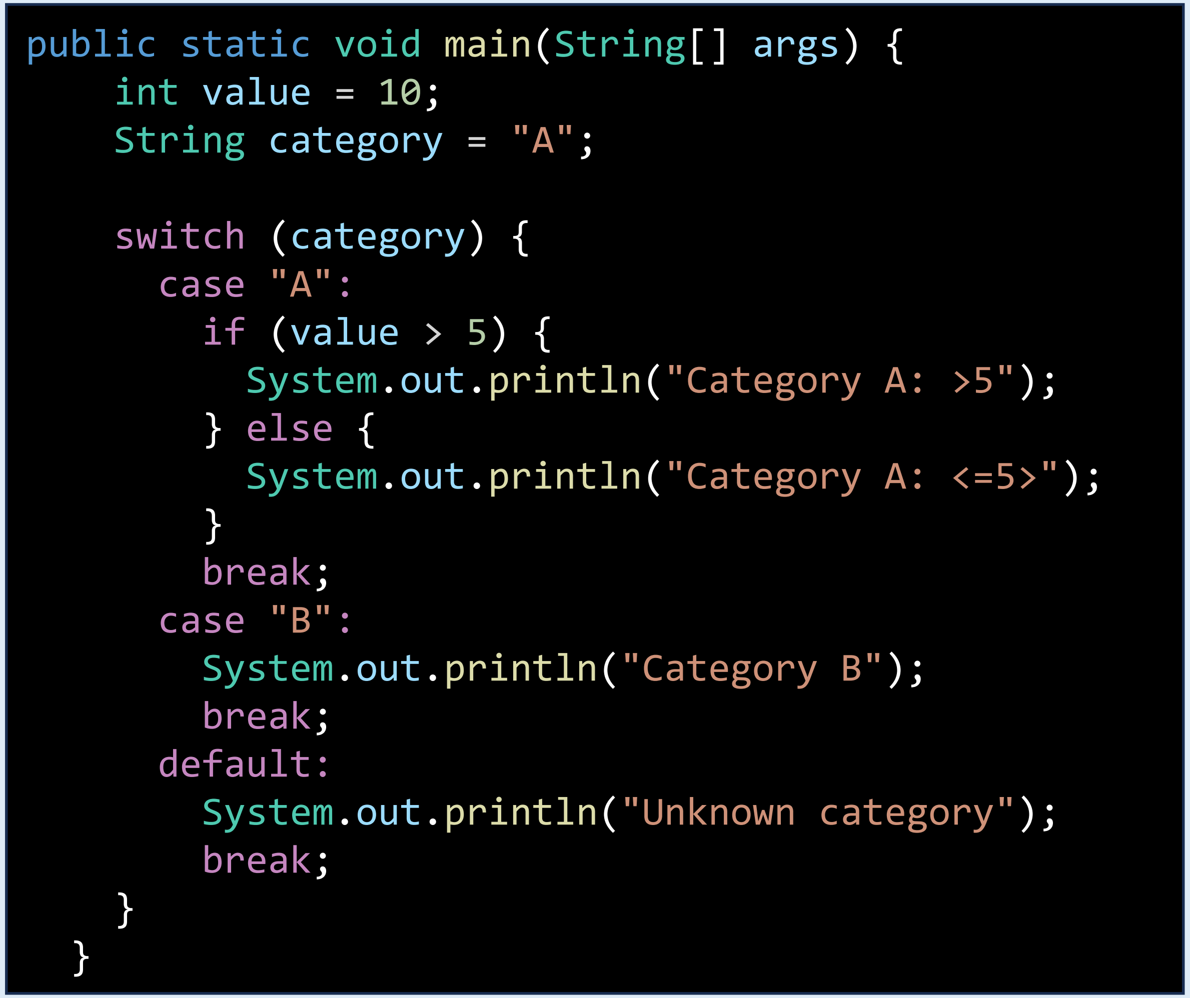


```
public static void main(String[] args) {
    int value = 10;

    if (value > 5) {
      // Empty if body
    } else {
      System.out.println(“Interview Happy");
    }
  }
```

back to chapter index

# Q. Can you nest if statements within switch statements and vice versa in Java?

- **Yes**, we can nest if statements within switch statements and vice versa.



```java
public static void main(String[] args) {
  int value = 10;
  String category = "A";

  switch (category) {
   case "A":
    if (value > 5) {
     System.out.println("Category A: >5");
    } else {
     System.out.println("Category A: <=5>");
    }
    break;
   case "B":
    System.out.println("Category B");
    break;
   default:
    System.out.println("Unknown category");
    break;
  }
 }
```

[back to chapter index]

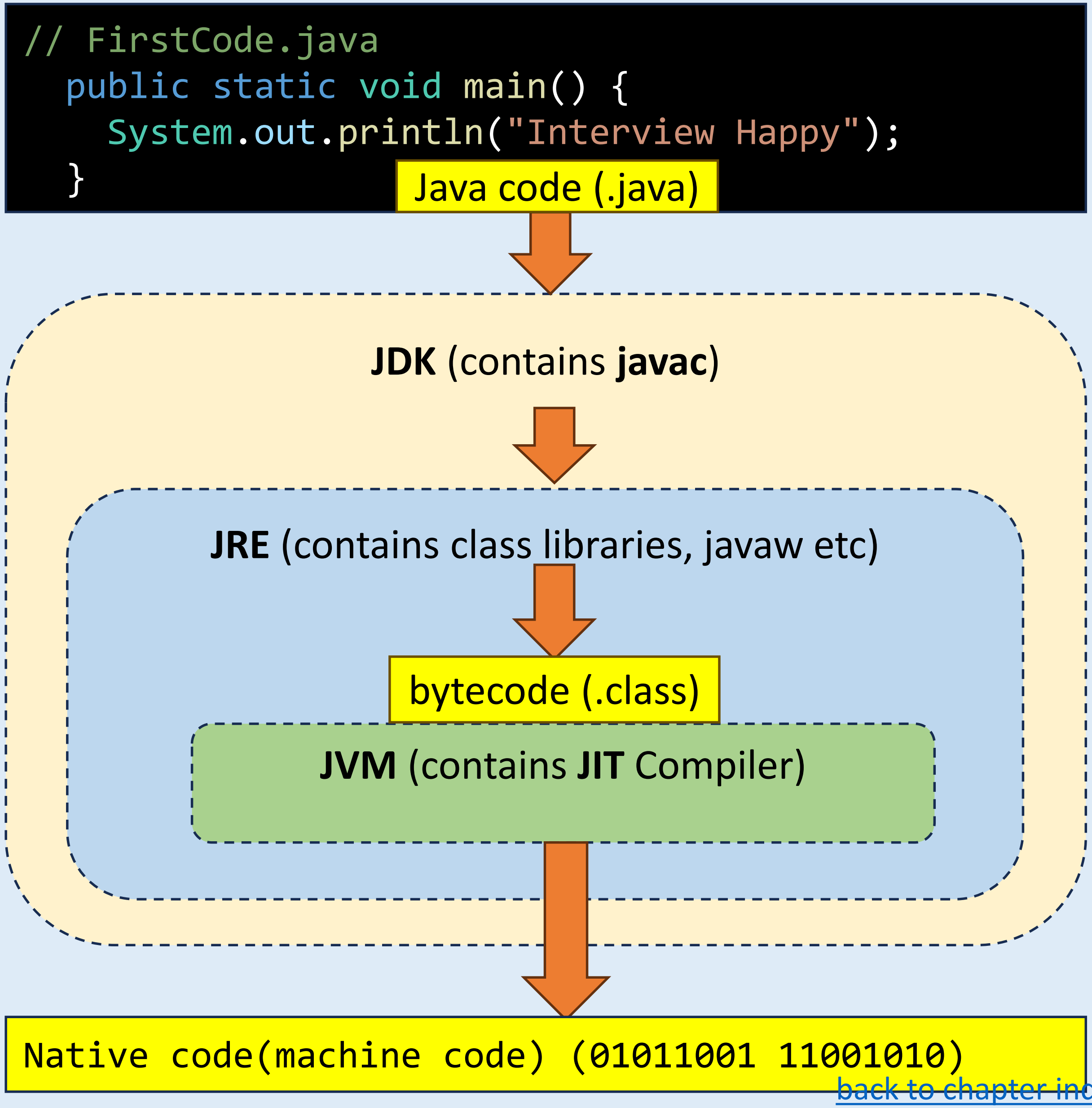
# Q. Can you have an else statement without an if statement in Java?

- **No**, you cannot have an else statement without a preceding if statement in Java.

back to chapter index

# Q. Can a Java program run without JDK or JRE or JVM installed on the system?

- **No**, a Java program cannot run without the JDK or JRE or JVM installed on the system.



```
// FirstCode.java
  public static void main() {
    System.out.println("Interview Happy");
  }
```

Java code (.java)

**JDK** (contains **javac**)

**JRE** (contains class libraries, javaw etc)

bytecode (.class)

**JVM** (contains **JIT** Compiler)

Native code(machine code) (01011001 11001010)

back to chapter index

# Q. What data type would you use to store a single character?

- **char** data type is used to store a single character.

```
public class Main {

  public static void main() {
    char letter = 'A';
    System.out.println(letter);
    // Output: A
  }
}
```

back to chapter index

# Q. What is the default value of a local variable in Java?

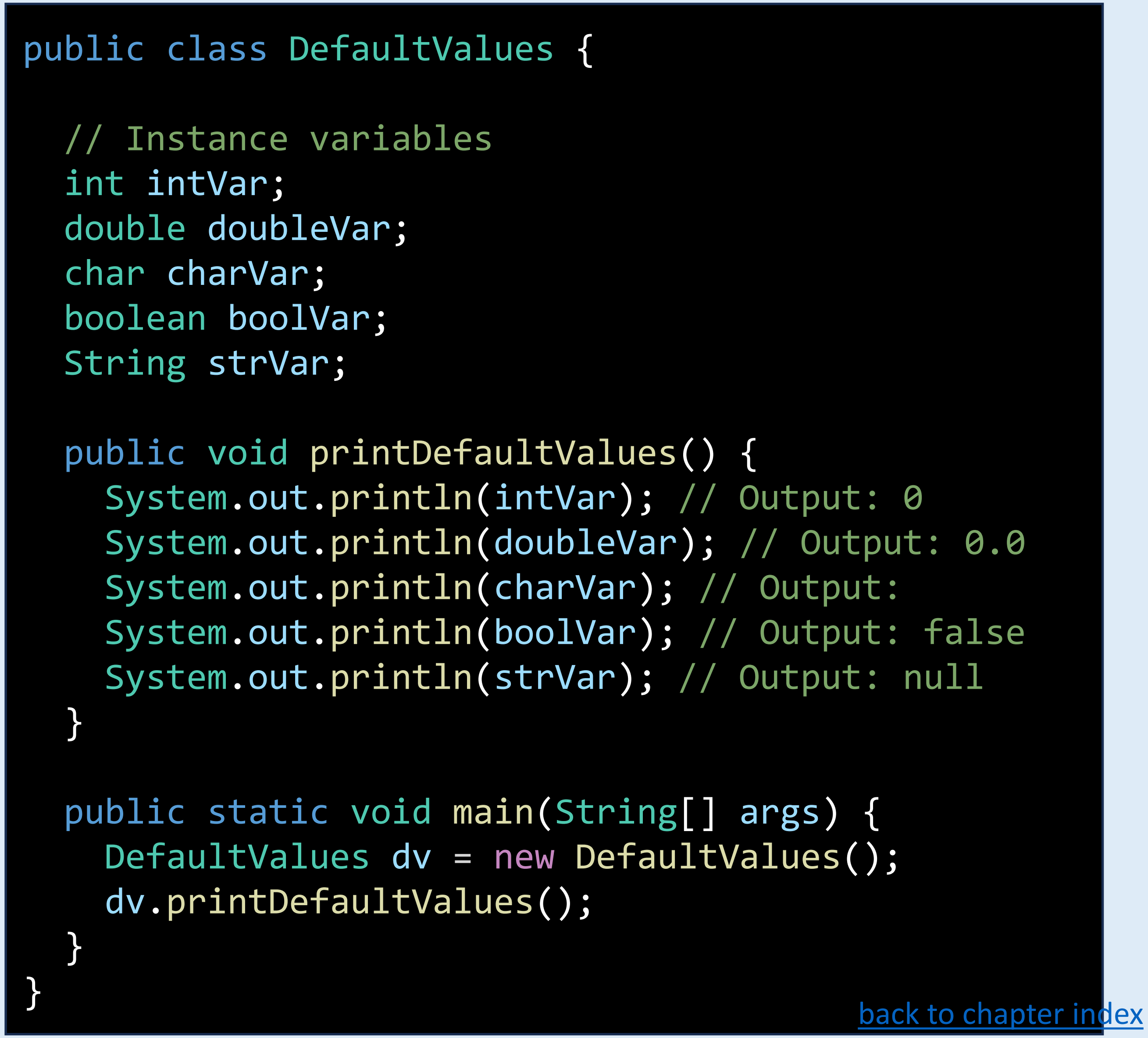
- Local variables in Java do not have a default value. They must be explicitly initialized before use; otherwise, the compiler will throw an error.

```
public class VariableExample {

  // Instance variable
  private int instanceVar = 5;

  public void display() {

    // Local variable
    int localVar = 10;
    System.out.println(instanceVar);
    System.out.println(localVar);
  }
}
```

back to chapter index

# Q. What is the default value of instance variables in Java?

- Instance variables in Java have a default value.
- ✓ int => 0
- ✓ double => 0.0
- ✓ char => blank
- ✓ boolean => false
- ✓ String => null

