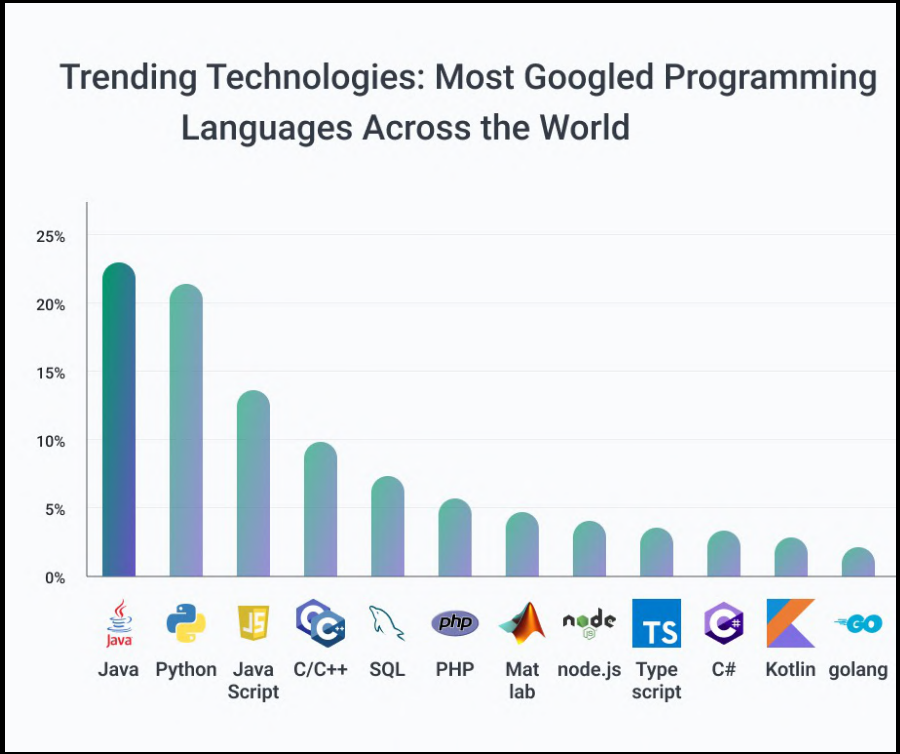
1. Introduction to Java

1. Why you must learn Java
2. What is a Programming Language
3. What is an Algorithm
4. What is Syntax
5. History of Java
6. Magic of Byte Code
7. How Java Changed the Internet
8. Java Buzzwords
9. Object Oriented Programming

1.1 Why you must learn Java



1. One of the most popular language. Java currently runs on 60,00,00,00,000 devices.
2. Wide Usage (Web-apps, backend, Mobile apps, enterprise software).
3. High paying and a lot of Jobs
4. Object Oriented
5. Rich APIs and Community Support

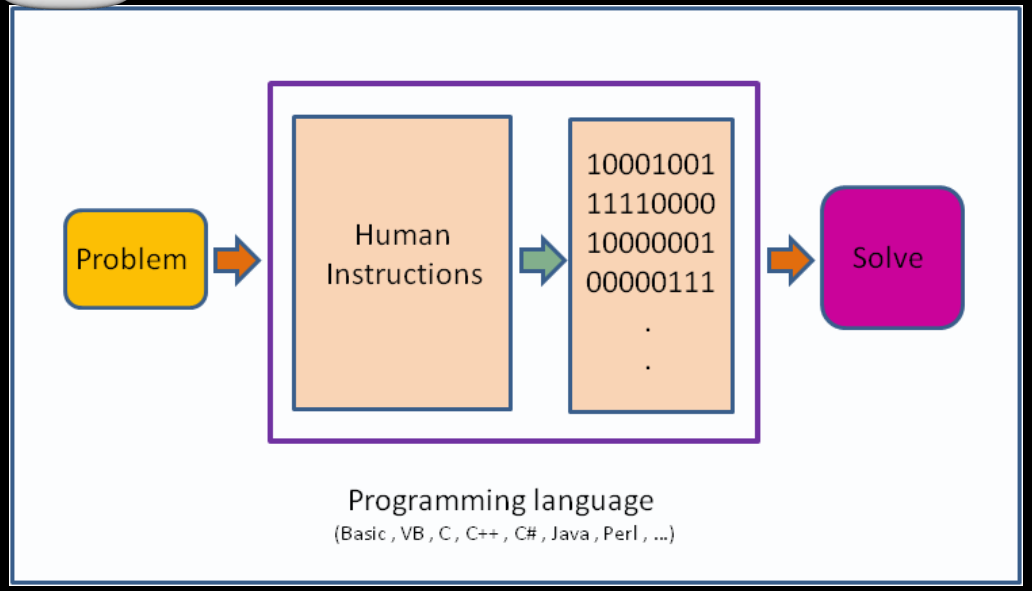
1.2 What is a Programming Language

Humans use natural language (like Hindi/English) to communicate

1.2 What is a Programming Language

Computers only understand 0/1 or on/off

1.2 What is a Programming Language



- Giving instructions to a computer
- **Instructions**: Tells computer what to do.
- These instructions are called code.
- Human instructions are given in High level languages.

Compiler converts high level languages to low level languages or machine code.

1.3 What is an Algorithm

1.3 What is an Algorithm

An algorithm is a step-by-step procedure for solving a problem or performing a task.

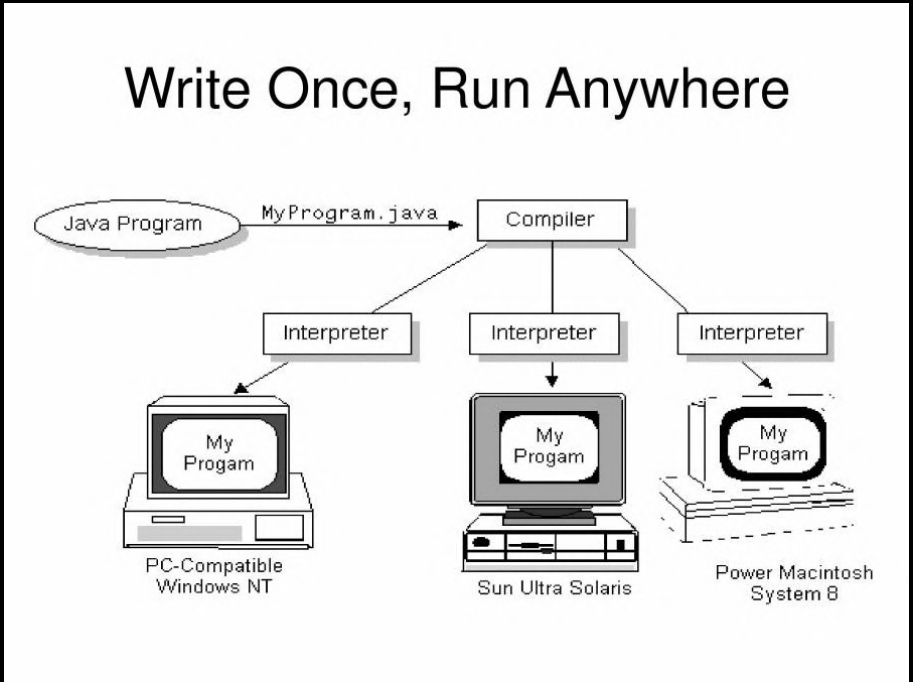
1.4 What is Syntax

- Structure of words in a sentence.
- Rules of the language.
- For programming exact syntax must be followed.

1.5 History of Java

Developed by James Gosling at Sun Microsystems (Early 1990s):
Originally named 'Oak', later renamed Java in 1995.

1.5 History of Java

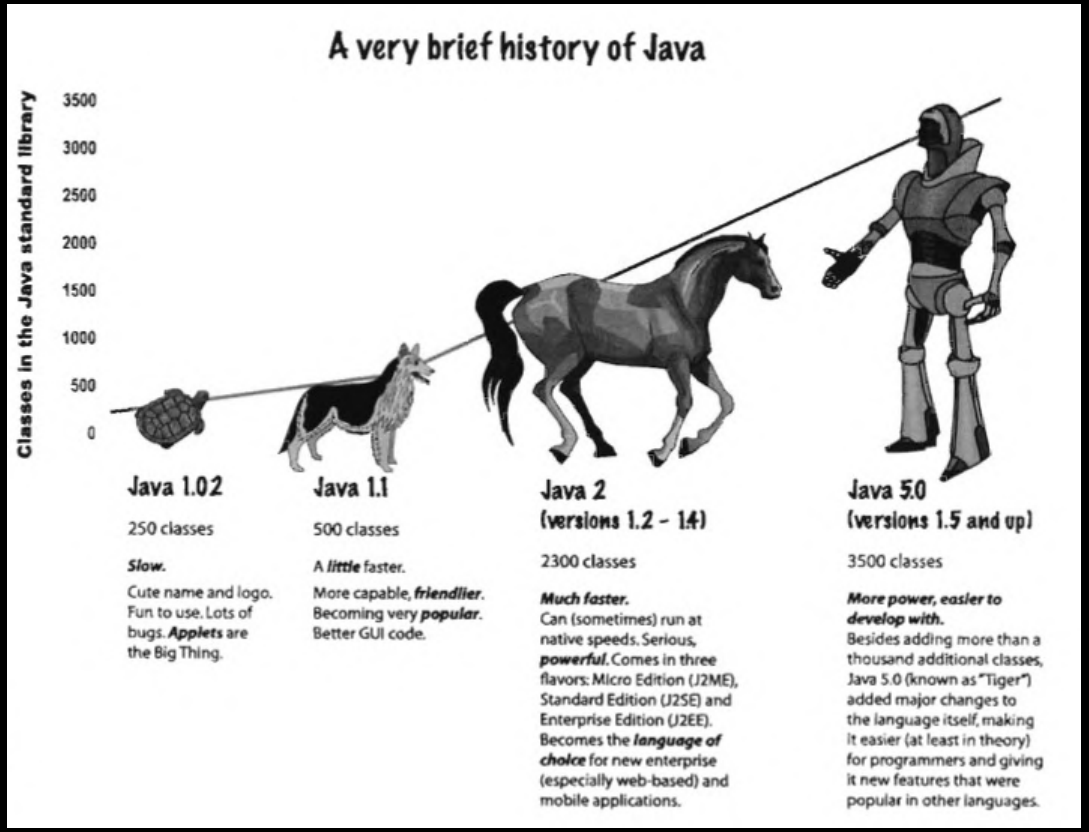


First Release (1995): Introduced "Write Once, Run Anywhere" concept with cross-platform compatibility.