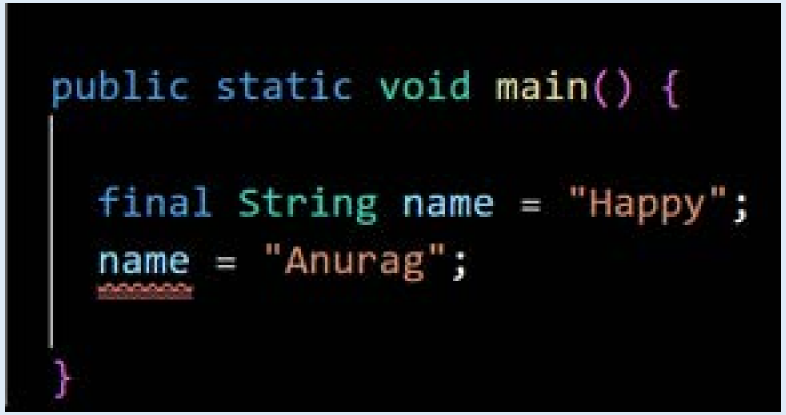


```java
public class DefaultValues {

  // Instance variables
  int intVar;
  double doubleVar;
  char charVar;
  boolean boolVar;
  String strVar;

  public void printDefaultValues() {
    System.out.println(intVar); // Output: 0
    System.out.println(doubleVar); // Output: 0.0
    System.out.println(charVar); // Output:
    System.out.println(boolVar); // Output: false
    System.out.println(strVar); // Output: null
  }

  public static void main(String[] args) {
    DefaultValues dv = new DefaultValues();
    dv.printDefaultValues();
  }
}
```

back to chapter index

# Q. What keyword is used to declare a constant variable?

- **final keyword** is used to declare a constant variable, meaning that once it has been assigned a value, it cannot be changed.



```
public static void main() {

    final String name = "Happy";
    name = "Anurag";

}
```

back to chapter index

# Mock Interview

## OOPS - Classes, Objects, Access Specifiers, Getter-Setter & this keyword

Q. How can private methods in a superclass be accessed in a subclass?

Q. Can you explain a scenario where this keyword is required?

Q. What would happen if you don't explicitly define a package in a Java class?

Q. How to make a class private in Java?

Q. What happens if you define a setter method but not a getter for a class field?

Q. What is the difference between a field and a property in Java?

Back to main index

# Q. How can private methods in a superclass be accessed in a subclass?

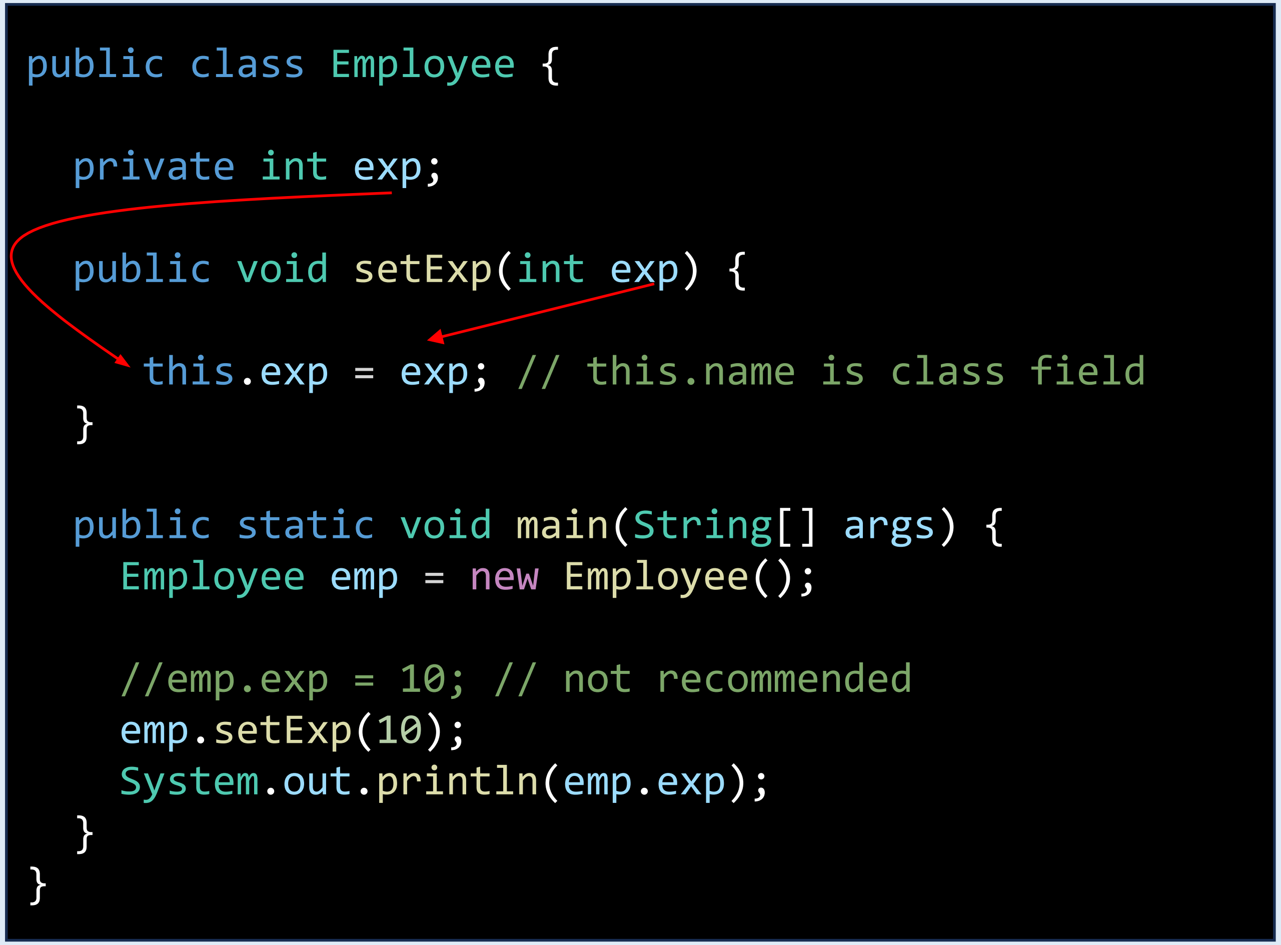
- Private methods in a superclass cannot be directly accessed or overridden by a subclass.
- Alternative approach: You can define a new method in the subclass with the same name as the superclass's private method. This new method is independent of the superclass's method.

```java
class Superclass {

  private void privateMethod() {
    System.out.println("Private method in Superclass");
  }
}

class Subclass extends Superclass {

  private void privateMethod() { // Redefine in Subclass
    System.out.println("Private method in Subclass");
  }
}
```

back to chapter index

# Q. Can you explain a scenario where this keyword is required?

- this keyword is required when there is a naming conflict between instance variables and parameters, such as in setters or constructors.



```java
public class Employee {

  private int exp;

  public void setExp(int exp) {

     this.exp = exp; // this.name is class field
  }

  public static void main(String[] args) {
    Employee emp = new Employee();

    //emp.exp = 10; // not recommended
    emp.setExp(10);
    System.out.println(emp.exp);
  }
}
```

back to chapter index

# Q. What would happen if you don't explicitly define a package in a Java class?

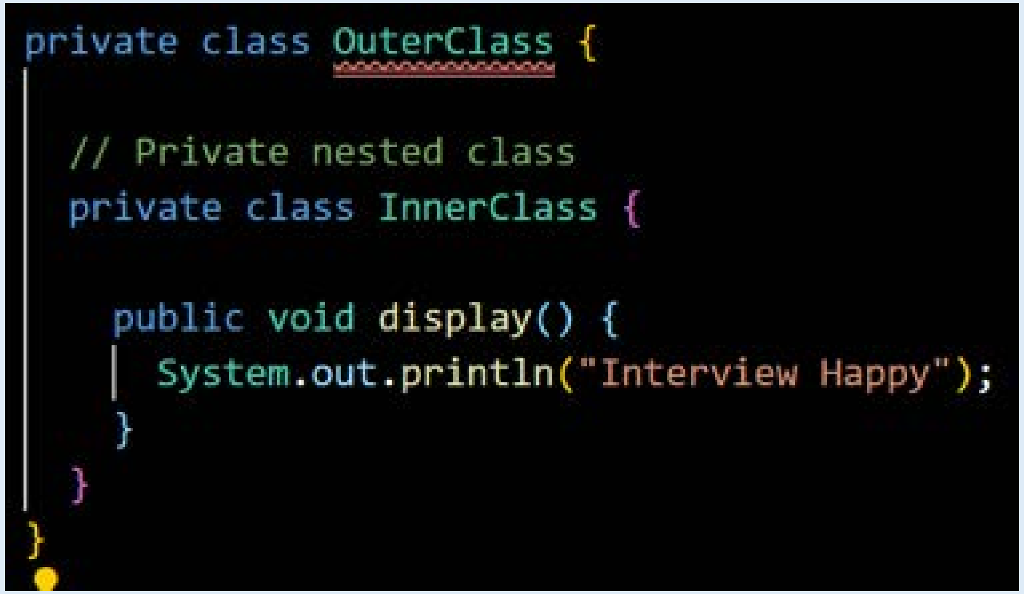
- If you don't explicitly define a package in a Java class, the class is placed in the **default package**.
- Classes in the default package have **package-private access**, which means they can only be accessed by other classes in the same default package.

```
~~package JavaFolder;~~

public class Main {

  public static void main() {

    System.out.println(“Interview Happy”);
  }
}
```

back to chapter index

# Q. How to make a class private in Java?

- A top-level class cannot be private.
- Only nested (inner) classes can be made private.



```java
private class OuterClass {

  // Private nested class
  private class InnerClass {

    public void display() {
      System.out.println("Interview Happy");
    }
  }
}
```

back to chapter index

# Q. What happens if you define a setter method but not a getter for a class field?

- If you define a setter method but not a getter for a class field in Java, the field can be modified but **not directly accessed** from outside the class.

```java
public class Employee {

  private int exp; // Field

  public void setExp(int exp) { // Setter method
    this.exp = exp;
  }

  public static void main(String[] args) {
    Employee emp = new Employee();

    emp.setExp(5); // Set exp using setter method

    // Compilation error
    // System.out.println(emp.getExp());
  }
}
```

back to chapter index

# Q. What is the difference between a field and a property in Java?

- Fields are variable declared inside a class.
- Properties(getter and setter methods) are abstraction of the fields.

```java
public class Employee {

  private int exp; // Field

  public int getExp() { // Getter method(Property)
    return exp;
  }

  public void setExp(int exp) {// Setter method(Property)
    this.exp = exp;
  }

  public static void main(String[] args) {
    Employee emp = new Employee();

    //emp.exp = 5; // Not recommended
    emp.setExp(5); // Set exp using setter method

    System.out.println(emp.getExp());
    // Output: 5
  }
}
```

back to chapter index

# Mock Interview 💡

## OOPS – Inheritance, Polymorphism, Encapsulation & Abstraction

Q. How can encapsulation be violated?

Q. What is the difference between extends and implements keywords in Java?

Q. Which is the most important and crucial OOPS principle in Java?

Q. Is protected field is accessible by any class within the same package?

Q. Which type of polymorphism needs inheritance?

Q. Explain the difference between static binding and dynamic binding in Java.

Q. What is the benefit of using polymorphism in Java applications?

Q. What happens if you don't use @override annotation in method overriding?

Back to main index

# Q. How can encapsulation be violated?

- Encapsulation can be violated by making **class fields public,** which allows any external code(main method/ other classes/ client code) to access and modify them directly.

```java
public class Employee {

  public int exp; // Field
  // private int exp; // Field

  // public int getExp() { // Getter method
  //  return exp;
  // }

  // public void setExp(int exp) { // Setter method
  //   this.exp = exp;
  // }

  public static void main(String[] args) {
    Employee emp = new Employee();

    emp.exp = 5; // Directly setting data(Not recommended)

    // emp.setExp(5); // Set exp using setter method
    // System.out.println(emp.getExp());
  }
}
```

back to chapter index

# Q. What is the difference between extends and implements keywords in Java?

- extends is used to inherit from a superclass (class to class inheritance).

```
class ParentClass {
  // class body
}

class ChildClass extends ParentClass {
  // class body
}
```

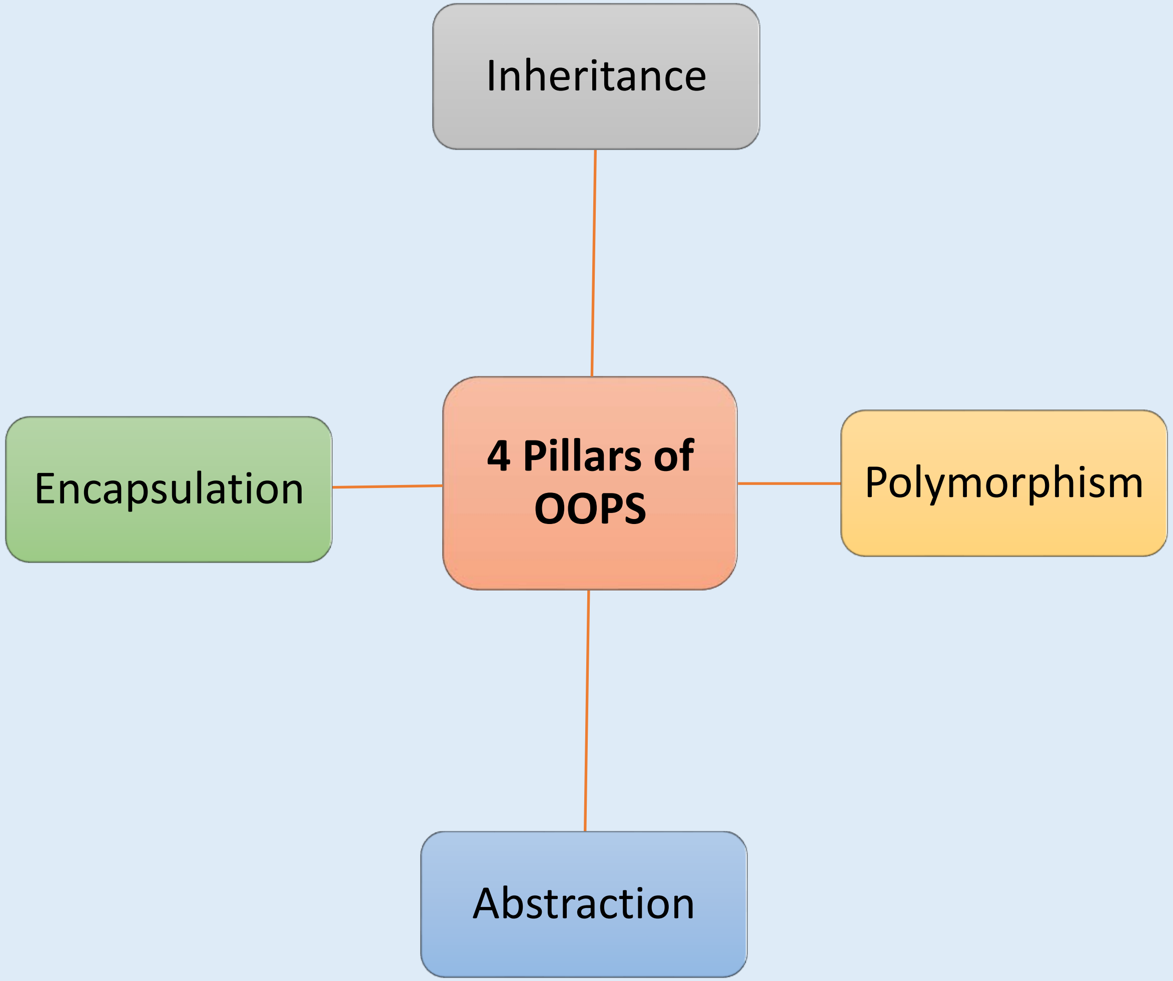
- implements is used to implement an interface (class to interface inheritance).

```
interface InterfaceName {
    // interface body
}

class ClassName implements InterfaceName {
  // class body
}
```

back to chapter index

# Q. Which is the most important and crucial OOPS principle in Java?

- **Encapsulation** because it allows you to hide the data which is very important from security perspective.