1.5 History of Java

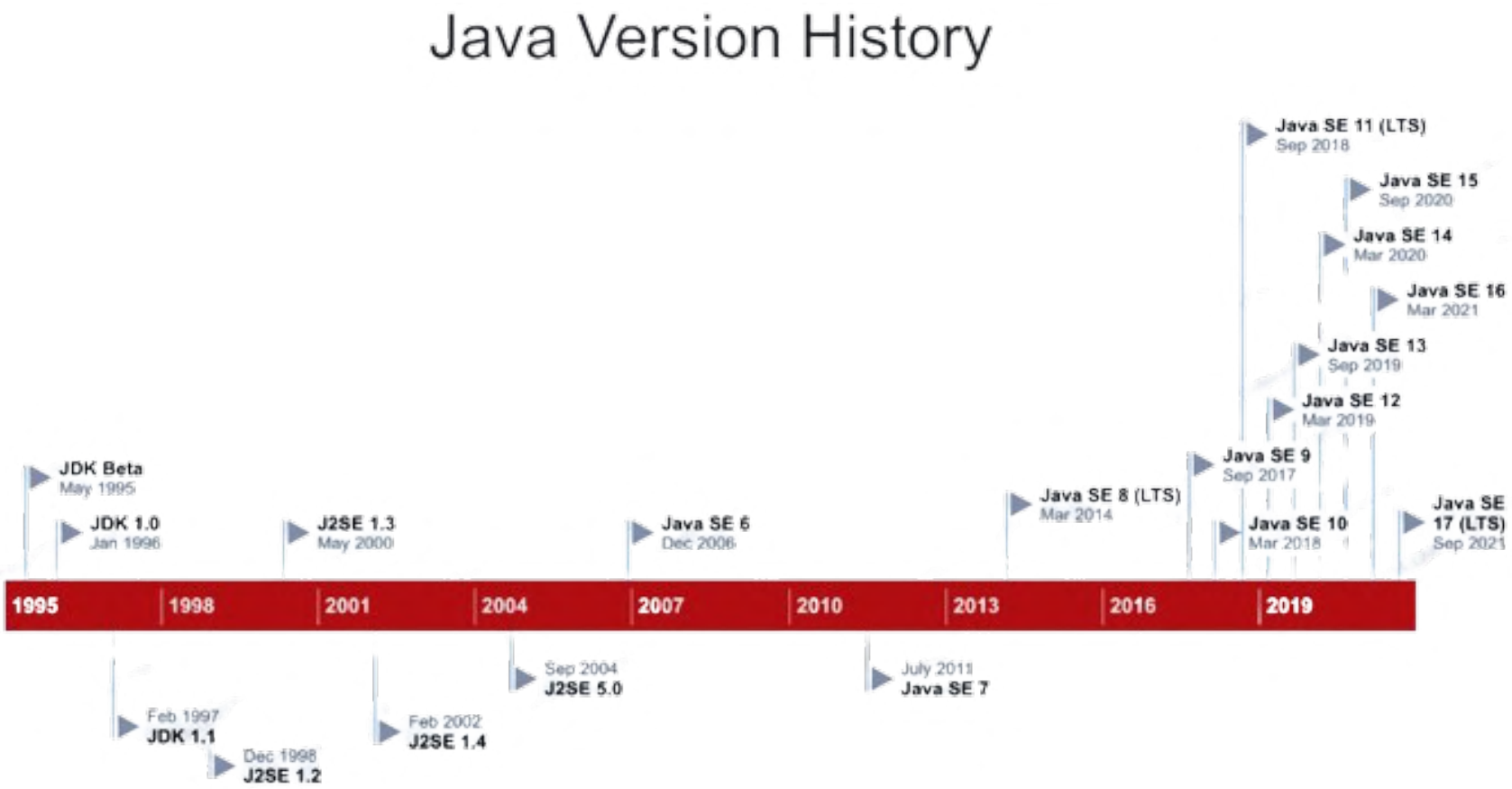
Developed with vision of backward compatibility. Should not break with new version release.

1.5 History of Java

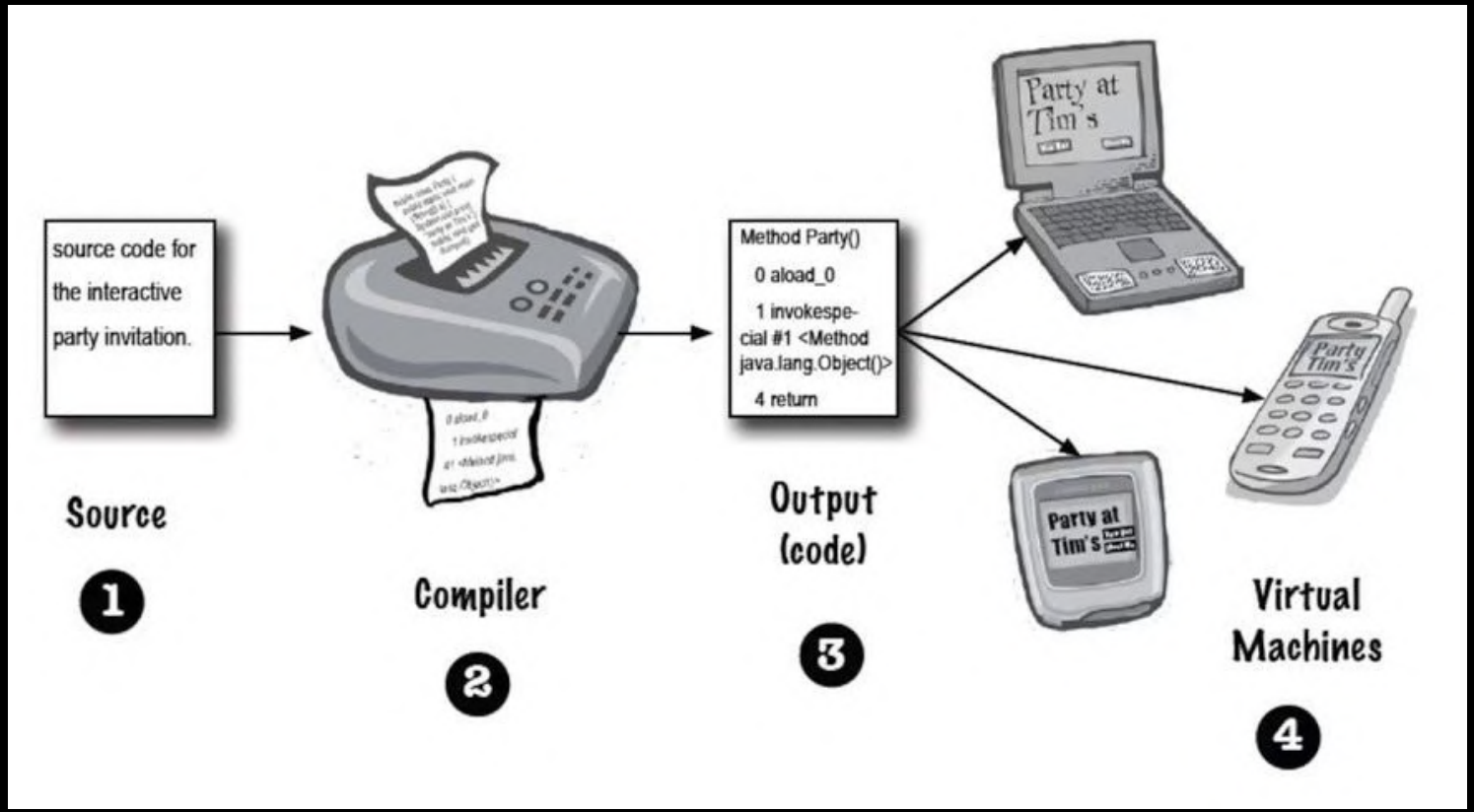


Rapid Growth and Diversification (Late 1990s - 2010s): Expanded from web applets to server-side applications; standardized into different editions for various computing platforms.

1.5 History of Java



1.6 Magic of Byte Code



1.7 How Java Changed the Internet

Portability with Write Once Run Anywhere

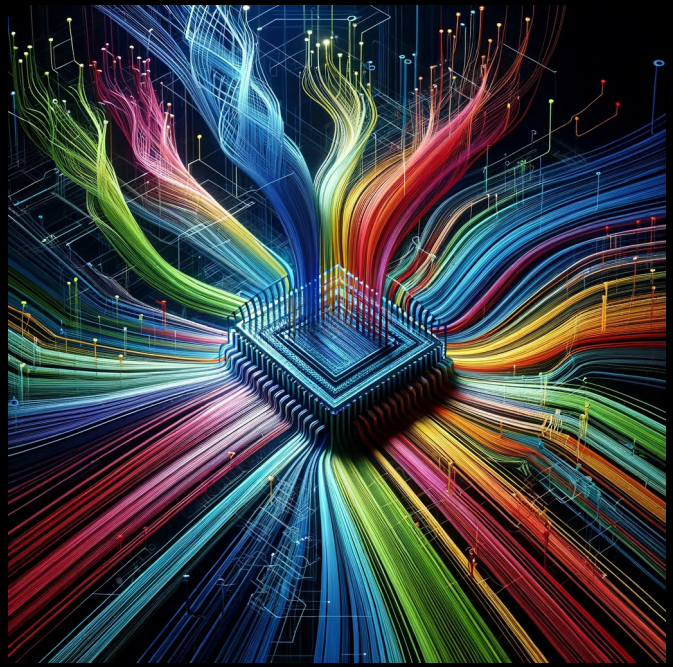
Security because of Code running on Virtual Machine

1.8 Java Buzzwords

Robust

Java is robust due to its strong memory management, exception handling, and type-checking mechanisms, which help in preventing system crashes and ensuring reliable performance.

1.8 Java Buzzwords



Multithreaded

Multithreading in programming is the ability of a CPU to execute multiple threads concurrently, allowing for more efficient processing and task management.

1.8 Java Buzzwords

Architecture Neutral

Java is architecturally neutral because its compiled code (bytecode) can run on any device with a Java Virtual Machine (JVM), regardless of the underlying hardware architecture.

1.8 Java Buzzwords

Interpreted and High Performance

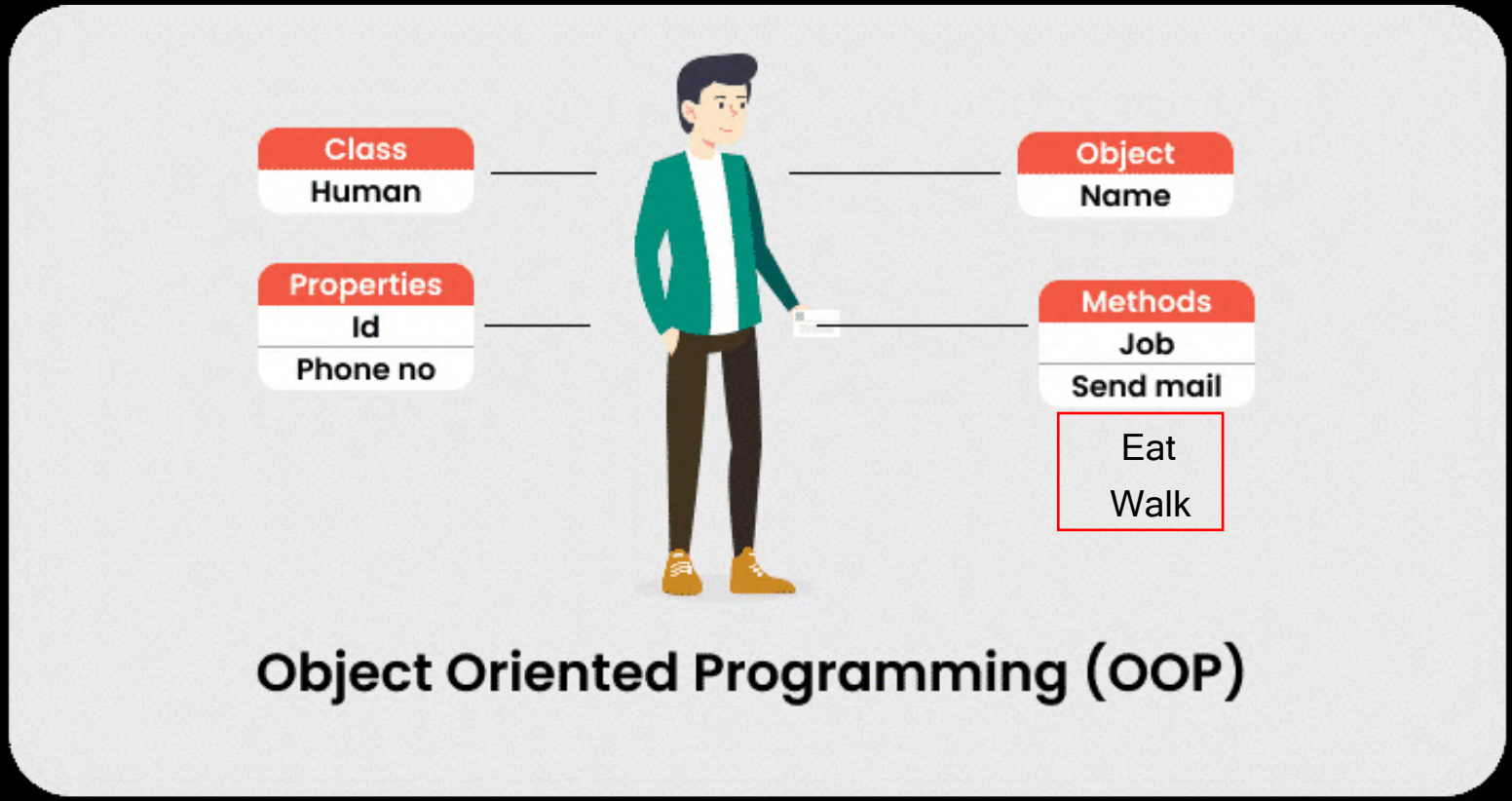
Java combines high performance with interpretability, as its bytecode is interpreted by the Java Virtual Machine (JVM), which employs Just-In-Time (JIT) compilation for efficient and fast execution.