back to chapter index

# Q. Is protected field is accessible by any class within the same package?

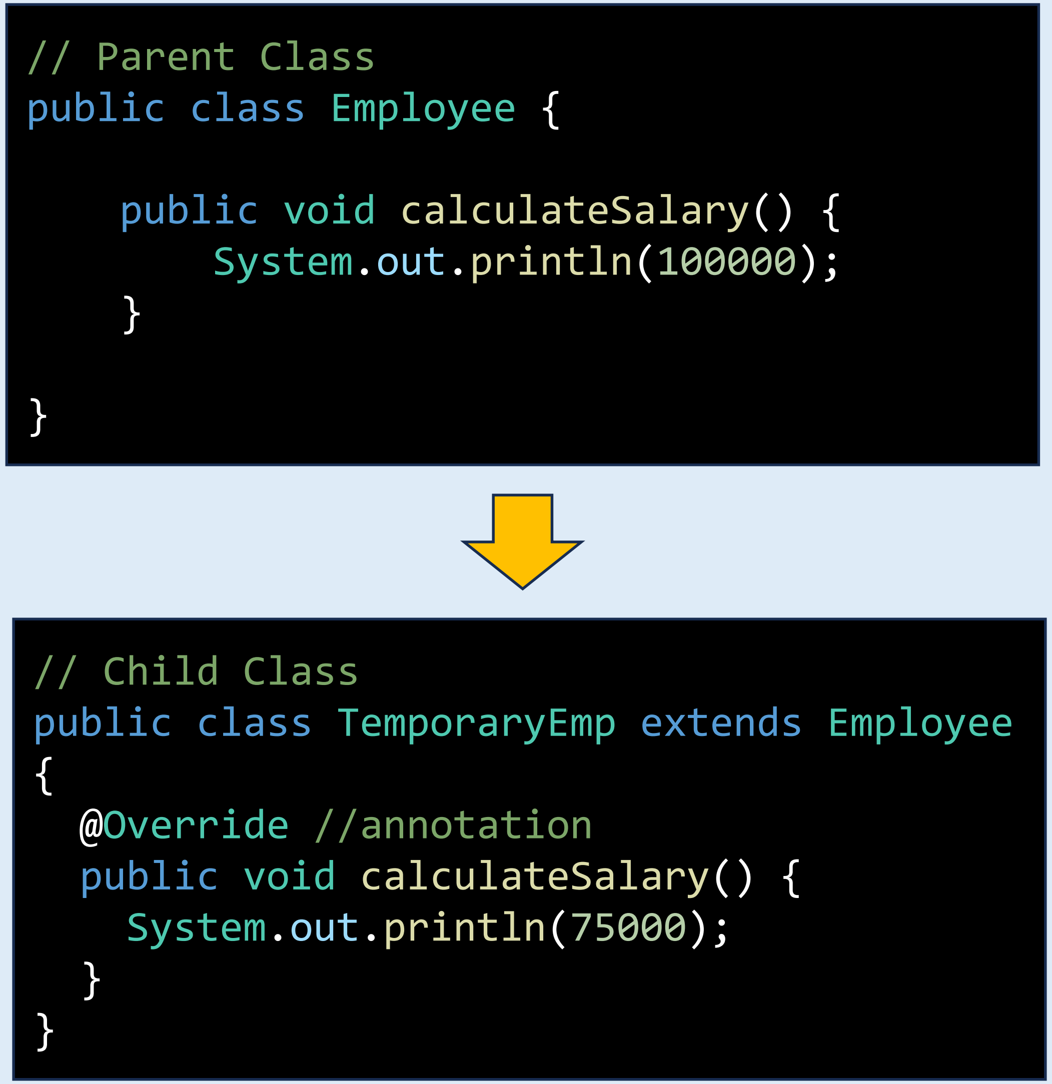
- The protected access specifier class members (fields, methods, constructors) are accessible within the class itself, its subclasses, and other classes in the same package.

| Access Specifier | Within Class | Within Package | Subclass (Same Package) | Subclass (Different Package) | Outside Package |
|---|---|---|---|---|---|
| public | Yes | Yes | Yes | Yes | Yes |
| protected | Yes | Yes | Yes | Yes | No |
| Default (no specifier) | Yes | Yes | Yes | No | No |
| private | Yes | No | No | No | No |

back to chapter index

# Q. Which type of polymorphism needs inheritance?

- Runtime polymorphism, also known as dynamic polymorphism or method overriding, requires inheritance.



```
// Parent Class
public class Employee {

    public void calculateSalary() {
        System.out.println(100000);
    }

}
```

```
// Child Class
public class TemporaryEmp extends Employee
{
  @Override //annotation
  public void calculateSalary() {
    System.out.println(75000);
  }
}
```

[back to chapter index](#)

# Q. Explain the difference between static binding and dynamic binding in Java.

- In Static binding (Early binding /method overloading / compile time polymorphism) method calls are resolved at compile-time.

```
public class Calculator {

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

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

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

- In Dynamic binding (Late binding /method overriding / run time polymorphism) method calls are resolved at run-time.

```
// Parent Class
public class Employee {
    public void calculateSalary() {
        System.out.println(100000);
    }
}
```

```
// Child Class
public class TemporaryEmp extends Employee
{
  @Override //annotation
  public void calculateSalary() {
    System.out.println(75000);
  }
}
```

back to chapter index

# Q. What is the benefit of using polymorphism in Java applications?

- Method overloading allows you to use the same method name with different parameters to perform various actions, which is good for **readability** and **flexibility**.

```
public class Calculator {

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

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

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

- Method overriding allows you to reuse the parent class method when needed and override it, when necessary, which is good for **reusability** and **flexibility**.

```
// Parent Class
public class Employee {
    public void calculateSalary() {
        System.out.println(100000);
    }
}
```

↓

```
// Child Class
public class TemporaryEmp extends Employee
{
  @Override //annotation
  public void calculateSalary() {
    System.out.println(75000);
  }
}
```

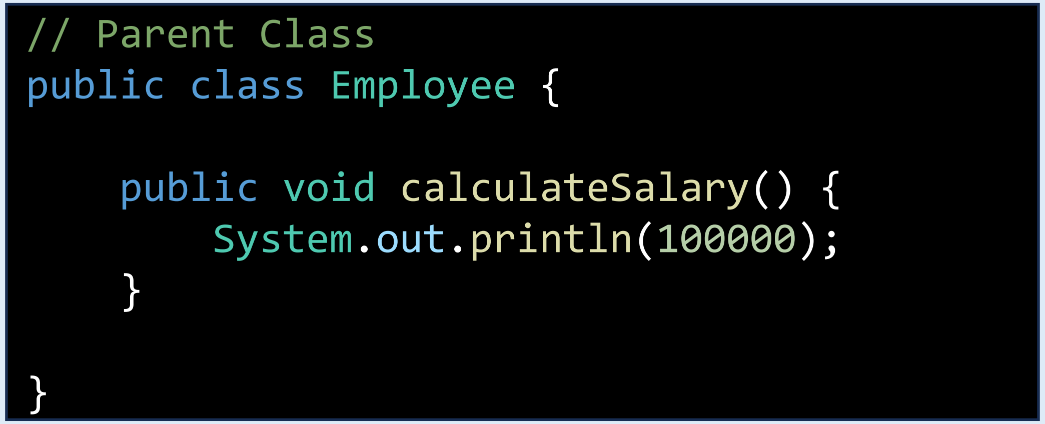
back to chapter index

# Q. What happens if you don't use @override annotation in method overriding?

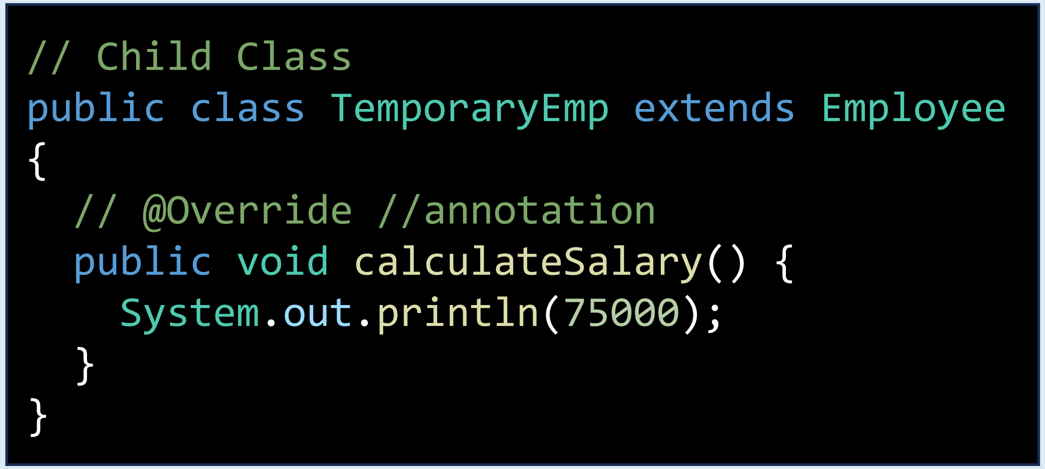
- If you don't use the @Override annotation, the code will still **compile and run correctly**.

- Benefits of Using @Override Annotation:

The compiler ensures that there's in no signature mismatch, like a typo or wrong parameters, it generates an error.



```java
// Parent Class
public class Employee {

    public void calculateSalary() {
        System.out.println(100000);
    }

}
```



```java
// Child Class
public class TemporaryEmp extends Employee
{
  // @Override //annotation
  public void calculateSalary() {
    System.out.println(75000);
  }
}
```

back to chapter index

# Mock Interview

## Abstract class & Interface, Constructors

Q. How can an interface inherit methods from another interface?

Q. How can you provide default behavior for methods in an interface?

Q. What is the impact of adding a new method to an existing interface?

Q. Can you use the final keyword with an abstract class? Why or why not?

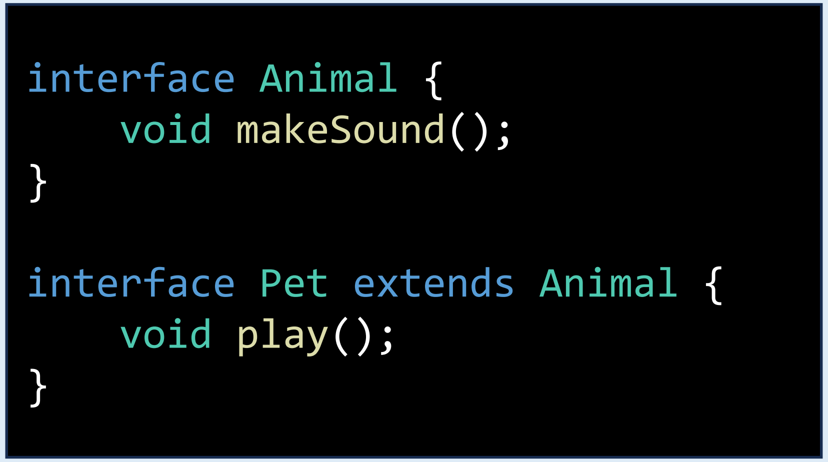
Q. What is a no-argument constructor?

Q. Can you store different data types in a single array? How?

Back to main index

# Q. How can an interface inherit methods from another interface?

- an interface can inherit methods from another interface by using the **extends keyword**.



```
interface Animal {
    void makeSound();
}

interface Pet extends Animal {
    void play();
}
```

back to chapter index

# Q. How can you provide default behavior for methods in an interface?

- You can provide default behavior for methods in an interface by using the **default method**.

```
public interface Vehicle {
  // Abstract method
  void start();

  // Default method
  default void stop() {
    System.out.println("Stop...");
  }
}
```

```
public class Car implements Vehicle {

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

  public static void main(String[] args) {
    Car myCar = new Car();
    myCar.start(); // Output: Start...
    myCar.stop(); // Output: Stop...
  }
}
```

back to chapter index

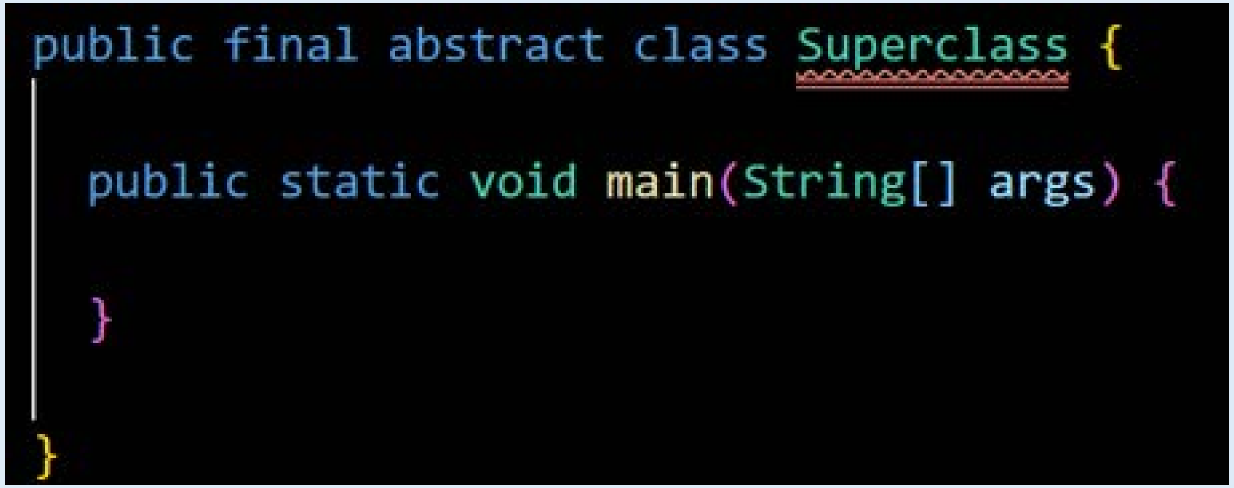
# Q. What is the impact of adding a new method to an existing interface?