1.8 Java Buzzwords

Distributed

Java is inherently distributed, designed to facilitate network-based application development and interaction, seamlessly integrating with Internet protocols and remote method invocation.

1.9 Object Oriented Programming



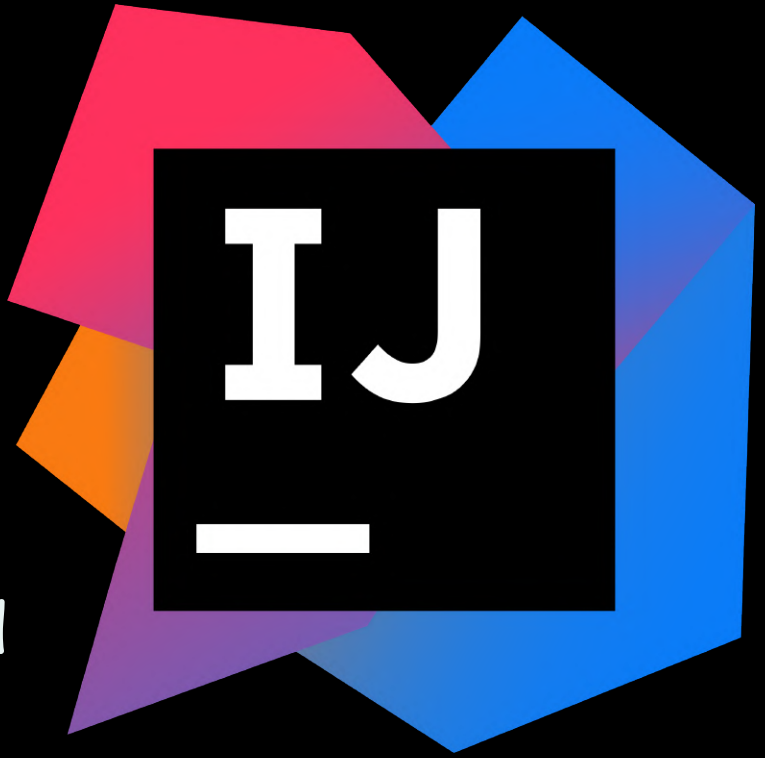
1.9 Object Oriented Programming

Revision

1. Why you must learn Java
2. What is a Programming Language
3. What is an Algorithm
4. What is Syntax
5. History of Java
6. Magic of Byte Code
7. How Java Changed the Internet
8. Java Buzzwords
9. Object Oriented Programming

2. Java Basics

1. Installing JDK
2. First Class using Text Editor
3. Compiling and Running
4. Anatomy of a Class
5. File Extensions
6. JDK vs JVM vs JRE
7. Showing Output
8. Importance of the main method
9. Installing IDE(Intellij Idea)
10. Creating first Project



2.1 Installing JDK

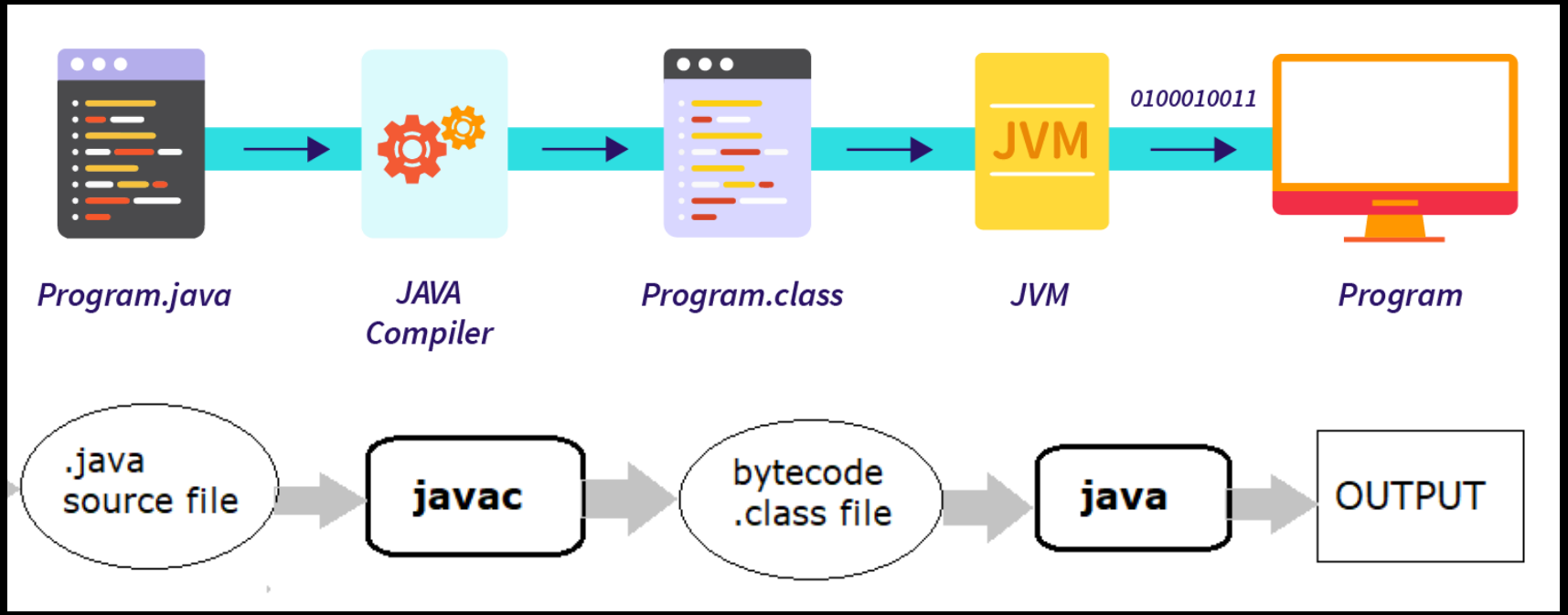
- Search JDK Download
- Make sure to download from the oracle website.
- Download the latest version

2.2 First Class using Text Editor

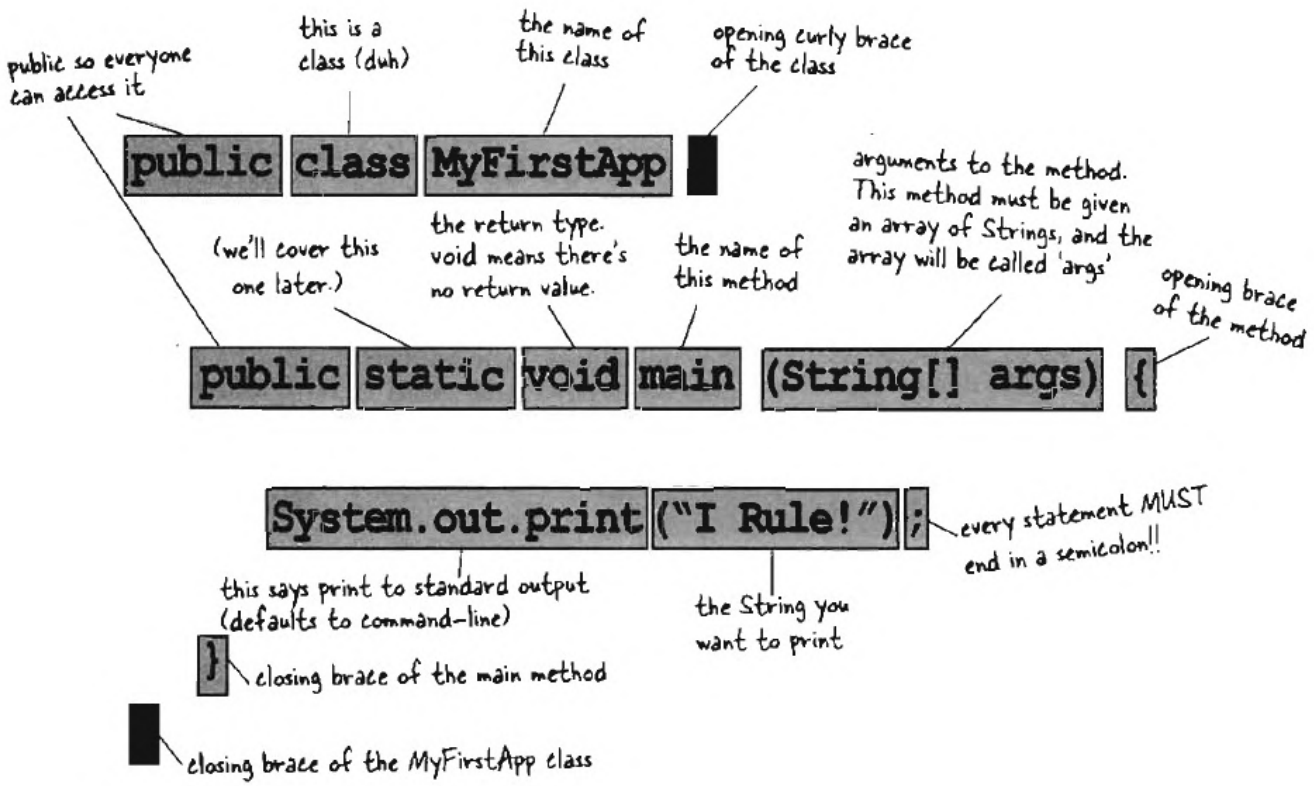
First.java

```
import java.lang.*;

public class First {
  public static void main(String[] args) {
    System.out.println("Welcome to KGCoding");
  }
}
```

2.3 Compiling and Running



2.4 Anatomy of a Class



2.5 File Extensions

.Java

- Contains Java Source Code
- High Level Human Readable
- Used for Development
- File is editable

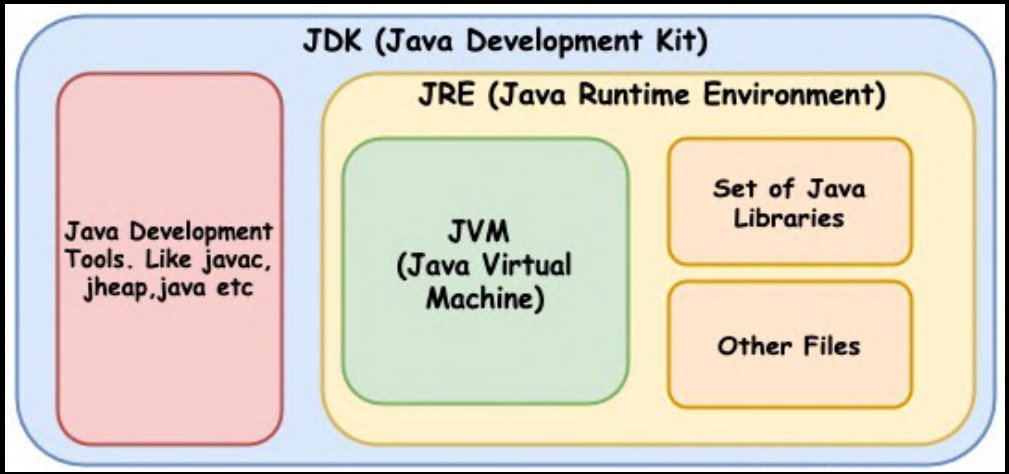
```
Main.java
1  public class Main {
2      public static void main(String[] args) {
3          System.out.println("Hello world!");
4      }
5  }
```

.Class

- Contains Java Bytecode
- For consumption of JVM
- Used for Execution
- Not meant to be edited

```
Main.class
.∫æ

java/lang/Object<init>()V


java/lang/SystemoutLjava/io/PrintStream;
Hello world!

java/io/PrintStreamprintln(Ljava/lang/
String;)VMainCodeLineNumberTableLocalVariableTablethisLMain;ma
in([Ljava/lang/String;)Vargs[Ljava/lang/String;
SourceFile      Main.java!/*∑±
        7       ≤
∂

        !
```

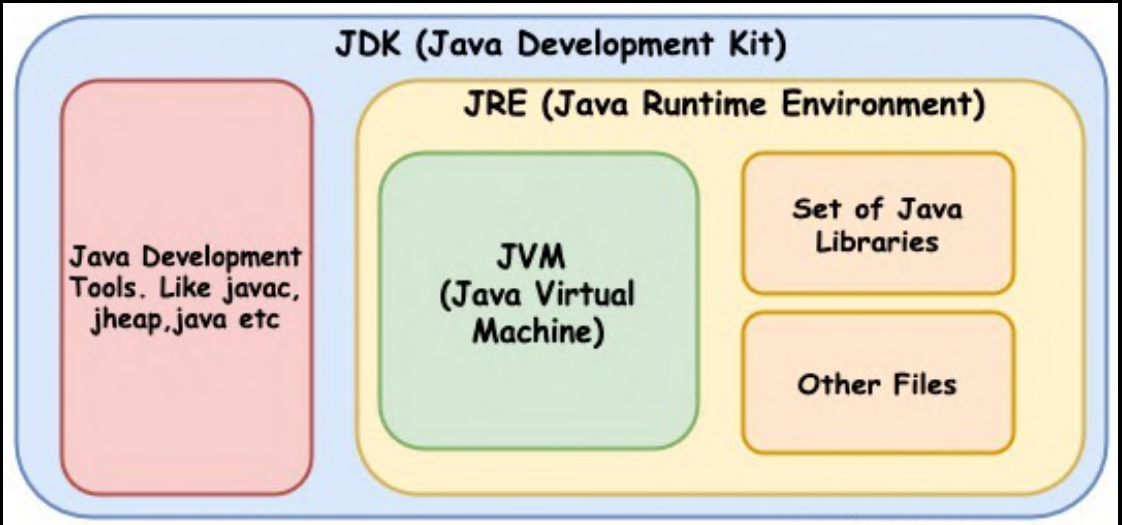
2.6 JDK vs JVM vs JRE



.JDK

- It's a software development kit required to develop Java applications.
- Includes the JRE, an interpreter/loader (Java), a compiler (javac), a doc generator (Javadoc), and other tools needed for Java development.
- Essentially, JDK is a superset of JRE.