- All existing classes that implement the interface will break unless you provide an implementation for the new method.

back to chapter index

# Q. Can you use the final keyword with an abstract class? Why or why not?

- No, you cannot use the final keyword with an abstract class in Java.
- Abstract classes are used as super class, but the final keyword prevents a class from being superclass. Therefore, these two concepts are contradictory.

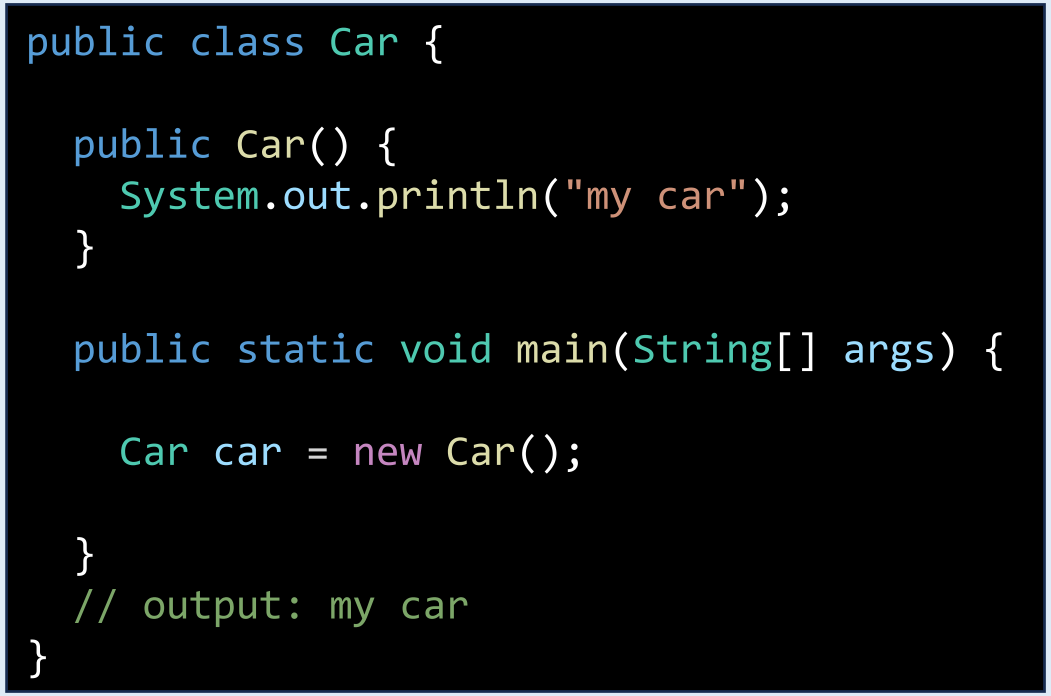


```
public final abstract class Superclass {

    public static void main(String[] args) {

    }

}
```

back to chapter index

# Q. What is a no-argument constructor?

- A no-argument constructor in Java is a constructor that does not take any parameters.



```java
public class Car {

  public Car() {
    System.out.println("my car");
  }

  public static void main(String[] args) {

    Car car = new Car();

  }
  // output: my car
}
```

back to chapter index

# Q. Can you store different data types in a single array? How?

- Standard arrays can only hold elements of the same type.
- However, you can achieve this by using an array of objects

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

  Object[] mixedArray = new Object[5];

  mixedArray[0] = "Hello";        // String
  mixedArray[1] = 123;            // Integer
  mixedArray[2] = 45.67;          // Double
  mixedArray[3] = 'A';            // Character
  mixedArray[4] = true;           // Boolean

  for (Object obj : mixedArray) {
      System.out.println(obj);
  }
}
```

back to chapter index

# Mock Interview

## Exception Handling

Q. What happens if a return statement is used inside try block of a try-catch-finally?

Q. What happens if an exception is thrown in the finally block?

Q. Can you explain how custom exceptions are created and used in Java?

Q. How do you handle multiple exceptions in a single catch block?

Q. What happens if an exception is not handled in Java?

Q. How can you suppress exceptions in Java?

Back to main index

# Q. What happens if a return statement is used inside try block of a try-catch-finally?

- When a return statement is used inside the try block of a try-catch-finally, the finally block will still be executed before the method returns a value.

```java
public class Main {

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

  public static int testMethod() {
    try {
      System.out.println("In try block");
      return 1;
    } catch (Exception e) {
      System.out.println("In catch block");
      return 2;
    } finally {
      System.out.println("In finally block");
    }
  }// Output: In try block  In finally block  1
}
```

# Q. What happens if an exception is thrown in the finally block?

- If an exception is thrown in the finally block in Java, it can override any previous exception thrown in the try or catch block.

```java
public class Main {

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

  public static void testMethod() {
    try {
      throw new RuntimeException("Exception in try block");
    } catch (Exception e) {
      throw new RuntimeException("Exception in catch block");
    } finally {
      throw new RuntimeException("Exception in finally block");
    }
  } // Output: Exception in finally block
}
```

back to chapter index

# Q. Can you explain how custom exceptions are created and used in Java?

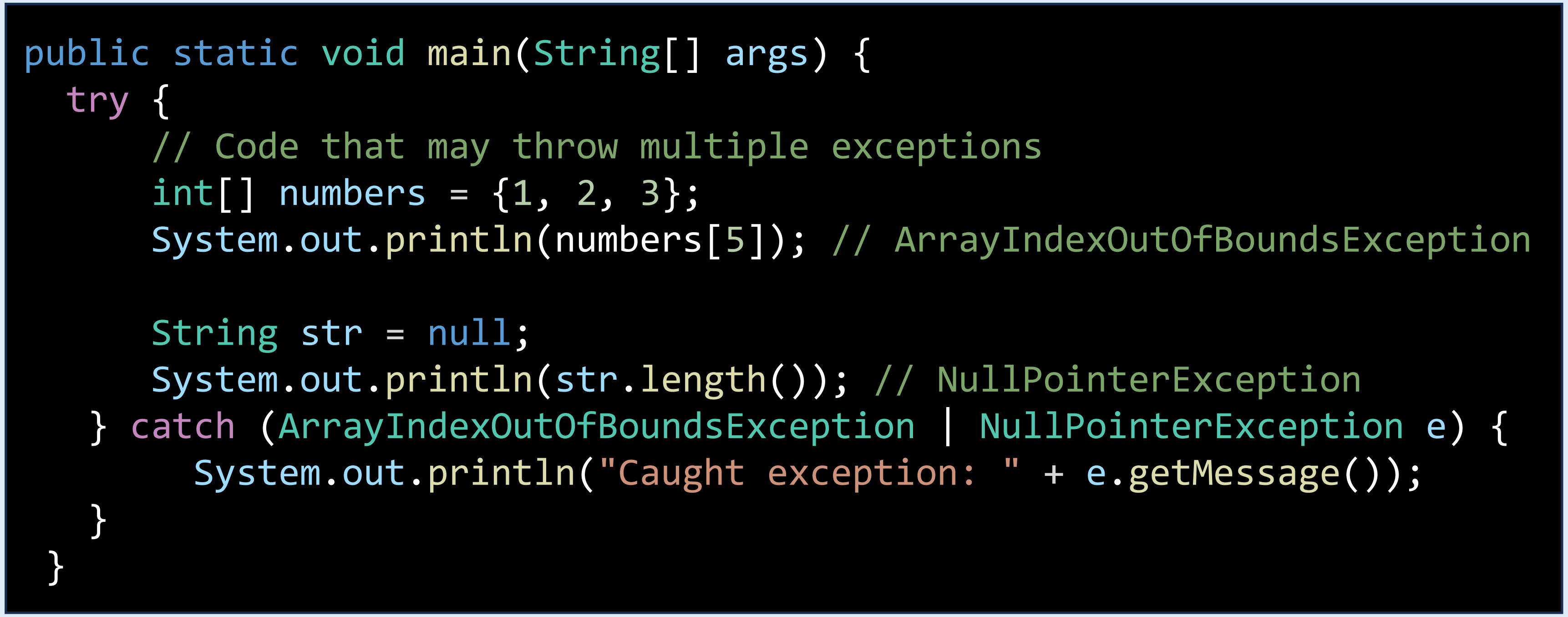
- Custom exceptions are user-defined exceptions that can be created  by extending the **Exception class** (for checked exceptions) or the **RuntimeException class** (for unchecked exceptions).

```java
class CustomCheckedException extends Exception {

  public CustomCheckedException(String message) {
    super(message);
  }
}

public class Main {

  public static void main(String[] args) {
    try {
      validateAge(15);
    } catch (CustomCheckedException e) {
      System.out.println(e.getMessage());
    }
  }
}
```

back to chapter index

# Q. How do you handle multiple exceptions in a single catch block?

- You can handle multiple exceptions in a single catch block using **multi-catch syntax** (separating the exception types with a vertical bar (|)).



```java
public static void main(String[] args) {
  try {
      // Code that may throw multiple exceptions
      int[] numbers = {1, 2, 3};
      System.out.println(numbers[5]); // ArrayIndexOutOfBoundsException

      String str = null;
      System.out.println(str.length()); // NullPointerException
   } catch (ArrayIndexOutOfBoundsException | NullPointerException e) {
        System.out.println("Caught exception: " + e.getMessage());
   }
 }
```

back to chapter index

# Q. What happens if an exception is not handled in Java?

- Handling exceptions properly ensures that your program can gracefully manage error conditions and continue to operate or shut down cleanly.

back to chapter index

# Q. How can you suppress exceptions in Java?

- You can suppress exceptions by catching exceptions and choose not to take any action.

```
public class Main {

  public static void main(String[] args) {
    try {
      int result = 10 / 0;
    } catch (ArithmeticException e) {
      // Exception is caught and suppressed
      method1(); // not recommended
    }
  }

  public static void method1() {
    System.out.println("from catch");
  } // from catch
}
```

back to chapter index

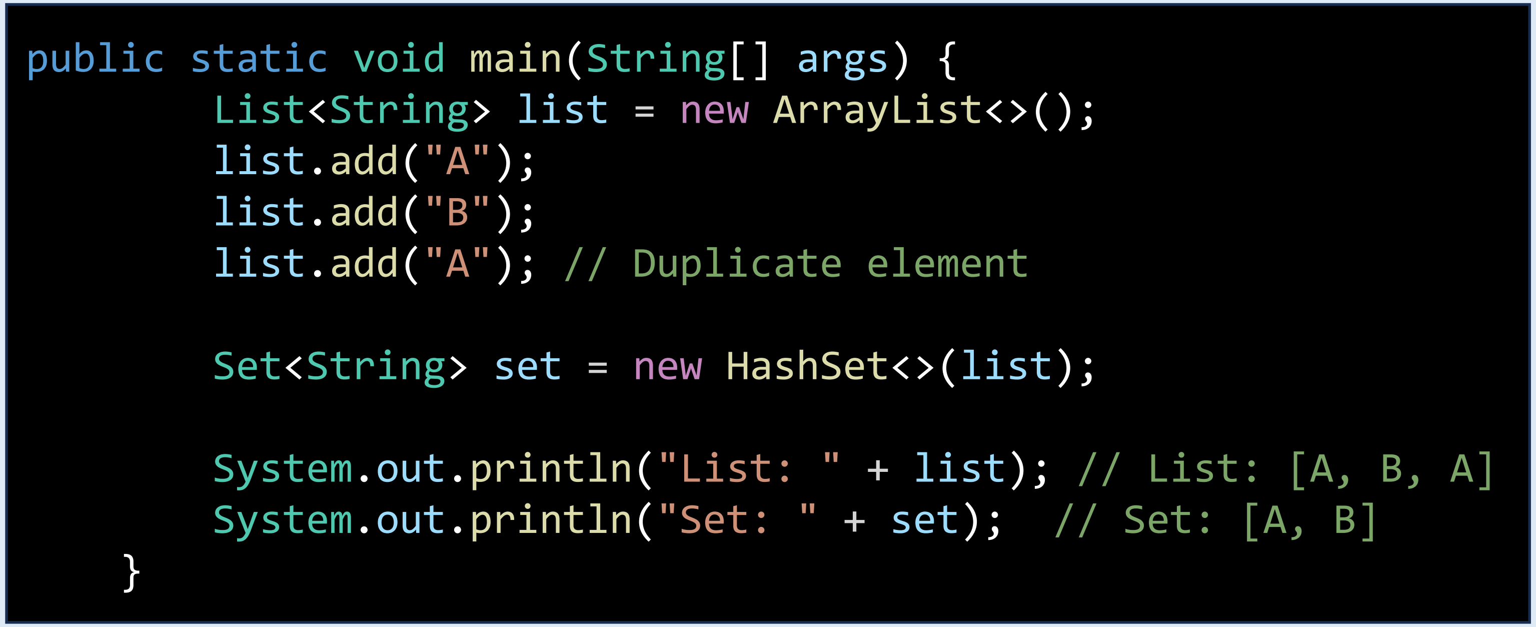
# Mock Interview

## Collections

Back to main index

# Q. Can you convert a List to a Set and vice versa? How?

- Use new HashSet<>(list) for List to Set, and new ArrayList<>(set) for Set to List.



```java
public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("A");
        list.add("B");
        list.add("A"); // Duplicate element

        Set<String> set = new HashSet<>(list);

        System.out.println("List: " + list); // List: [A, B, A]
        System.out.println("Set: " + set);  // Set: [A, B]
    }
```

[back to chapter index](#)

# Q. How do you remove duplicates from an ArrayList?

- By converting the ArrayList to a Set and back to ArrayList then.

back to chapter index

# Q. How do you decide between using a List, Set, or Map?

- Use List for ordered collections with duplicates, Set for unique elements, and Map for key-value pairs.

back to chapter index

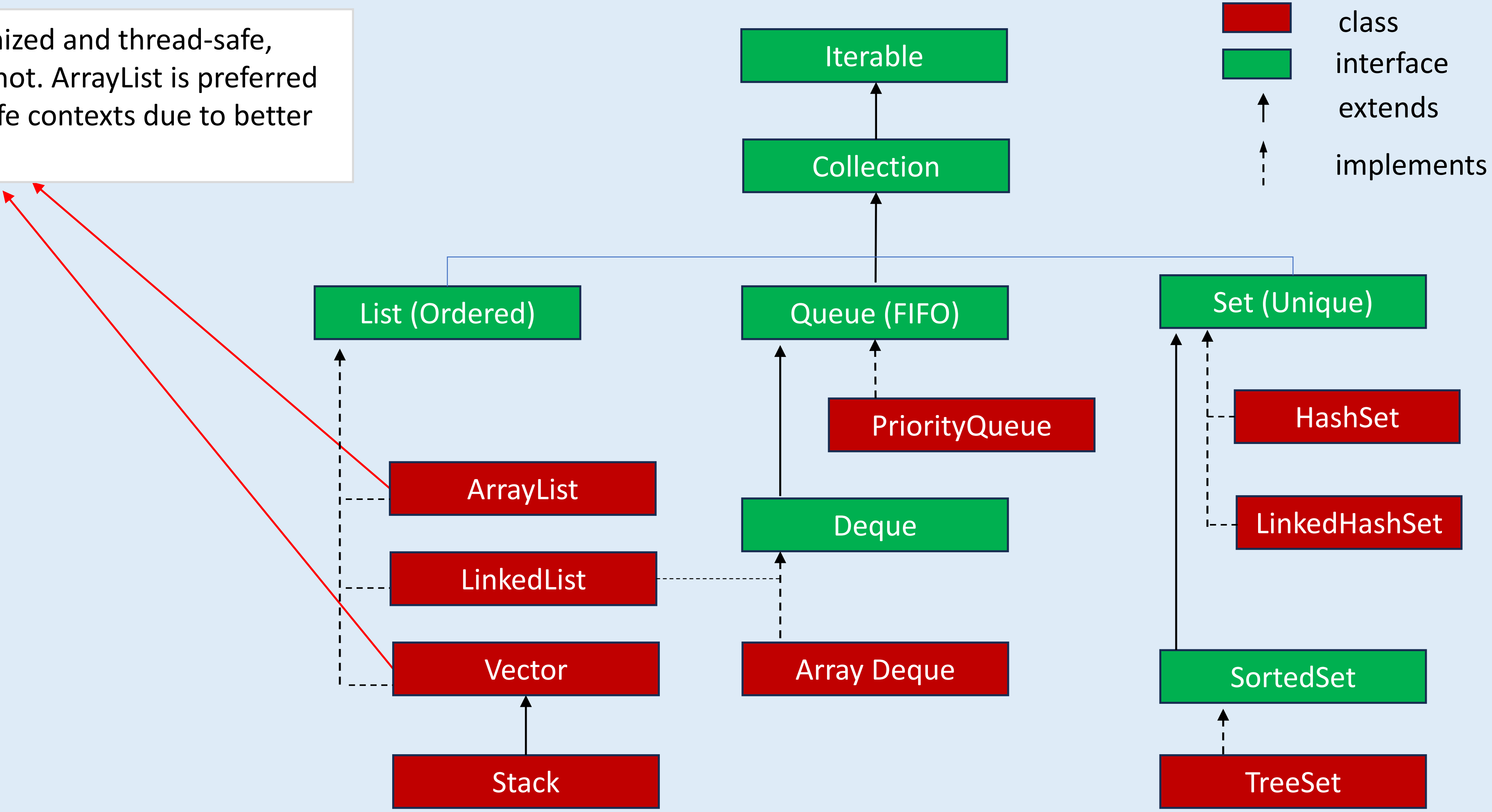
# Q. What is the purpose of the Collections utility class?

- The Collections class provides static methods for common operations like sorting, searching, and synchronizing collections.

back to chapter index

# Q. What is the difference between Vector and ArrayList?

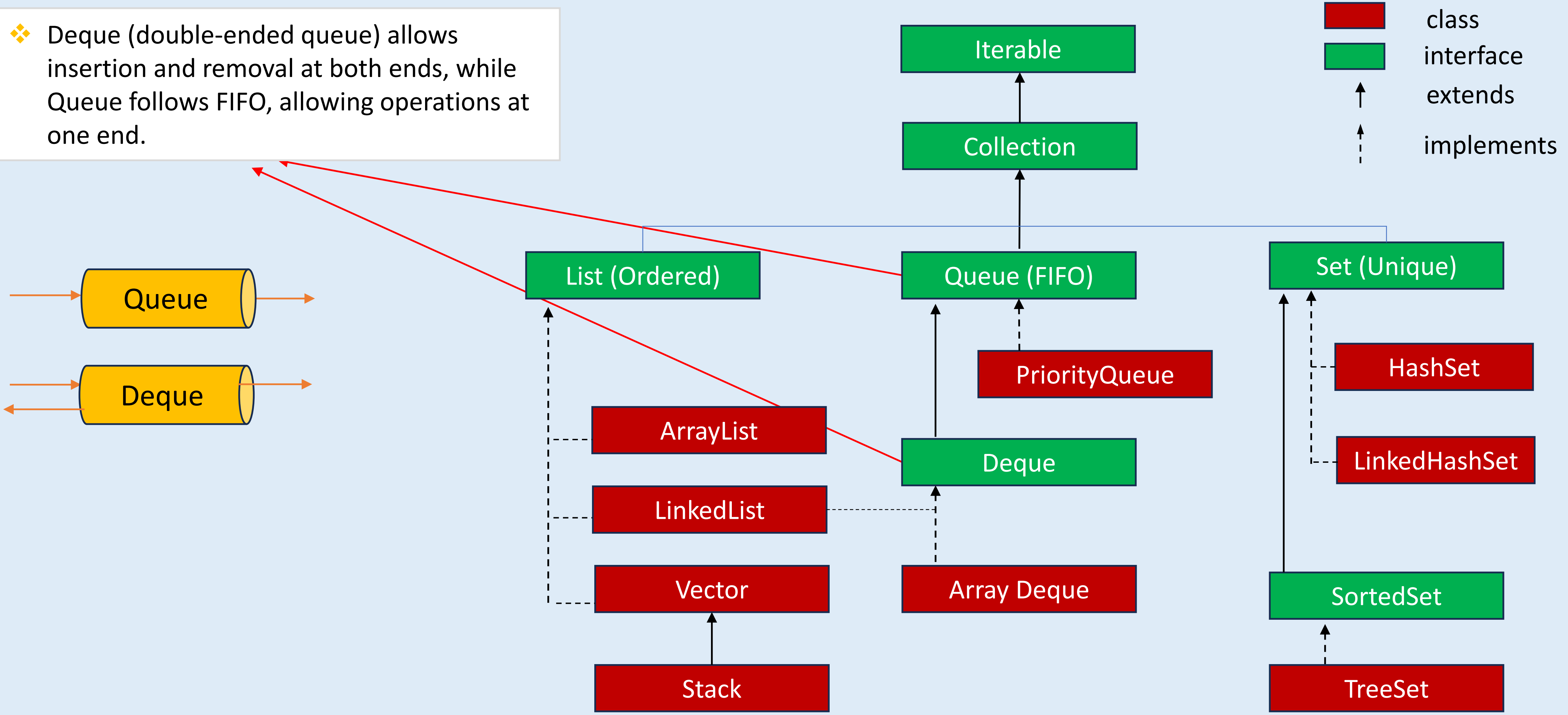
- Vector is synchronized and thread-safe, while ArrayList is not. ArrayList is preferred for non-thread-safe contexts due to better performance.



back to chapter index

# Q. Explain the difference between Deque and Queue?

- Deque (double-ended queue) allows insertion and removal at both ends, while Queue follows FIFO, allowing operations at one end.



back to chapter index

# Mock Interview

## Multithreading

Q. What is the difference between start() and run() method?

Q. What is the meaning of Thread safe?

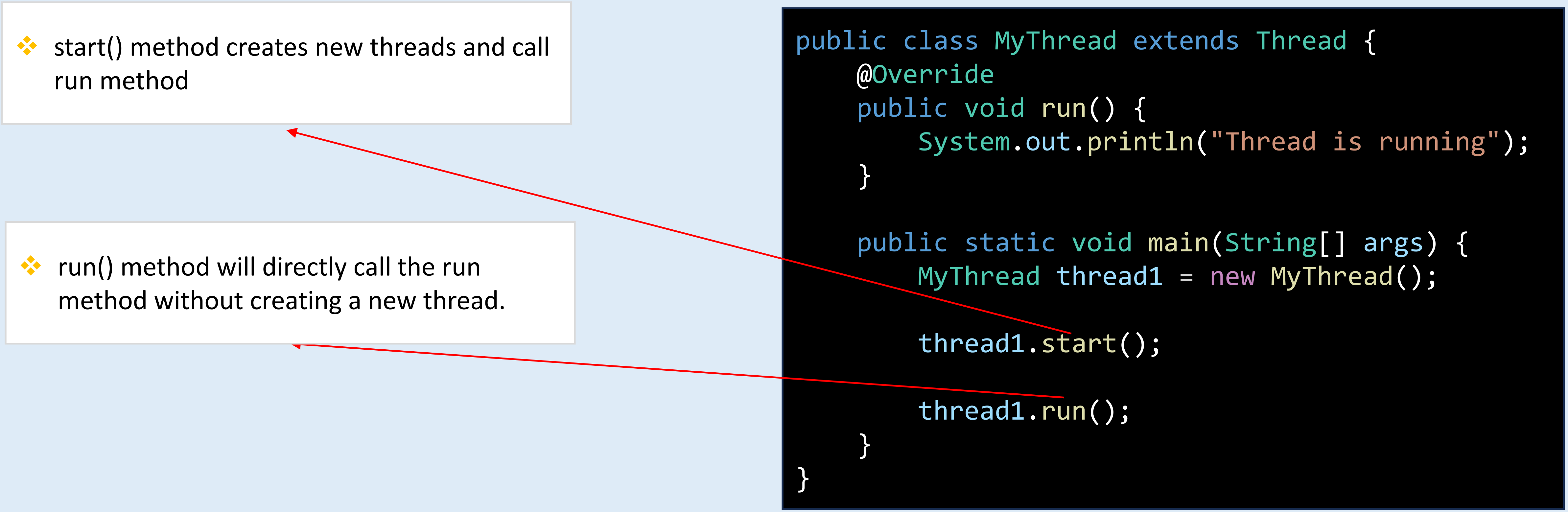
Q. What is thread pool?

Q. What is a synchronized block in Java?

Q. What is the difference between Runnable and Callable interface?

Back to main index

# Q. What is the difference between start() and run() method?



- start() method creates new threads and call run method

- run() method will directly call the run method without creating a new thread.

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

    public static void main(String[] args) {
        MyThread thread1 = new MyThread();

        thread1.start();

        thread1.run();
    }
}
```

back to chapter index

# Q. What is the meaning of Thread safe?

- Thread-safe means that a piece of code or a class or a function can be safely invoked and used by **multiple threads simultaneously without causing any errors** (race conditions, data corruption, or unexpected behavior).

back to chapter index

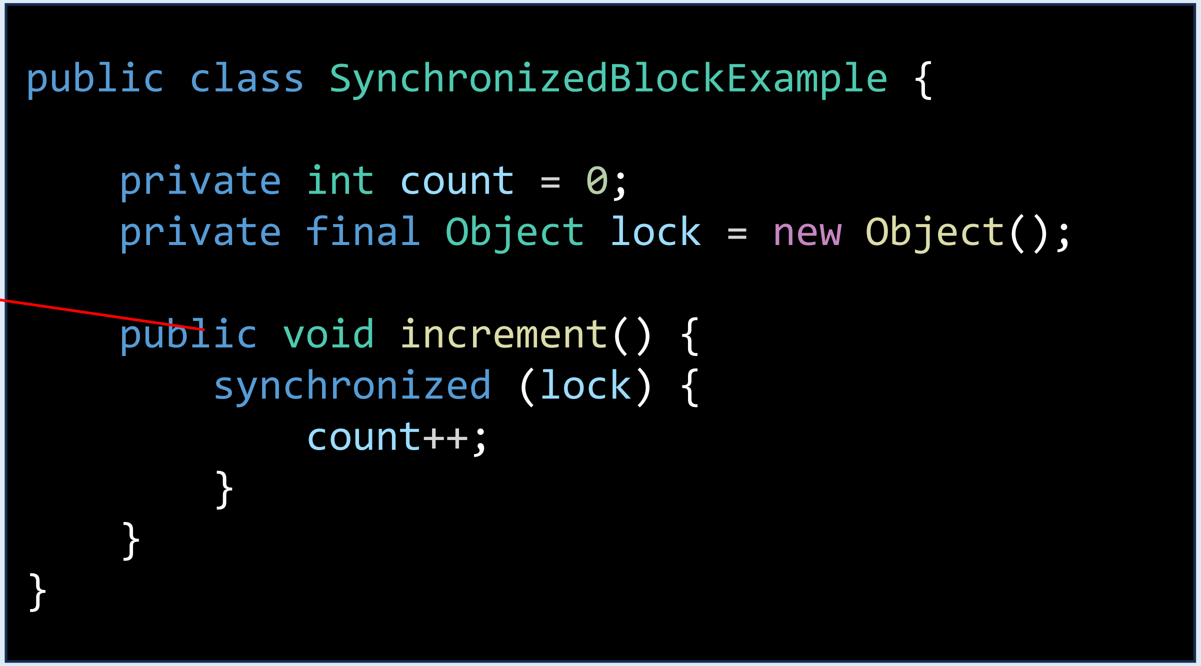
# Q. What is thread pool?

- A thread pool maintains a pool of threads that are ready to execute tasks.

back to chapter index

# Q. What is a synchronized block in Java?

- synchronized (lock) ensures that only one thread at a time can execute the block of code

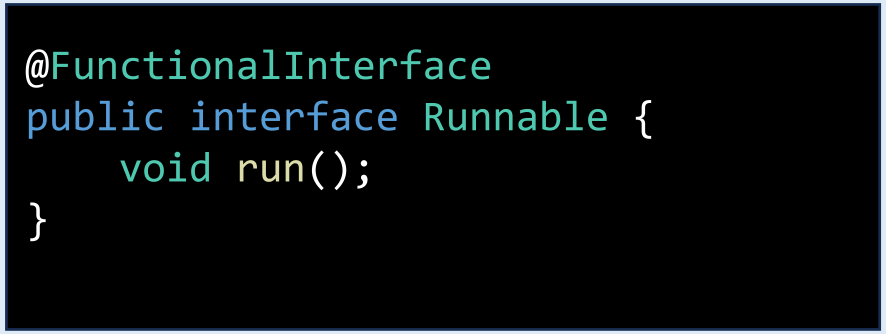


```java
public class SynchronizedBlockExample {

    private int count = 0;
    private final Object lock = new Object();

    public void increment() {
        synchronized (lock) {
            count++;
        }
    }
}
```

back to chapter index

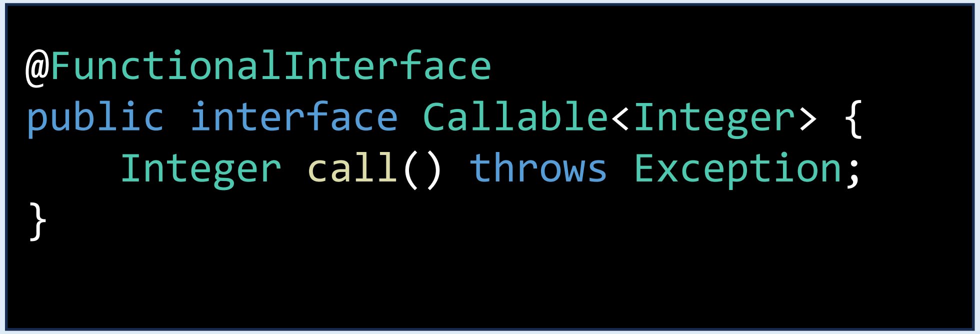
# Q. What is the difference between Runnable and Callable interface?