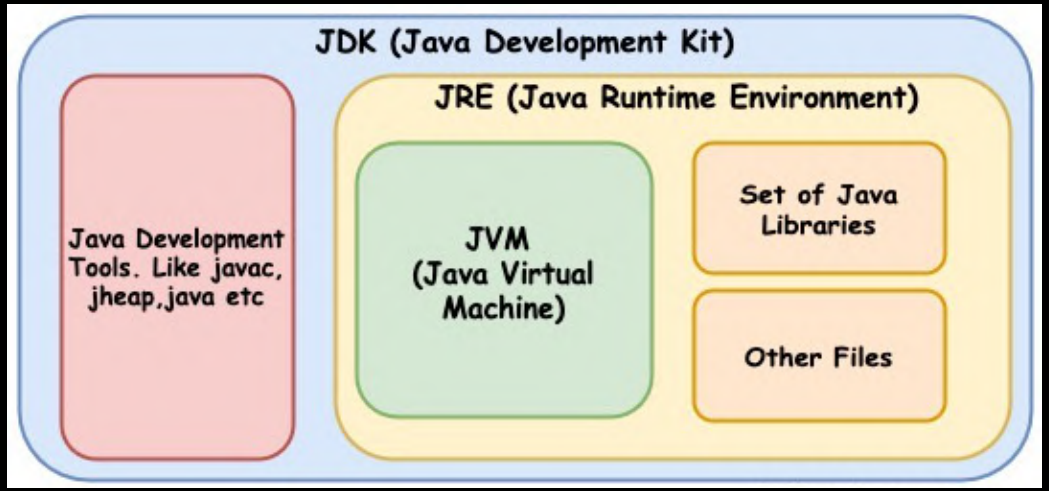
2.6 JDK vs JVM vs JRE



.JRE

- It's a part of the JDK but can be downloaded separately.
- Provides the libraries, the JVM, and other components to run applications
- Does not have tools and utilities for developers like compilers or debuggers.

2.6 JDK vs JVM vs JRE



.JVM

- It's a part of JRE and responsible for executing the bytecode.
- Ensures Java's write-once-run-anywhere capability.
- Not platform-independent: a different JVM is needed for each type of OS.

2.7 Showing Output

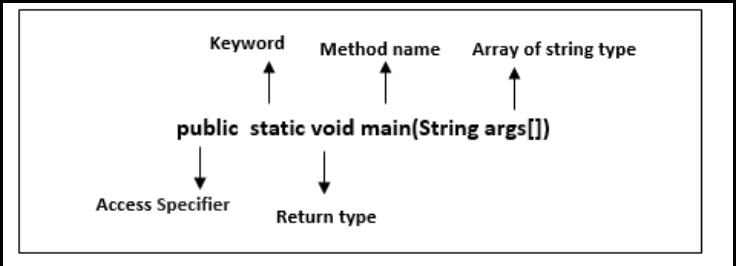
Example	Result

`System.out.print("one");`
`System.out.print("two");`
`System.out.println("three");`
	onetwothree

`System.out.println("one");`
`System.out.println("two");`
`System.out.println("three");`
	one
	two
	three

`System.out.println();`	[new line]

2.8 Importance of the main method



- Entry Point: It's the entry point of a Java program, where the execution starts. Without the main method, the Java Virtual Machine (JVM) does not know where to begin running the code.
- Public and Static: The main method must be public and static, ensuring it's accessible to the JVM without needing to instantiate the class.
- Fixed Signature: The main method has a fixed signature: public static void main(String[] args). Deviating from this signature means the JVM won't recognize it as the starting point.

2.9 What is IDE

1. IDE stands for Integrated Development Environment.
2. Software suite that consolidates basic tools required for software development.
3. Central hub for coding, finding problems, and testing.
4. Designed to improve developer efficiency.

2.9 Need of IDE

1. Streamlines development.
2. Increases productivity.
3. Simplifies complex tasks.
4. Offers a unified workspace.
5. IDE Features
    1. Code Autocomplete
    2. Syntax Highlighting
    3. Version Control
    4. Error Checking

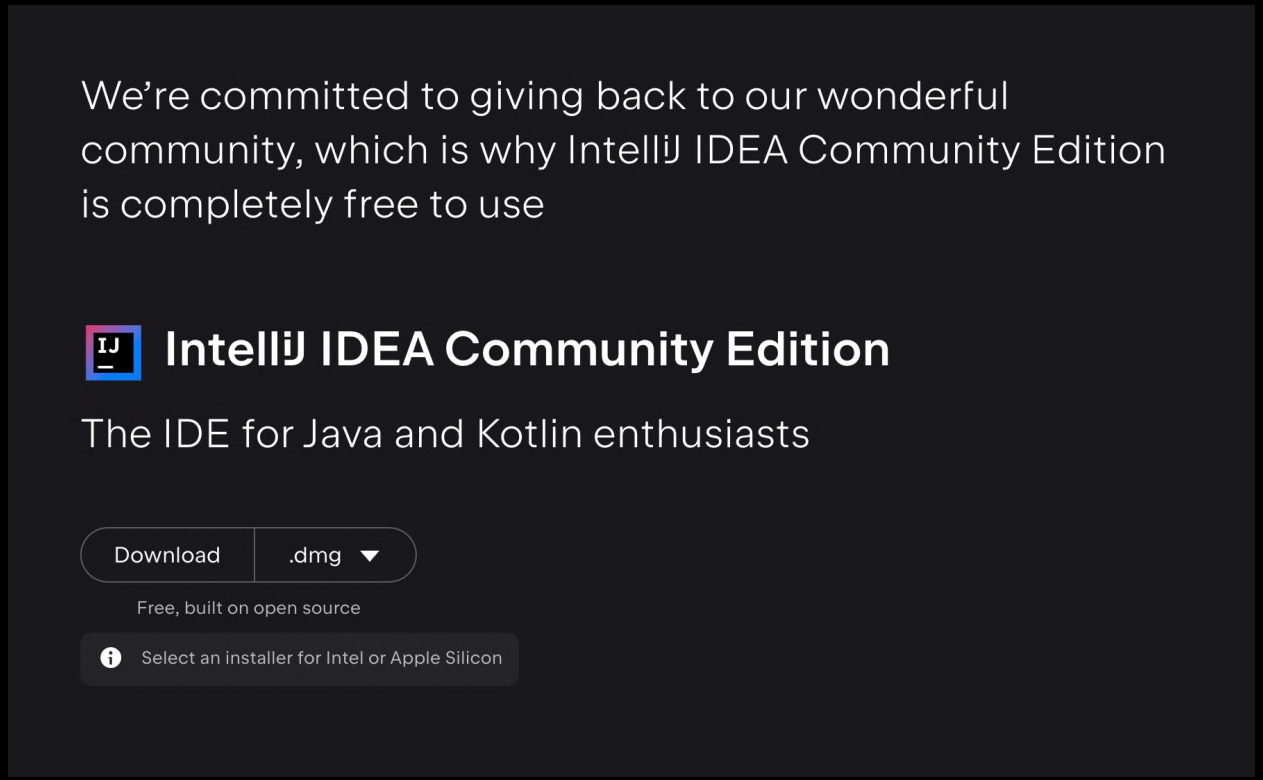
MainActivity.kt

```
@Composable
fun MessageCard(msg: Message) {
    Row(modifier = Modifier.padding(all = 8.dp)) {
        Image(
            painter = painterResource(R.drawable.android_studio_logo),
            contentDescription = "Profile Picture",
            modifier = Modifier
                .size(45.dp)

        )


        Spacer(modifier = Modifier.width(8.dp))
        Column (Modifier
            .background(color = Color.White)) {
            Text(text = msg.author, color = Color.Black)
            Spacer(modifier = Modifier.height(1.dp))
            Text(text = msg.body, color = Color.Black)
        }
    }
}
```

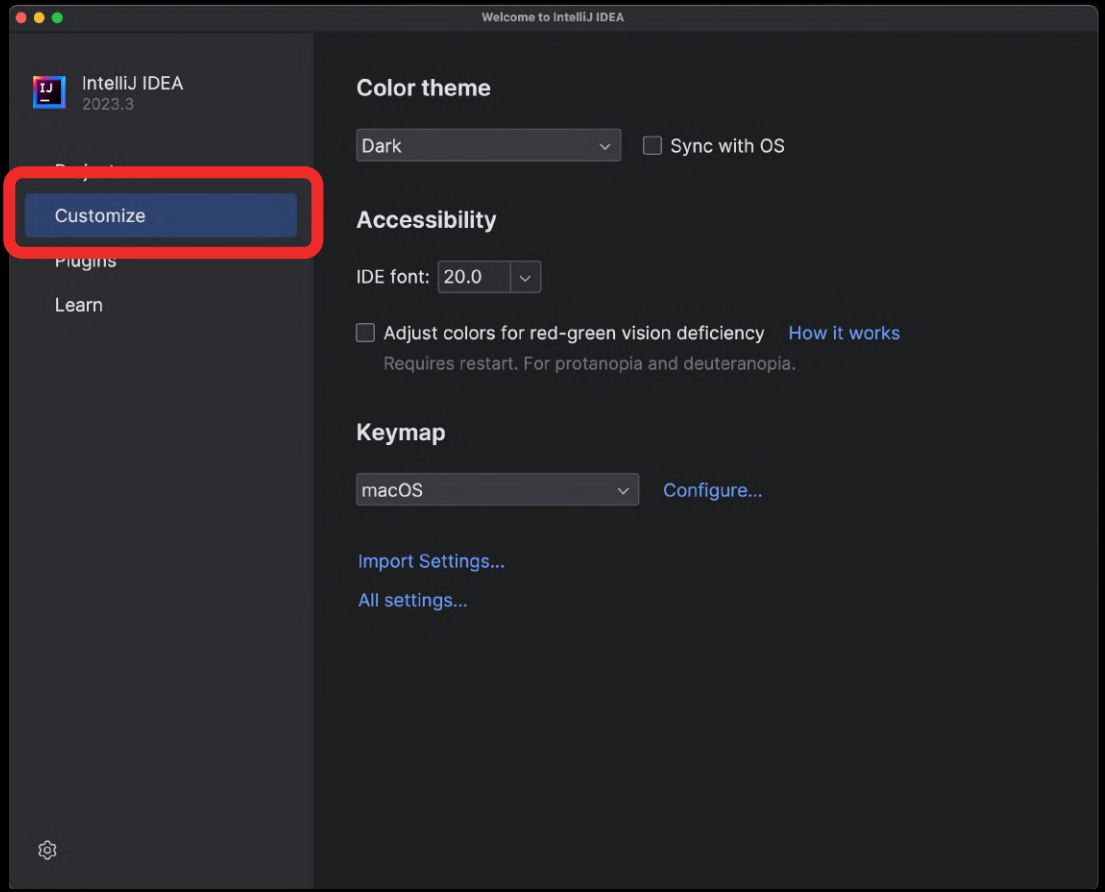
2.9 Installing IDE(Intellij Idea)

- Search Intellij IDEA
- Make sure to download the community Version.
- Keep Your Software up to date.