❖ Runnable does not return a result and cannot throw checked exceptions.



```
@FunctionalInterface
public interface Runnable {
    void run();
}
```

❖ Callable return a result and can throw checked exceptions.



```
@FunctionalInterface
public interface Callable<Integer> {
    Integer call() throws Exception;
}
```

back to chapter index

# Mock Interview

## Generics

Q. How do you define a generic method in Java?

Q. Can you create generic interfaces?

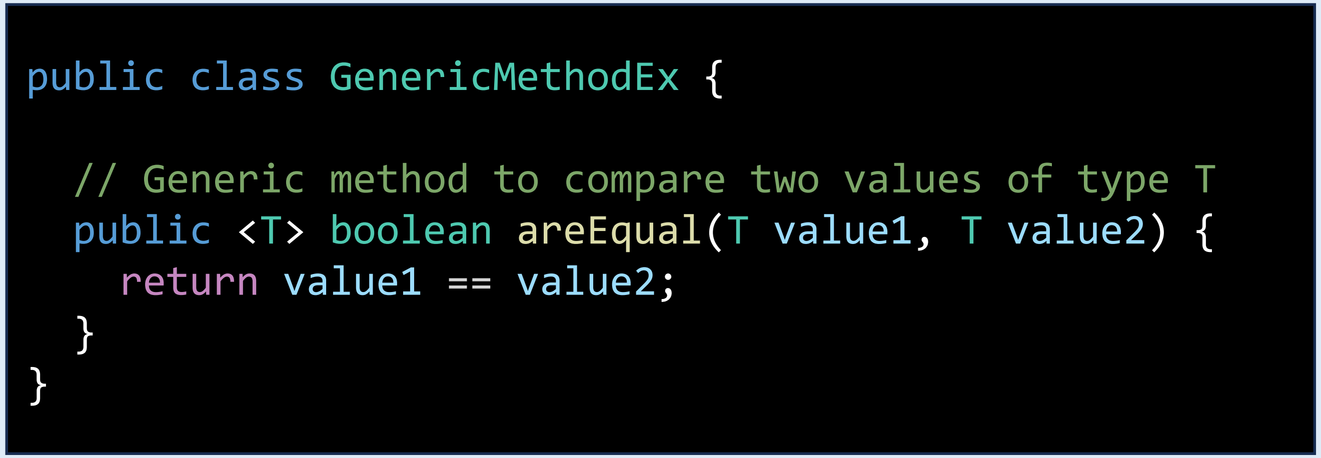
Q. Can you create generic constructor?

Q. What is the difference between List<Object> and List<?>?

Back to main index

# Q. How do you define a generic method in Java?

- A generic method is defined with type parameters, which appear before the return type.

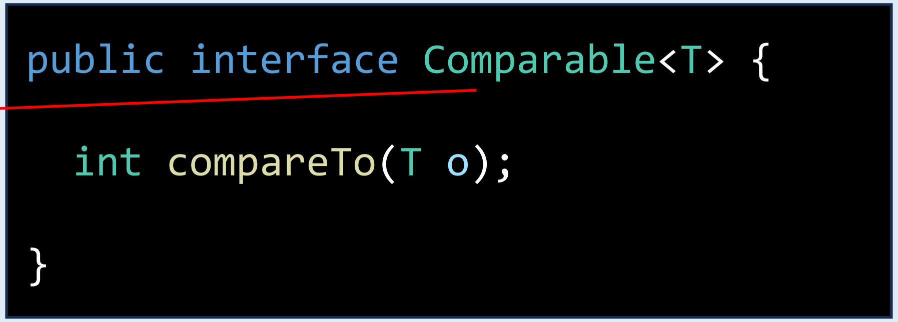


```java
public class GenericMethodEx {

  // Generic method to compare two values of type T
  public <T> boolean areEqual(T value1, T value2) {
    return value1 == value2;
  }
}
```

back to chapter index

# Q. Can you create generic interfaces?

- Yes, we can create generic interfaces.

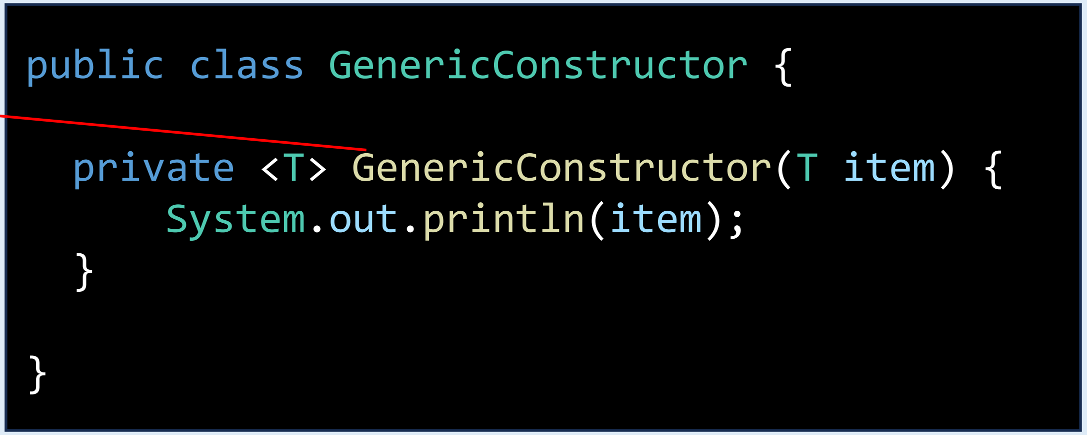


```
public interface Comparable<T> {

  int compareTo(T o);

}
```

back to chapter index

# Q. Can you create generic constructor?

- Yes, we can create generic constructor..



```
public class GenericConstructor {

  private <T> GenericConstructor(T item) {
      System.out.println(item);
  }

}
```

back to chapter index

# Q. What is the difference between List<Object> and List<?>?

- List<Object> can hold any type of object and adding elements is allowed in it.
- Use List<Object> for manipulating list.

- List<?> is a wildcard that represents an unknown type and adding elements is not allowed in it.
- Use List<?> to perform read-only operations type safely.

```
public static void main(String[] args) {
    // List<Object> example
    List<Object> objectList = new ArrayList<>();
    // Adding elements to List<Object> is allowed
    objectList.add("String");
    objectList.add(123);

    // List<?> example
    List<?> wildcardList = new ArrayList<String>();
    // Adding elements to List<?> is not allowed (except null)
    // wildcardList.add("String"); // Compile-time error
    wildcardList.add(null); // This is allowed
  }
```

back to chapter index

# Mock Interview

## Types of Classes

Q. How can an inner class access the members of its outer class?

Q. Can a final class have final methods?

Q. Can a static nested class access instance variables of the outer class?

Q. Can enums have methods in Java?

Q. How can you iterate over all values of an enum in Java?

Q. When to use enums?

Q. What is the difference between a static and a non-static method in Java?

Q. How do static variables differ from instance variables in terms of memory allocation?

Q. Can a static method be overridden in Java?

Q. How can static methods and variables be accessed within the same class?

Back to main index

# Q. How can an inner class access the members of its outer class?

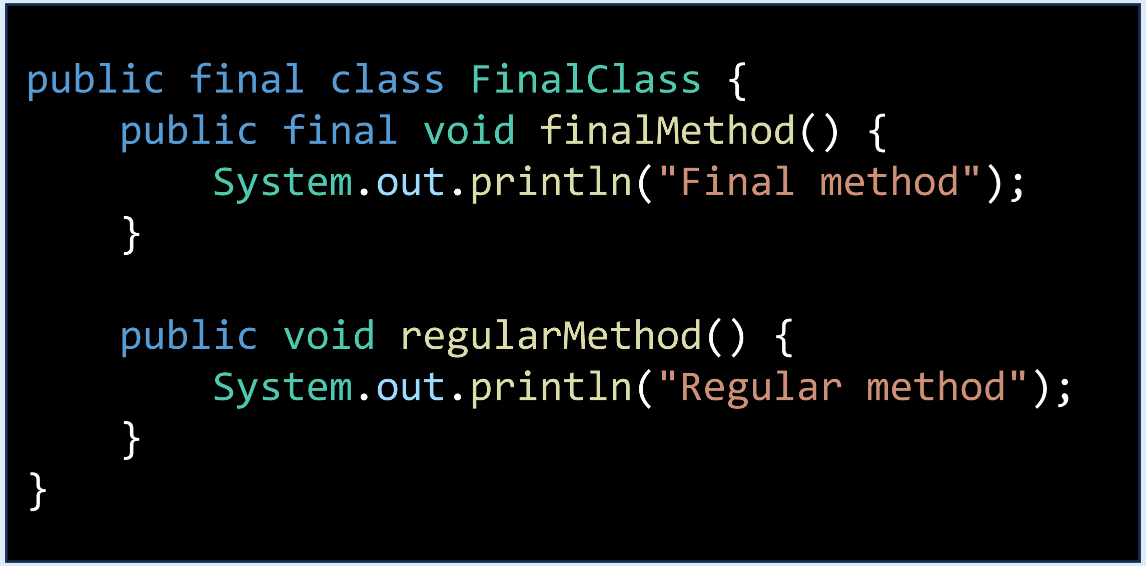
- An inner class can access all members (including private) of its outer class directly.

```
// Outer Class
public class OuterInnerClassEx {

  private int outerField = 10;

  // Member inner class
  public class InnerClass {

    public void displayOuterField() {
    // Accessing outer class field
      System.out.println(outerField);
    }
  }
}
```

back to chapter index

# Q. Can a final class have final methods?

- Yes, a final class can have final methods.
- However, declaring methods as final within a final class is unnecessary because a final class cannot be subclassed, so its methods cannot be overridden.



```java
public final class FinalClass {
    public final void finalMethod() {
        System.out.println("Final method");
    }

    public void regularMethod() {
        System.out.println("Regular method");
    }
}
```

back to chapter index

# Q. Can a static nested class access instance variables of the outer class?

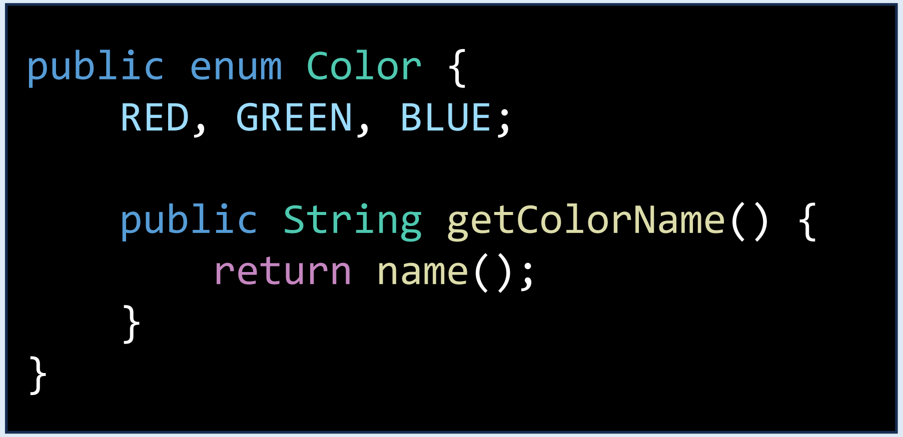
- No, a static nested class cannot directly access instance variables or methods of the outer class, it can only access the **static members**.

```
public class OuterClass {

  private int instanceVar = 10;
  private static int staticVar = 20;

  static class StaticNestedClass {

    void display() {
      // Can access static variable directly
      System.out.println(staticVar);

      // Cannot access instance variable directly
      // This will cause a compile-time error
      // System.out.println(instanceVar);

      // To access instance variable,
      // create an instance of the outer class
      OuterClass outer = new OuterClass();
      System.out.println(outer.instanceVar);
    }
  }
```

back to chapter index

# Q. Can enums have methods in Java?

- Yes, enums can have methods.