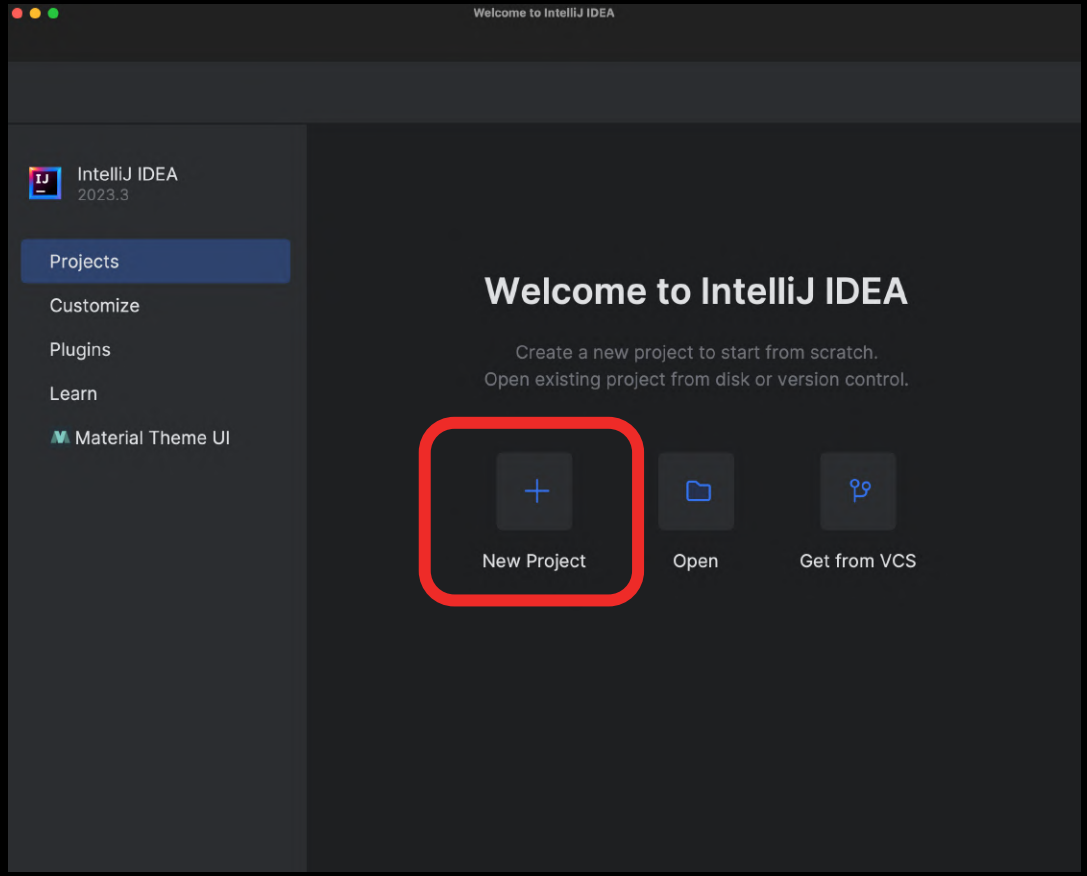
2.9 Installing IDE(Intellij Idea)



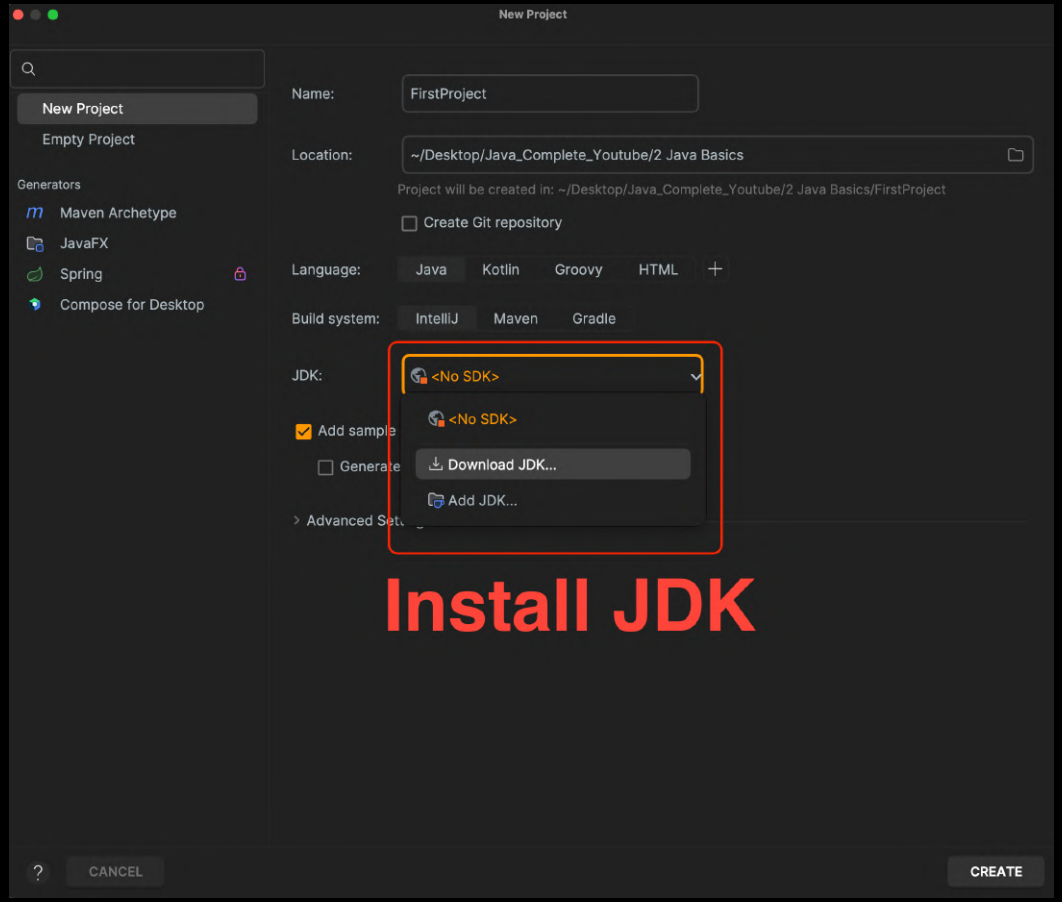
2.9 Installing IDE(Intellij Idea)



2.9 Installing IDE(Intellij Idea)

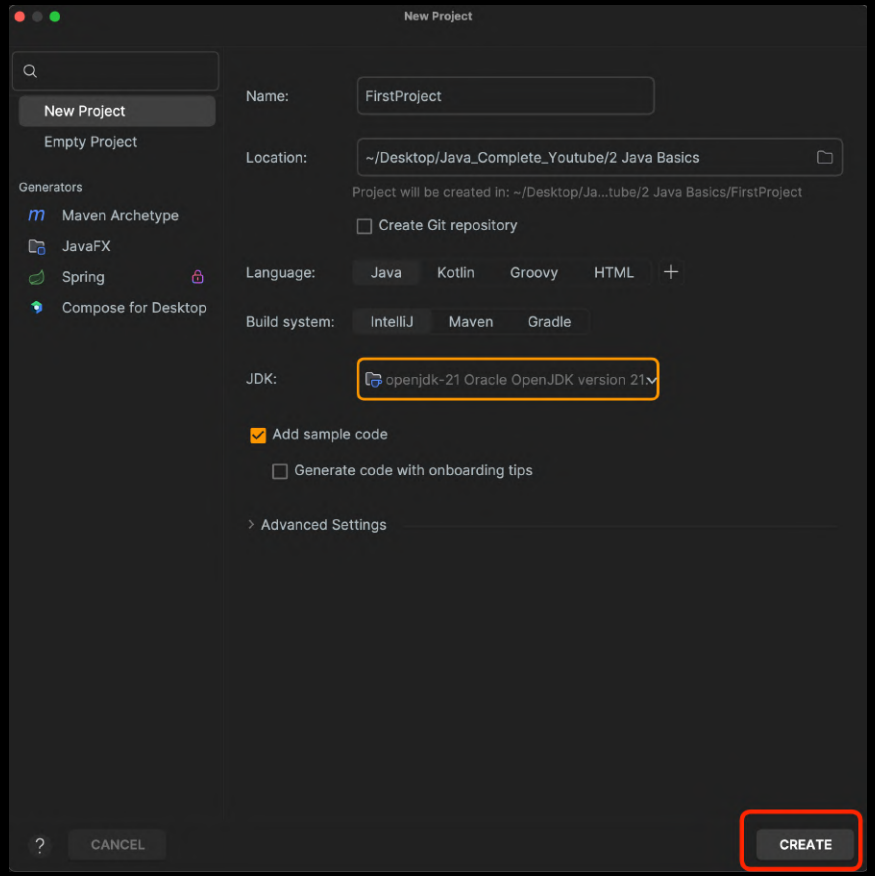


2.9 Installing IDE(Intellij Idea)



Install JDK

2.10 Creating first Project

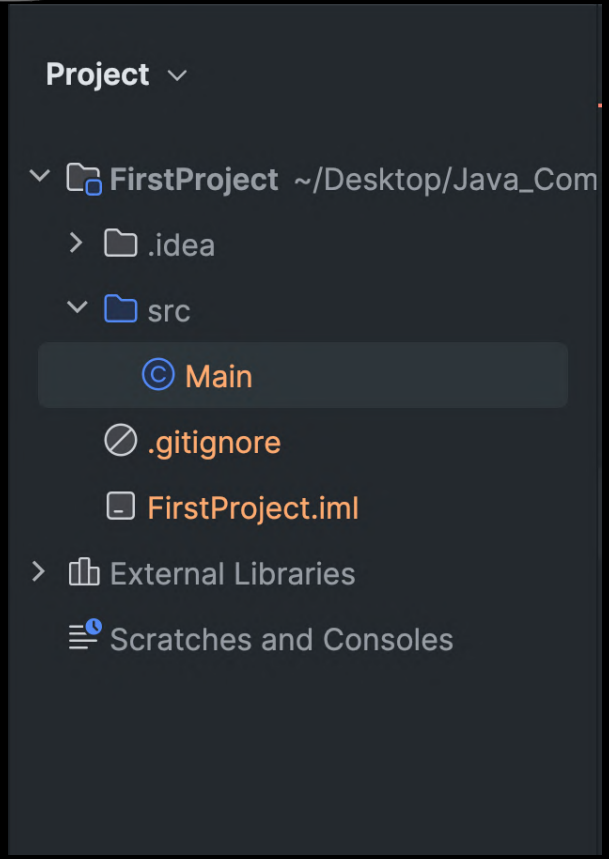


2.10 First Java Class

Main.java

```java
public class Main {
    public static void main(String[] args) {
        System.out.println("Hello world!");
    }
}
```

2.10 Project Structure



Revision

1. Installing JDK
2. First Class using Text Editor
3. Compiling and Running
4. Anatomy of a Class
5. File Extensions
6. JDK vs JVM vs JRE
7. Showing Output
8. Importance of the main method
9. Installing IDE(Intellij Idea)
10. Creating first Project

Tasks:

1. Create a class to output “good morning” using a text editor and check output.
2. Create a new Project in Intelij Idea and output “subscribe” on the console.
3. Show the following patterns:

Practice Exercise

Java Basics

Answer in True/False:

1. Computers understand high level languages like Java, C.
2. An Algorithm is a set of instructions to accomplish a task.
3. Computer is smart enough to ignore incorrect syntax.
4. Java was first released in 1992.
5. Java was named over a person who made good coffee.
6. ByteCode is platform independent.
7. JDK is a part of JRE.
8. It's optional to declare main method as public.
9. .class file contains machine code.
10. println adds a new line at the end of the line.

Practice Exercise

Java Basics

Answer in True/False:

1. Computers understand high level languages like Java, C. — False
2. An Algorithm is a set of instructions to accomplish a task. — True
3. Computer is smart enough to ignore incorrect syntax. — False
4. Java was first released in 1992. — False
5. Java was named over a person who made good coffee. — False
6. ByteCode is platform independent. — True
7. JDK is a part of JRE. — False
8. It's optional to declare main method as public. — False
9. .class file contains machine code. — False
10. println adds a new line at the end of the line. — True

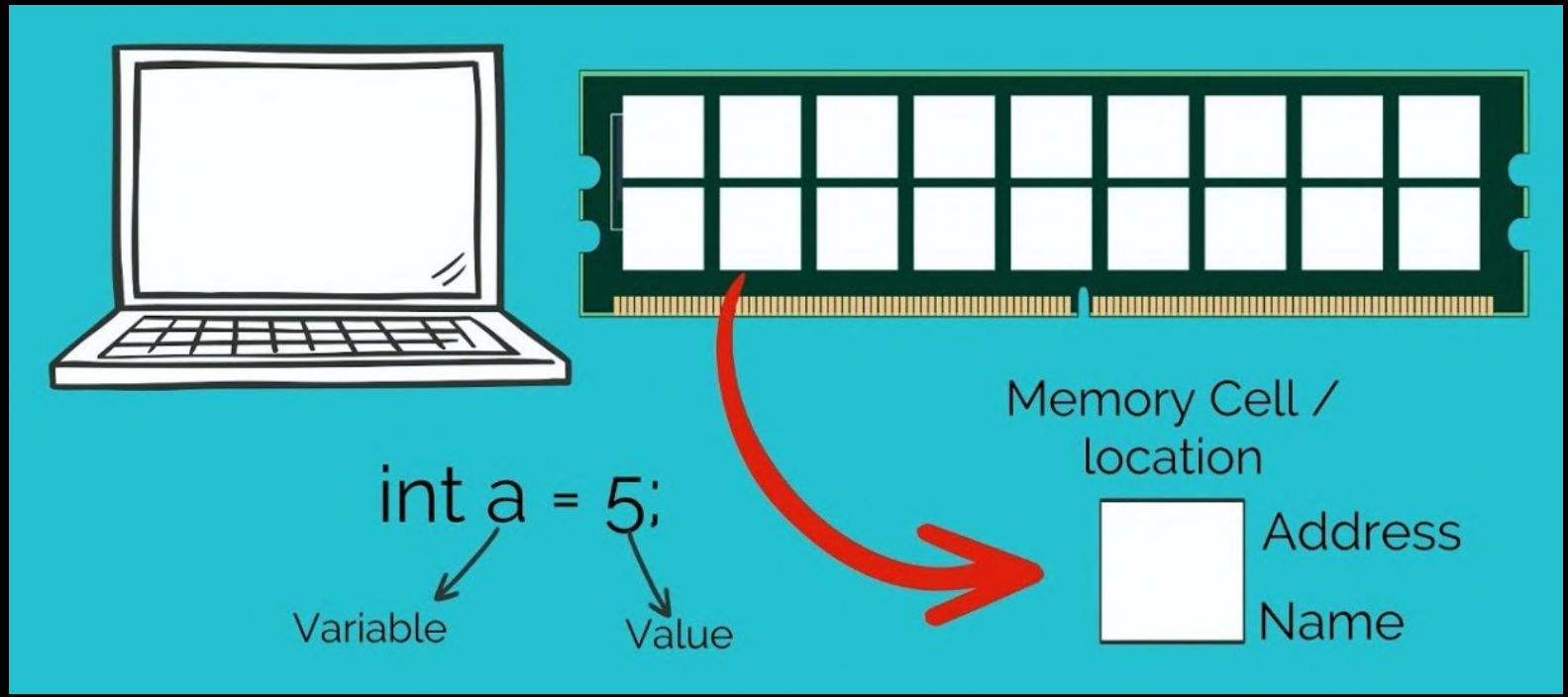
3 Data Types, Variables & Input

1. Variables
2. Data Types
3. Naming Conventions
4. Literals
5. Keywords
6. Escape Sequences
7. User Input
8. Type Conversion and Casting

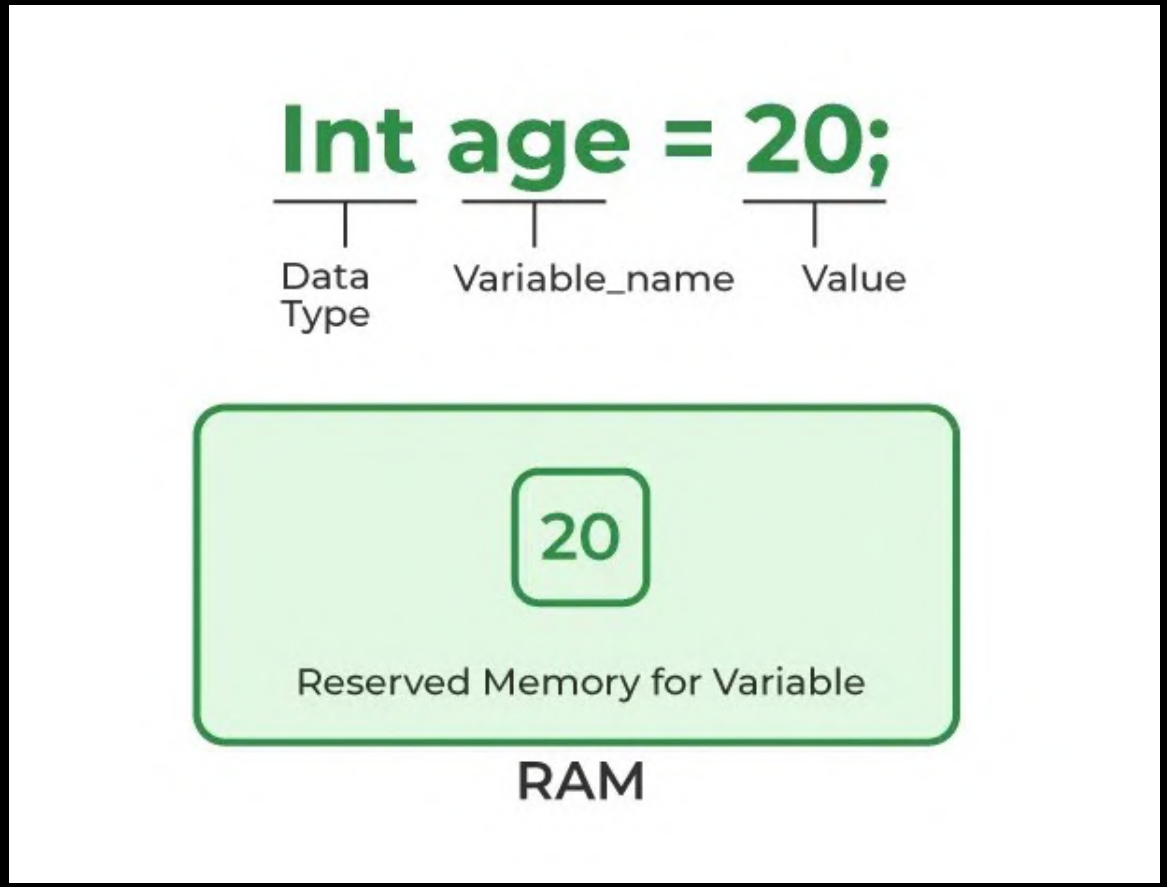
3.1. What are Variables?

Variables are like containers used for storing data values.

3.1. Memory Allocation



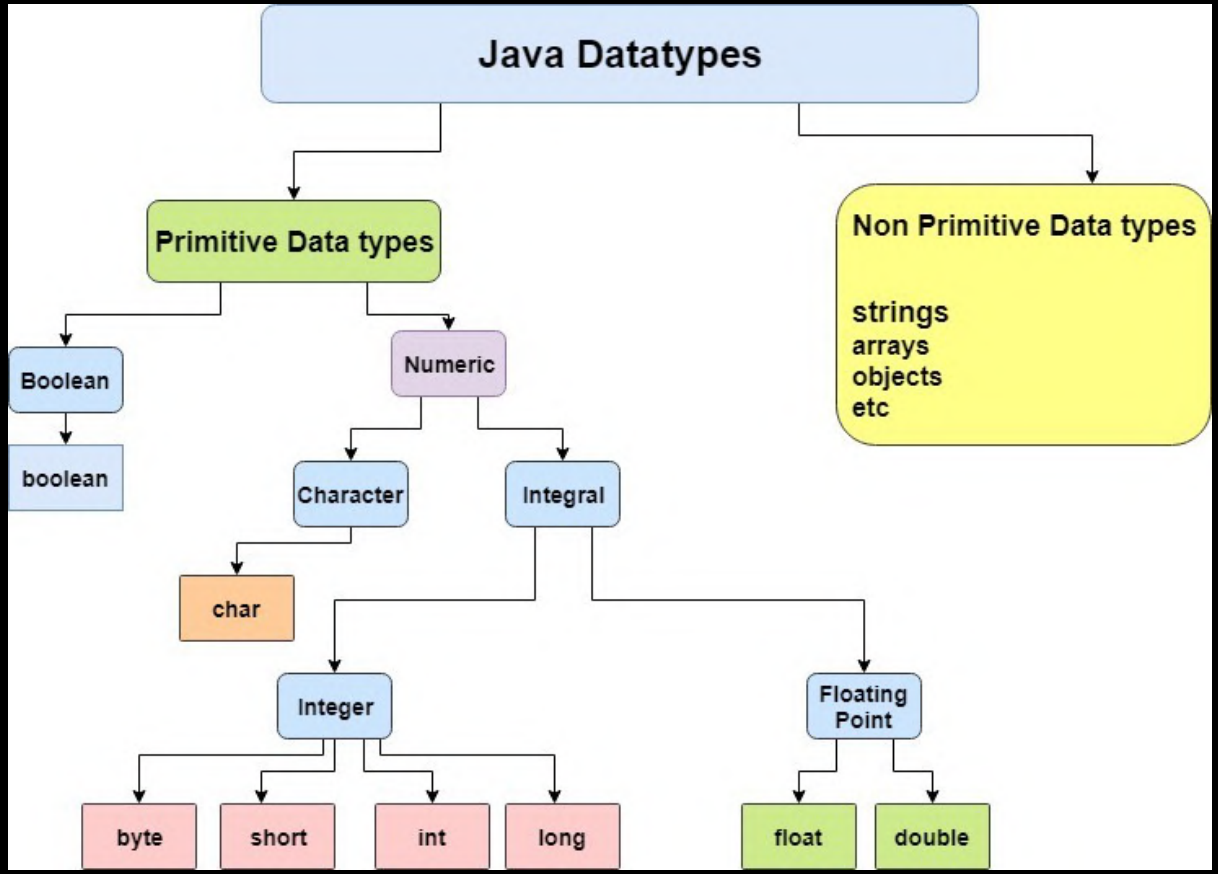
3.1. Variable Declaration



3.2 Data Types

	Size	Default Value	Type of Value Stored
byte	1 byte	0	Integral
short	2 byte	0	Integral
int	4 byte	0	Integral
long	8 byte	0L	Integral
char	2 byte	'\u0000'	Character
float	4 byte	0.0f	Decimal
double	8 byte	0.0d	Decimal
boolean	1 bit (till JDK 1.3 it uses 1 byte)	false	True or False

3.2 Data Types



3.3. Naming Conventions

camelCase

- Start with a lowercase letter. Capitalize the first letter of each subsequent word.
- Example: myVariableName

snake_case

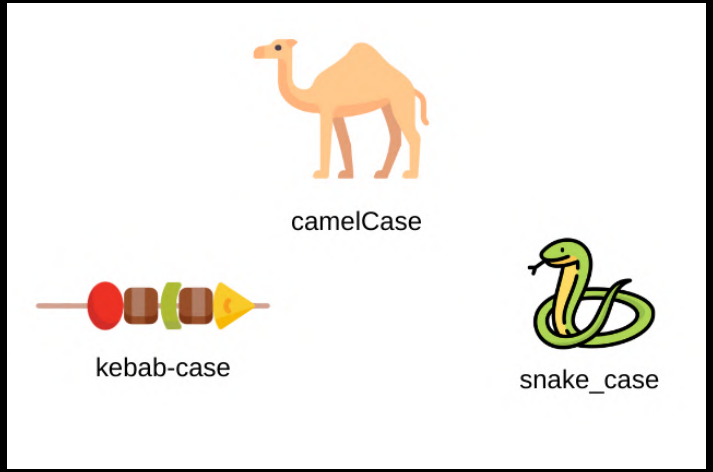
- Start with an lowercase letter. Separate words with underscore
- Example: my_variable_name

Kebab-case

- All lowercase letters. Separate words with hyphens. Example: my-variable-name

Keep a Good and Short Name

- Choose names that are descriptive but not too long. It should make it easy to understand the variable's purpose.
- Example: age, firstName, isMarried



3.3. Java Identifier Rules

1. The only allowed characters for identifiers are all alphanumeric characters([A-Z],[a-z],[0-9]), '$' (dollar sign) and '_' (underscore).
2. Can't use keywords or reserved words
3. Identifiers should not start with digits([0-9]).
4. Java identifiers are case-sensitive.
5. There is no limit on the length of the identifier but it is advisable to use an optimum length of 4 – 15 letters only.