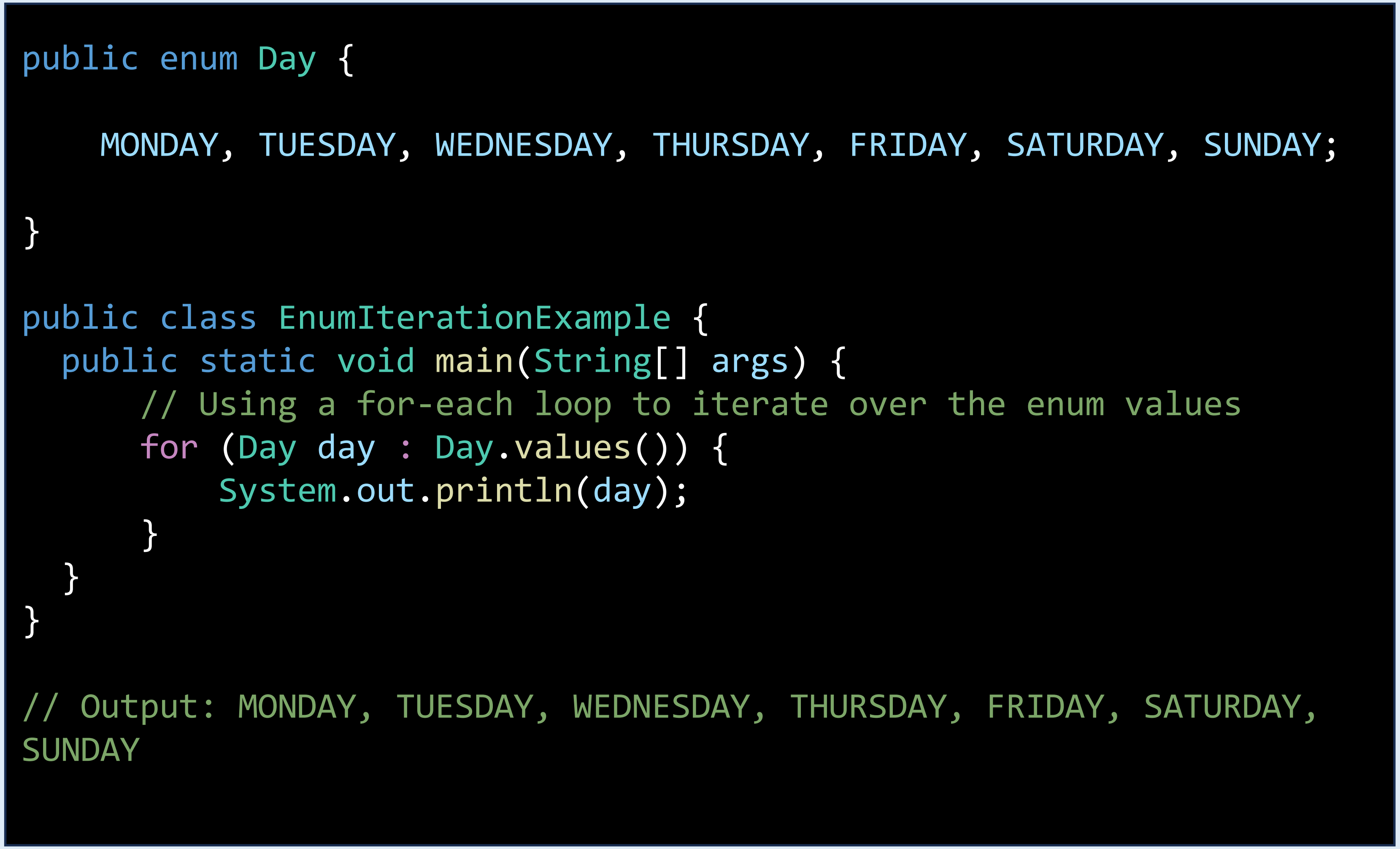


```java
public enum Color {
    RED, GREEN, BLUE;

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

back to chapter index

# Q. How can you iterate over all values of an enum in Java?

- You can iterate over all values of an enum in Java using the **values() method**, which returns an array of all the enum constants.



```java
public enum Day {

    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY;

}

public class EnumIterationExample {
  public static void main(String[] args) {
      // Using a for-each loop to iterate over the enum values
      for (Day day : Day.values()) {
          System.out.println(day);
      }
  }
}

// Output: MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY,
SUNDAY
```

back to chapter index

# Q. When to use enums?

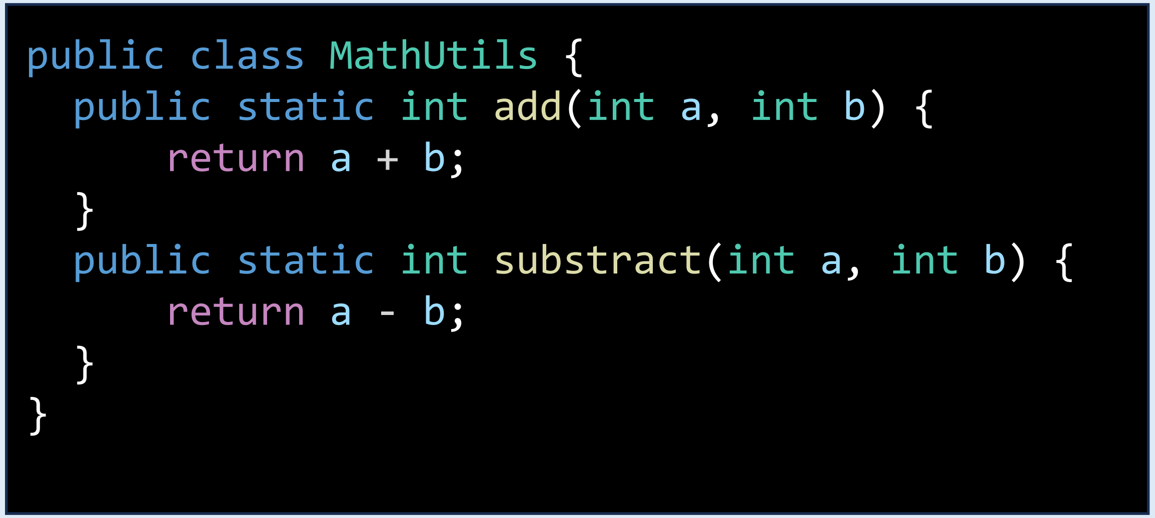
- Instead of constants, use enums.

```
// Instead of constants
public static final int MONDAY = 1;
public static final int TUESDAY = 2;

// Use Enums
public enum Day {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,
SATURDAY, SUNDAY;
}
```

back to chapter index

# Q. What is the difference between a static and a non-static method in Java?

- Static methods are invoked directly by the class name rather than by the instances of the class.



```java
public class MathUtils {
  public static int add(int a, int b) {
      return a + b;
  }
  public static int substract(int a, int b) {
      return a - b;
  }
}
```

back to chapter index

# Q. How do static variables differ from instance variables in terms of memory allocation?

- Static variables once initialized then exist for the entire duration of the program, from class loading until the program terminates.

back to chapter index

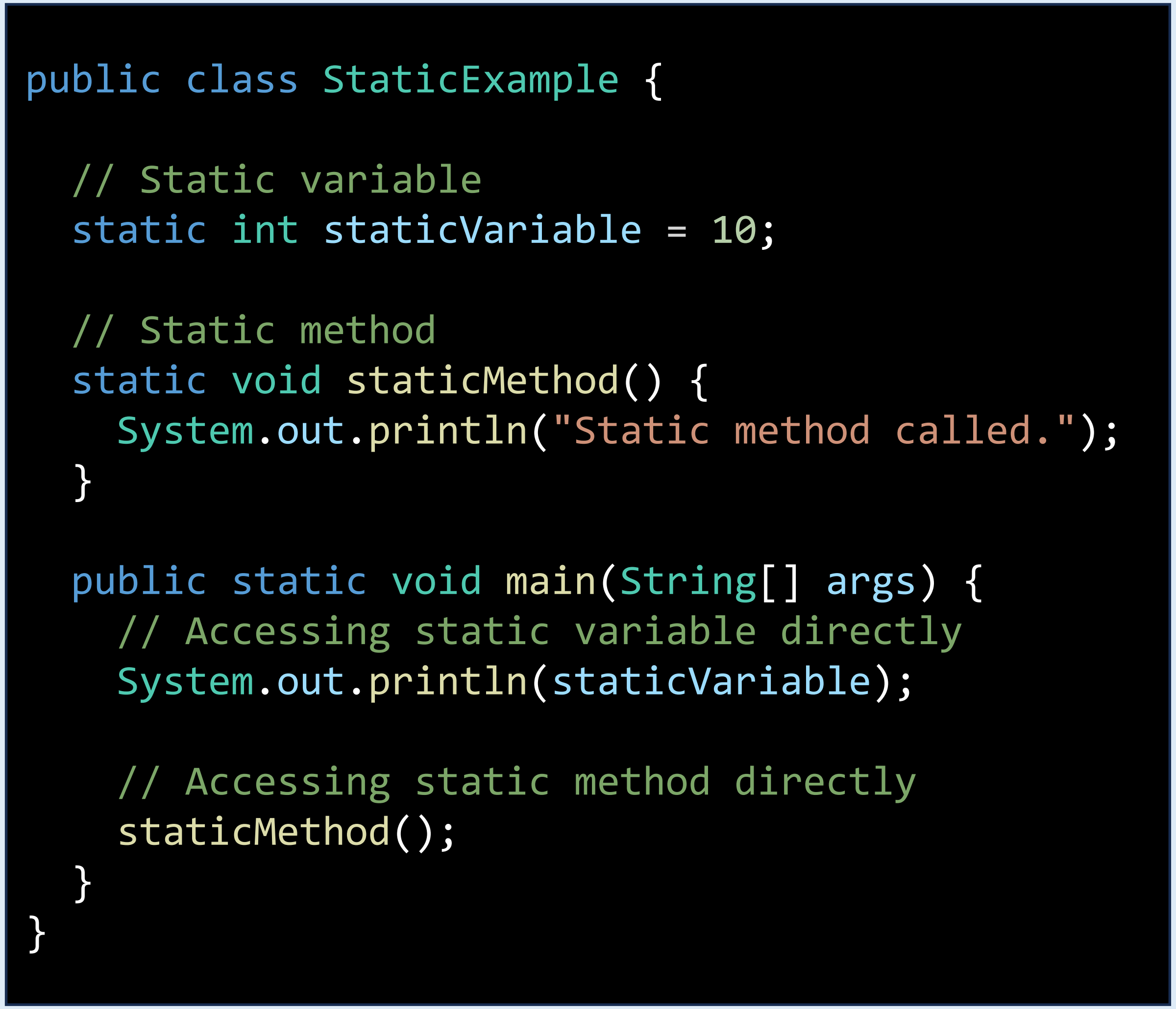
# Q. Can a static method be overridden in Java?

- No, a static method cannot be overridden in Java.

back to chapter index

# Q. How can static methods and variables be accessed within the same class?

- Static methods and variables can be accessed within the same class directly by their names.



```java
public class StaticExample {

  // Static variable
  static int staticVariable = 10;

  // Static method
  static void staticMethod() {
    System.out.println("Static method called.");
  }

  public static void main(String[] args) {
    // Accessing static variable directly
    System.out.println(staticVariable);

    // Accessing static method directly
    staticMethod();
  }
}
```

back to chapter index

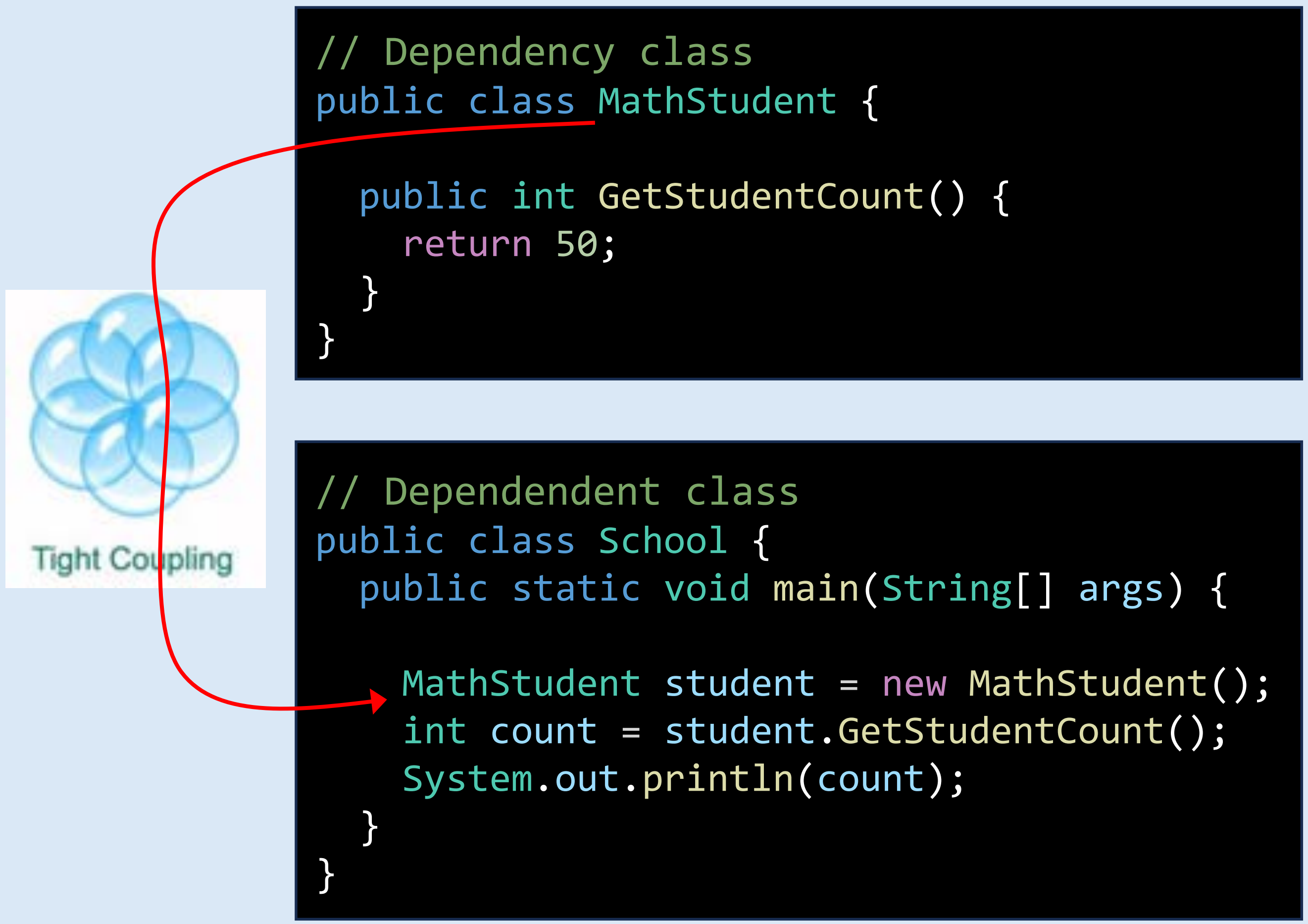
# Mock Interview - Spring Basics, IoC, DI, Components & Beans

Back to main index

# Q. Why the name is Inversion of Control (IoC)?

- Normally classes(School) directly creates and controls the lifecycle of objects (object creation), whereas in IoC the control of object creation and object lifecycle shifts or inverted to an external entity, such as an IoC container or framework.

School Management Application



```
// Dependency class
public class MathStudent {

  public int GetStudentCount() {
    return 50;
  }
}
```

```
// Dependendent class
public class School {
  public static void main(String[] args) {