1)_$_ (valid)
2)Ca$h (valid)
3)Java2share (valid)
4)all@hands (invalid)
5)123abc (invalid)
6)Total# (invalid)
7)Int (valid)
8)Integer (valid)
9)int (invalid)
10) tot123

3.4 Literals

Integer literals -> 10, 5, -8 etc...

Floating-point literals -> 1.2, 0.25, -1.999, etc...

Boolean literals -> true, false

Character literals -> ‘a’, ‘A’, ‘N’, ‘q’, etc...

String literals -> “hi”, “hello”, “What’s up?”

3.5 Keywords

abstract default super switch strictfp
void static break protected
volatile case implements assert
throw native throws
final class impact catch
transtient double
try new enum goto long
synchronized const byte char float
short interface return package
interface continue public
finally import boolean
private instanceof int extends

3.6 Escape Sequences

Escape Sequence	Description
\t	Insert a tab in the text at this point.
\b	Insert a backspace in the text at this point.
\n	Insert a newline in the text at this point.
\'	Insert a single quote character in the text at this point.
\"	Insert a double quote character in the text at this point.
\\	Insert a backslash character in the text at this point.

Task:

4. Show the following patterns using single print statement:

3.7 User Input

```java
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter your name:");
        String name = scanner.nextLine();
        System.out.println("Welcome " + name);
    }
}
```

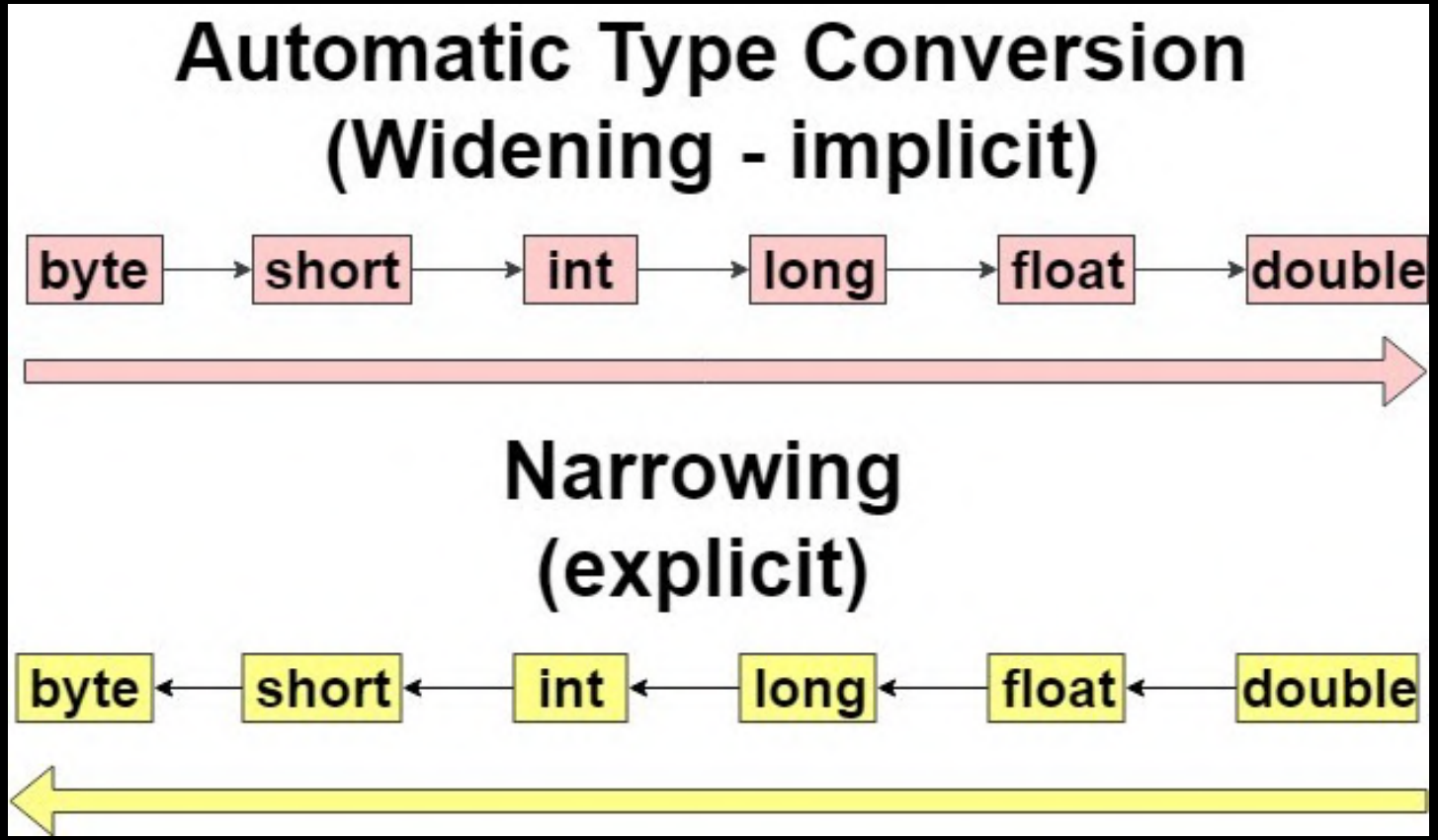
NAME
nextInt();
nextDouble();
nextFloat();
nextLong();
nextShort();
next();
nextLine();

CHALLENGE

5.Create a program to input name of the person and respond with ”Welcome NAME to KG Coding”

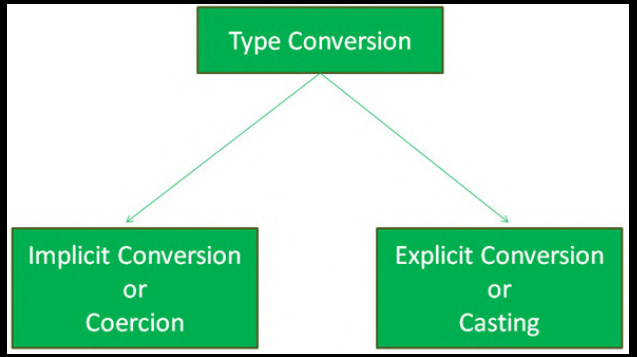
6.Create a program to add two numbers.

3.8 Type Conversion and Casting



3.8 Type Conversion and Casting

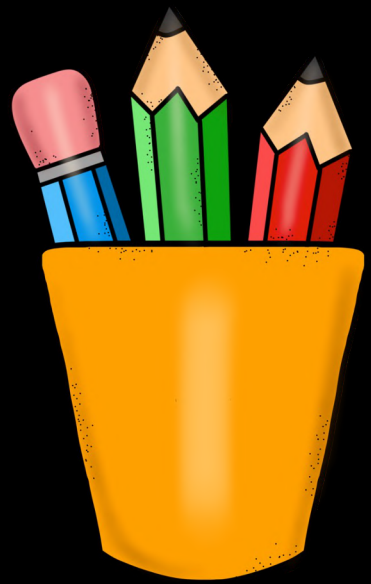
```java
// implicit
long big = 45;
float dec = 3;
double d = 3.4f;
```




```java
// explicit
float eDec = (float) 3.4;
long eBig = (long) 3.4;
int eInt = (int) 3.4;
```

Revision

1. Variables
2. Data Types
3. Naming Conventions
4. Literals
5. Keywords
6. Escape Sequences
7. User Input
8. Type Conversion and Casting



Practice Exercise

Data Types, Variables & Input

Answer in True/False:

1. In Java, a variable's name can start with a number.
2. char in Java can store a single character.
3. Class names in Java typically start with a lowercase letter.
4. 100L is a valid long literal in Java.
5. \d is an escape sequence in Java for a digit character.
6. Scanner class is used for reading console input.
7. In Java, an int can be automatically converted to a byte.
8. Java variable names are case-sensitive.
9. Scanner class can be used to read both primitive data types and strings.
10. Explicit casting is required to convert a double to an int.

Practice Exercise

Data Types, Variables & Input

Answer in True/False:

1. In Java, a variable's name can start with a number. — False
2. char in Java can store a single character. — True
3. Class names in Java typically start with a lowercase letter. — False
4. 100L is a valid long literal in Java. — True
5. \d is an escape sequence in Java for a digit character. — False
6. Scanner class is used for reading console input. — True
7. In Java, an int can be automatically converted to a byte. — False
8. Java variable names are case-sensitive. — True
9. Scanner class can be used to read both primitive data types and strings. — True
10. Explicit casting is required to convert a double to an int. — True

4. Operators, If-else & Number System

1. Assignment Operator
2. Arithmetic Operators
3. Order of Operation
4. Shorthand Operators
5. Unary Operators
6. If-else
7. Relational Operators
8. Logical Operators
9. Operator Precedence
10. Intro to Number System
11. Intro to Bitwise Operators

4.1 Assignment Operator

Assigns the right-hand operand's value to the left-hand operand.

Example: int a = 5;.

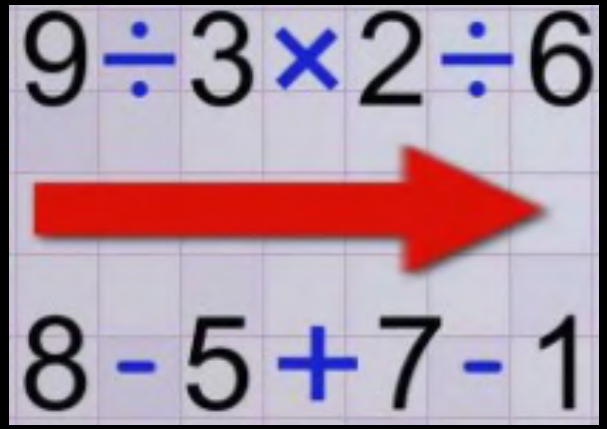
7. Create a program to swap two numbers.

4.2 Arithmetic Operators

Operators	Meaning	Example	Result
+	Addition	4+2	6
-	Subtraction	4-2	2
*	Multiplication	4*2	8
/	Division	4/2	2
%	Modulus operator to get remainder in integer division	5%2	1

4.3 Order of Operation

B	O	D	M	A	S
Bracket	Order	Divide	Multiply	Add	Subtract
( )	$\sqrt{x}$ x^2	÷ or x		+ or -	
Parentheses	Exponents	Multiply	Divide	Add	Subtract
P	E	M	D	A	S



4.4 Shorthand Operators

Operator symbol	Name of the operator	Example	Equivalent construct
+=	Addition assignment	x += 4;	x = x + 4;
-=	Subtraction assignment	x -= 4;	x = x - 4;
*=	Multiplication assignment	x *= 4;	x = x * 4;
/=	Division assignment	x /= 4;	x = x / 4;
%=	Remainder assignment	x %= 4;	x = x % 4;

4.5 Unary Operators

Operator	Description	Example
-	Converts a positive value to a negative	`x = -y`
Pre Increment	Increment the value by 1 and then use it in our statement	`x = ++y`
Pre Decrement	Decrement the value by 1 and then use it in our statement	`x = --y`
Post Increment	Use current value in the statement and then increment it by 1	`x = y++`
Post Decrement	Use current value in the statement and then decrement it by 1	`x = y--`

CHALLENGE

8. Create a program that takes two numbers and shows result of all arithmetic operators (+,-,*,/,%).
9. Create a program to calculate product of two floating points numbers.
10. Create a program to calculate Perimeter of a rectangle.
    Perimeter of rectangle ABCD = A+B+C+D
11. Create a program to calculate the Area of a Triangle.
    Area of triangle = ½*B*H
12. Create a program to calculate simple interest.
    Simple Interest = (P x T x R)/100
13. Create a program to calculate Compound interest.
    Compound Interest = P(1 + R/100)t
14. Create a program to convert Fahrenheit to Celsius
    °C = (°F - 32) × 5/9

4.6 if-else

1. Syntax: Uses if () {} to check a condition.
2. What is if: Executes block if condition is true, skips if false.
3. What is else: Executes a block when the if condition is false.
4. Curly Braces can be omitted for single statements, but not recommended.
5. If-else Ladder: Multiple if and else if blocks; only one executes.
6. Use Variables: Can store conditions in variables for use in if statements.