    MathStudent student = new MathStudent();
    int count = student.GetStudentCount();
    System.out.println(count);
  }
}
```

Violation of IoC

back to chapter index

# Q. Why do developers prefer Spring for enterprise applications?

- Spring offers features like **dependency injection**, **loose coupling**, **simplified configuration**, **unit testing**, making it a robust choice for scalable enterprise applications.

back to chapter index

# Q. What principle does Spring follow to ensure components are loosely coupled?

- Spring follows the **IoC principle** and **Dependency Injection** (DI) design pattern to decouple component dependencies, which aligns with the IoC concept.

back to chapter index

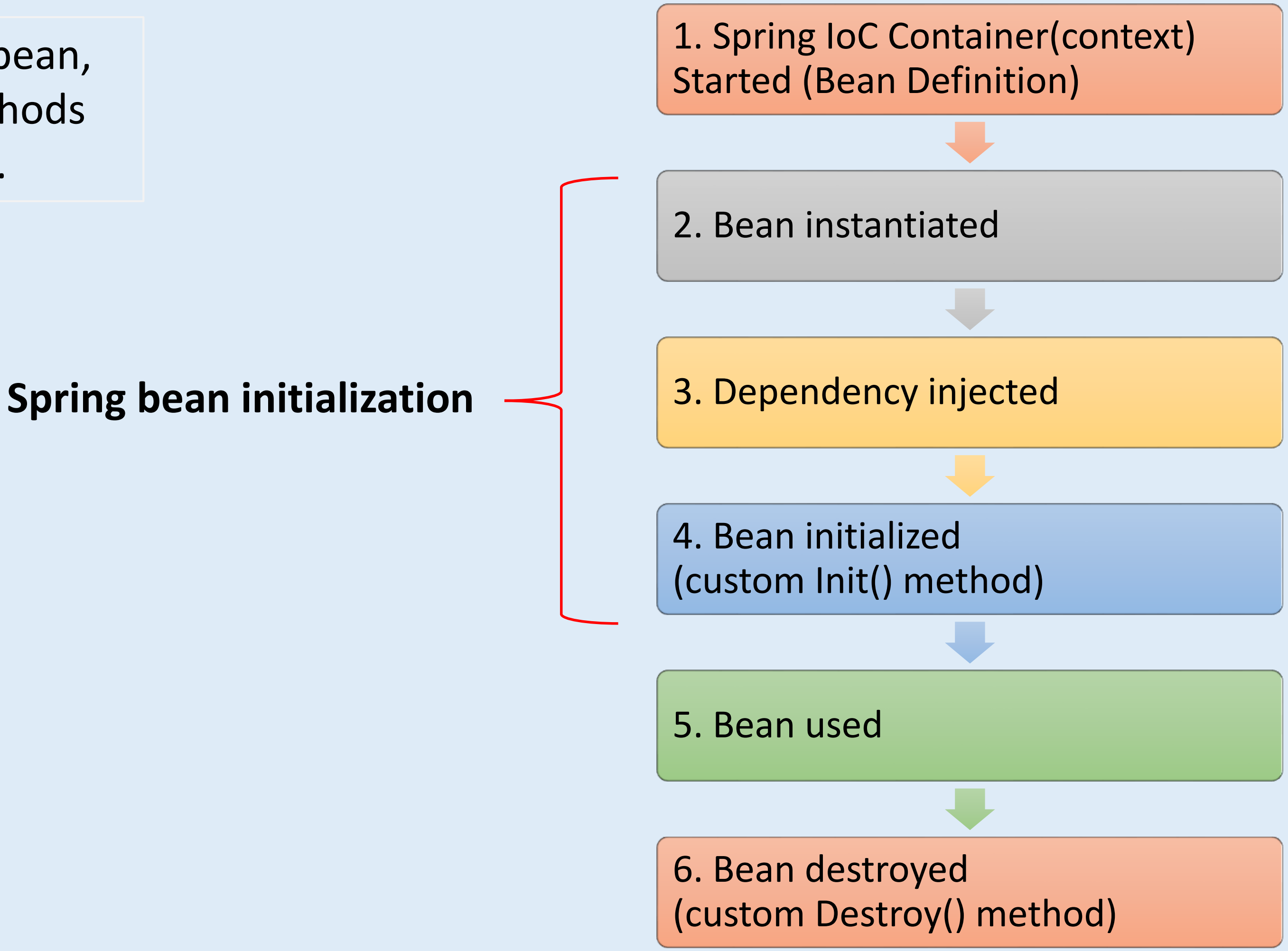
# Q. What role does a container play in Spring?

- The Spring IoC container **manages object creation**, configuration, and lifecycle, ensuring proper dependency injection and loose coupling.

back to chapter index

# Q. What happens during the initialization of a Spring bean?

- During initialization, Spring instantiates the bean, injects dependencies, and calls lifecycle methods like @PostConstruct or custom init methods.



1. Spring IoC Container(context) Started (Bean Definition)
2. Bean instantiated
3. Dependency injected
4. Bean initialized (custom Init() method)
5. Bean used
6. Bean destroyed (custom Destroy() method)
Spring bean initialization

back to chapter index

# Q. How does Spring create and manage beans?

- Spring manages beans through its **IoC container**, handling their lifecycle from instantiation to destruction, with support for annotations like @Bean and @Component.

back to chapter index

# Q. What is the role of Dependency Injection in testing?

- Dependency Injection facilitates **loose coupling** because of which independent classes can be easily tested by easily substituting the dependencies with mocks.

back to chapter index

# Mock Interview - Spring Configuration, Annotations, AOP, Scope of a Bean

Back to main index

# Q. How does Spring decide which beans to inject?

- Spring inject the **@Primary** annotated bean.

back to chapter index

# Q. How do you define a Spring-managed bean without annotations?

- Spring-managed beans can be defined in a configuration file **using XML**, specifying the bean’s class and properties.

back to chapter index

# Q. How does Spring determine which constructor to use when multiple are present?

- Spring uses the constructor with the **@Autowired** annotation.

back to chapter index

# Q. When would you choose ApplicationContext and when BeanFactory?

- Use ApplicationContext when you need **advanced features** like event propagation, declarative mechanisms for creating a Bean, and AOP integration.

- Use BeanFactory in **resource-constrained** environments where startup time is critical, as it **lazily loads Beans**. However, ApplicationContext is generally preferred for its additional features.

back to chapter index

# Q. How would you ensure cross-cutting concerns are modularized in a Spring application?

- By applying Aspect-Oriented Programming (AOP) with Spring to modularize concerns like logging, transaction management, and security.

back to chapter index

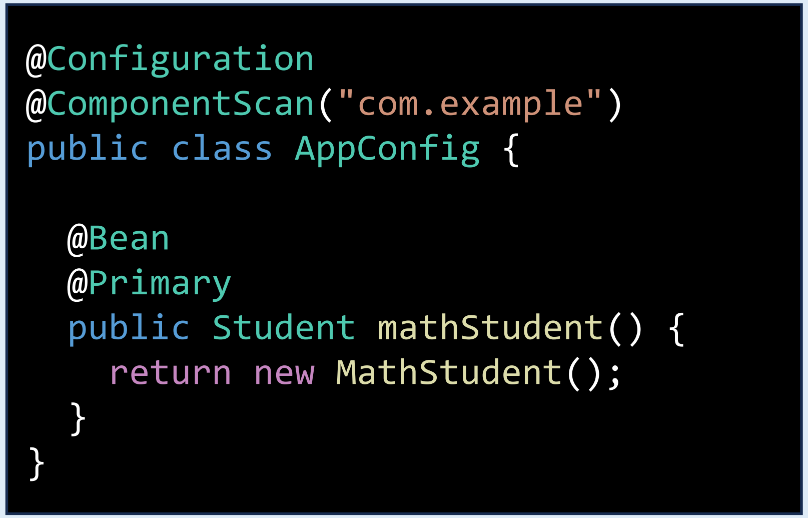
# Q. What you will do to resolve Bean circular dependencies in Spring?

- By implementing setter injection for resolving the circular dependency.

back to chapter index

# Q. How can you ensure that all the components in your Spring application are correctly processed?

- By using **@ComponentScan** to automatically detect and register Beans from your package structure or @Import to include other configurations.



```java
@Configuration
@ComponentScan("com.example")
public class AppConfig {

  @Bean
  @Primary
  public Student mathStudent() {
    return new MathStudent();
  }
}
```

back to chapter index

# Q. In what way can Spring AOP improve your application design?

- Spring AOP allows you to separate concerns and reducing code duplication.

back to chapter index

# Q. In a multi-threaded application, which bean scope you will implement to ensure thread safety?

- For thread safety, **use Prototype scope** to create a new instance of a Bean for each request, avoiding shared state across threads.

back to chapter index

# Q. What Bean scope you will prefer for handling HTTP requests independently?

- **Use Request scope**, where a new Bean instance is created for each HTTP request, ensuring that each user interaction is handled independently.

back to chapter index

# Q. How does Spring ensure that a Singleton Bean remains unique across the application?

- Spring creates and maintains a single instance of the Bean within the **application context**, ensuring that it's reused across the application.

back to chapter index

# Mock Interview - Spring Boot

Back to main index

# Q. How would you setup of a new Spring project?

- Use **Spring Initializr**, a web-based tool that generates a Spring Boot project structure with dependencies, allowing for quick setup.

back to chapter index

# Q. What advantage does Spring Boot offer for microservices architecture? 💡

- Spring Boot simplifies microservices by providing embedded servers, auto-configuration, and production-ready features, reducing the need for extensive XML configuration.

back to chapter index

# Q. How does the @SpringBootApplication annotation simplify Spring Boot application setup?

- The @SpringBootApplication annotation consolidates three crucial annotations: **@Configuration, @EnableAutoConfiguration, and @ComponentScan**, simplifying the configuration.

back to chapter index

# Q. When might you choose application.yml over application.properties for configuration?

- Use application.yml for hierarchical data structures and improved readability, especially when dealing with **complex configurations**, while application.properties is more suited for simpler key-value pairs.

back to chapter index

# Q. How do you manage multiple environments in a Spring Boot application?

- By using profiles with **application-{profile}.properties** or application-{profile}.yml files, allowing for environment-specific configurations that can be switched easily.

back to chapter index

# Q. What tools would you use to monitor and manage a Spring Boot application?

- **Spring Boot Actuator** provides various endpoints to monitor application health, metrics, and other management features, making it easier to maintain the application.

back to chapter index

# Q. How does Spring Boot simplify deployment in cloud environments?

- Spring Boot’s **embedded server** and self-contained packaging allow the application to be deployed as a simple executable, making it ideal for containerized environments like Docker.

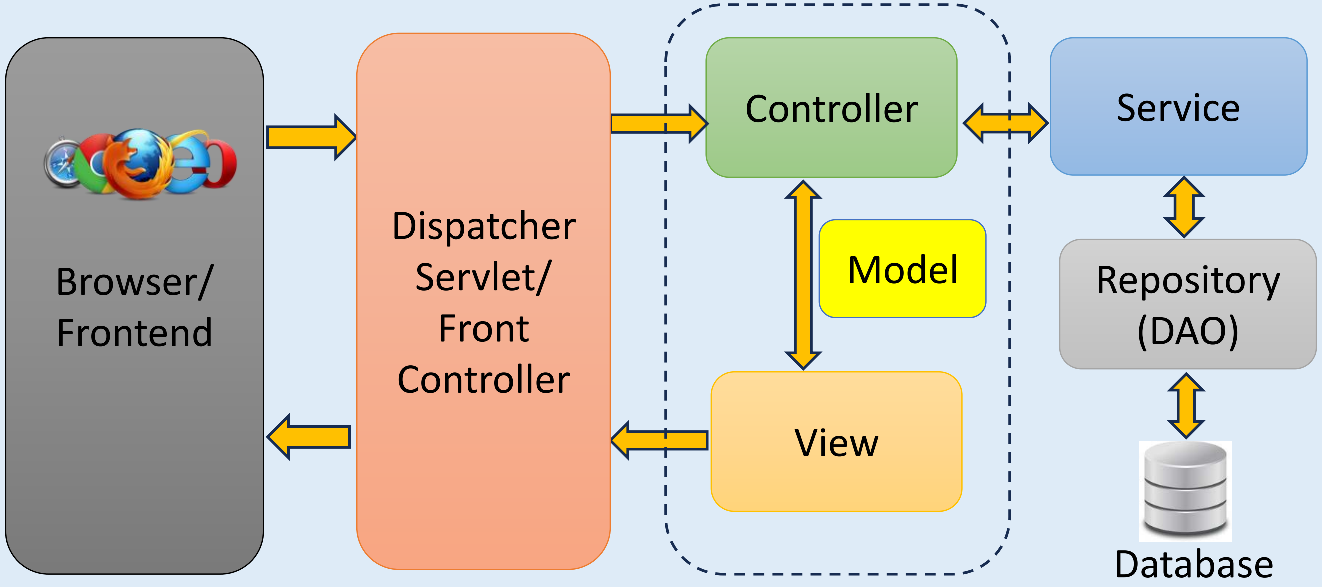
back to chapter index

# Mock Interview - Spring MVC

Back to main index

# Q. How do you pass data from the controller to the view in Spring MVC?

- In Spring MVC, data is passed from the controller to the view using the **Model**.



back to chapter index

# Q. What mechanism does Spring MVC use to route requests to the appropriate handler?

- The **DispatcherServlet** routes incoming HTTP requests to the appropriate controller based on the URL patterns defined with annotations like @RequestMapping.

back to chapter index

# Q. How would you map a specific URL to a method in a Spring MVC controller?

- Use the **@RequestMapping annotation** to map a specific URL pattern to a method within a controller, allowing the method to handle requests for that URL.

back to chapter index

# Q. How do you handle HTTP GET, PUT, POST, DELETE requests in Spring MVC?

- Use the @GetMapping, @PUTMapping, @POSTMapping and @DELETEMapping annotations over controller methods to handle HTTP GET, PUT, POST, DELETE requests in Spring MVC.

back to chapter index

# Q. When would you use @PostMapping versus @PutMapping in a Spring MVC application?

- @PostMapping is used for creating new resources, while @PutMapping is for updating existing resources.

back to chapter index

# Q. How does Spring MVC manage the data in an application?

- The Model in Spring MVC represents the data or the state of the application.

back to chapter index

# Q. How do you capture specific request parameters in a Spring MVC controller?

- Use @RequestParam to capture query parameters from the request URL, and @PathVariable to capture values from the URL path itself.

back to chapter index

# Q. In what scenarios would you use @ModelAttribute in Spring MVC?

- Use @ModelAttribute to populate a model before any request handler method is executed or to bind form data to a model object.

back to chapter index

# Q. What approach would you take to ensure form data is valid in a Spring MVC application?

- Use the @Valid annotation along with BindingResult to validate the data in the model object and handle validation errors within the controller.

back to chapter index

cognizant

# Thank You

- Moment Platform DXP- Sreeni