4.7 Relational Operators

< , > , <= , >= , == , !=

Equality

- == Checks value equality.

Inequality

- != Checks value inequality.

Relational

- > Greater than.
- < Less than.
- >= Greater than or equal to.
- <= Less than or equal to.

Order of Relational operators is less than arithmetic operators

4.8 Logical Operators


AND


Or

NOT

1. Types: && (AND), || (OR), ! (NOT)
2. AND (&&): All conditions must be true for the result to be true.
3. OR (||): Only one condition must be true for the result to be true.
4. NOT (!): Inverts the Boolean value of a condition.
5. Lower Priority than Math and Comparison operators

CHALLENGE

15.Create a program that determines if a number is positive, negative, or zero.

16.Create a program that determines if a number is odd or even.

17. Create a program that determines the greatest of the three numbers.

18.Create a program that determines if a given year is a leap year (considering conditions like divisible by 4 but not 100, unless also divisible by 400).

19.Create a program that calculates grades based on marks

A -> above 90% B -> above 75%

C -> above 60% D -> above 30%

F -> below 30%

20.Create a program that categorize a person into different age groups

Child -> below 13 Teen -> below 20

Adult -> below 60 Senior-> above 60

4.9 Operator Precedence

Operator Type	Category	Precedence	Associativity
Unary	postfix	a++, a--	Right to left
	prefix	++a, --a, +a, -a, ~, !	Right to left
Arithmetic	Multiplication	*, /, %	Left to Right
	Addition	+, -	Left to Right
Shift	Shift	<<, >>, >>>	Left to Right
Relational	Comparison	<, >, <=, >=, instanceOf	Left to Right
	equality	==, !=	Left to Right
Bitwise	Bitwise AND	&	Left to Right
	Bitwise exclusive OR	^	Left to Right
	Bitwise inclusive OR	|	Left to Right
Logical	Logical AND	&&	Left to Right
	Logical OR	||	Left to Right
Ternary	Ternary	? :	Right to Left
Assignment	assignment	=, +=, -=, *=, /=, %=, &=, ^=, |=, <<=, >>=, >>>=	Right to Left

Operator Precedence: Determines the evaluation order of operators in an expression based on their priority levels.

Associativity: Defines the order of operation for operators with the same precedence, usually left-to-right or right-to-left.

4.10 Intro to Number System

Number System	Base or Radix	Digits or symbols
Binary	2	0,1
Octal	8	0,1,2,3,4,5,6,7
Hexadecimal	16	0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F

Number System	Base or Radix	Digits or symbols
Unary	1	0/1
Decimal	10	0,1,2,3,4,5,6,7,8,9

4.10 Intro to Number System

Decimal Number System

- Decimal Number System is a base-10 system that has ten digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.
- The decimal number system is said to be of base, or radix, 10 because it uses 10 digits and the coefficients are multiplied by powers of 10.
- This is the base that we often use in our day to day life.
- Example: $(7,392)_{10}$, where 7,392 is a shorthand notation for what should be written as

$$7 * 10^3 + 3 * 10^2 + 9 * 10^1 + 2 * 10^0$$

4.10 Intro to Number System

Binary Number System

- The coefficients of the binary number system have only two possible values: 0 and 1.
- Each coefficient a_j is multiplied by a power of the radix, e.g., 2^j, and the results are added to obtain the decimal equivalent of the number.

Example: $(11010.11)_2$ value in Decimal?

$$1 * 2^4 + 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 0 * 2^0 + 1 * 2^{-1} + 1 * 2^{-2} = 26.75$$

4.10 Intro to Number System

Decimal	Binary
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

4.11 Intro to Bitwise Operators

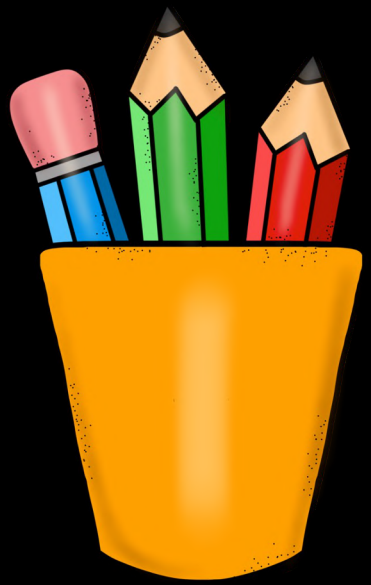
1.AND Operator (&): Performs on two integers. Each bit of the output is 1 if the corresponding bits of both operands are 1, otherwise 0.
2.OR Operator (|): Performs on two integers. Each bit of the output is 0 if the corresponding bits of both operands are 0, otherwise 1.
3.XOR Operator (^): Performs on two integers. Each bit of the output is 1 if the corresponding bits of the operands are different.
4.NOT Operator (~): Performs a bitwise complement. It inverts the bits of its operand (0 becomes 1, and 1 becomes 0).
5.Left Shift Operator (<<): Shifts the left operand's bits to the left by the number of positions specified by the right operand, filling the new rightmost bits with zeros.
6.Right Shift Operator (>>): Shifts the left operand's bits to the right. If the left operand is positive, zeros are filled into the new leftmost bits; if negative, ones are filled in.

CHALLENGE

21. Create a program that shows bitwise AND of two numbers.
22. Create a program that shows bitwise OR of two numbers.
23. Create a program that shows bitwise XOR of two numbers.
24. Create a program that shows bitwise compliment of a number.
25. Create a program that shows use of left shift operator.
26. Create a program that shows use of right shift operator.
27. Write a program to check if a given number is even or odd using bitwise operators.

Revision

1. Assignment Operator
2. Arithmetic Operators
3. Order of Operation
4. Shorthand Operators
5. Unary Operators
6. If-else
7. Relational Operators
8. Logical Operators
9. Operator Precedence
10. Intro to Number System
11. Intro to Bitwise Operators



Practice Exercise

Operators, If-else & Number System

Answer in True/False:

1.In Java, && and || operators perform short-circuit evaluation.
2.In an if-else statement, the else block executes only when the if condition is false.
3.Java allows an if statement without the else part.
4.The ^ operator in Java is used for exponentiation.
5.Unary minus operator can be used to negate the value of a variable in Java.
6.a += b is equivalent to a = a + b in Java.
7.In Java, the binary number system uses base 10.
8.The number 1010 in binary is equivalent to 10 in decimal.
9.& and | are logical operators in Java.
10.In Java, a >> 2 shifts the binary bits of a to the left by 2 positions.

Practice Exercise

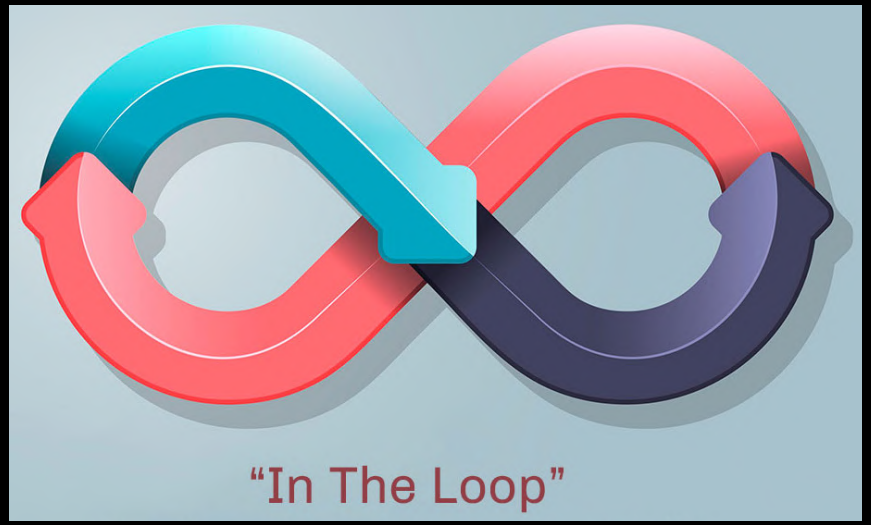
Operators, If-else & Number System

Answer in True/False:

1. In Java, && and || operators perform short-circuit evaluation. — True
2. In an if-else statement, the else block executes only when the if condition is false. — True
3. Java allows an if statement without the else part. — True
4. The ^ operator in Java is used for exponentiation. — False
5. Unary minus operator can be used to negate the value of a variable in Java. — True
6. a += b is equivalent to a = a + b in Java. — True
7. In Java, the binary number system uses base 10. — False
8. The number 1010 in binary is equivalent to 10 in decimal. — True
9. & and | are logical operators in Java. — False
10. In Java, a >> 2 shifts the binary bits of a to the left by 2 positions. — False

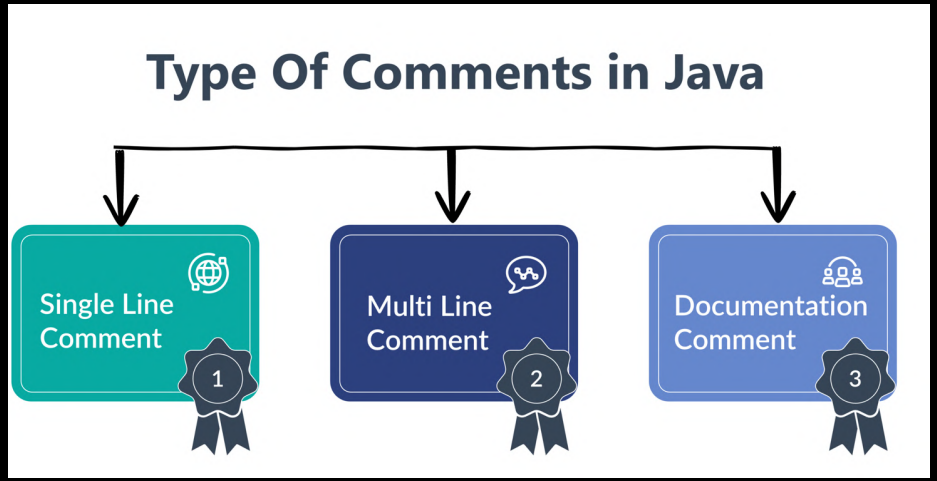
5 While Loop, Methods & Arrays

1. Comments
2. While Loop
3. Methods
4. Return statement
5. Arguments
6. Arrays
7. 2D Arrays

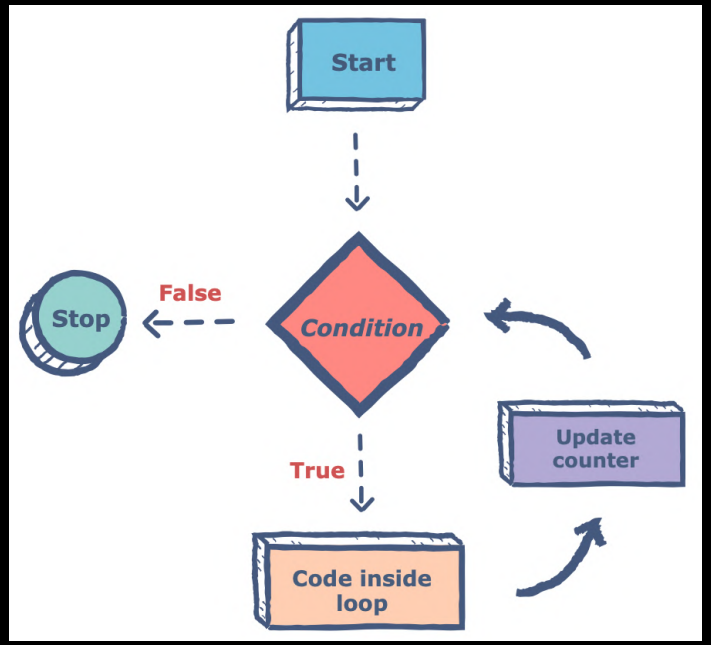


5.1 Java Comments

1. Used to add notes in Java code
2. Not displayed on the web page
3. Helpful for code organization
4. Syntax:
    1. Single Line: //
    2. Multi Line: /* */
    3. Java Docs: /** */

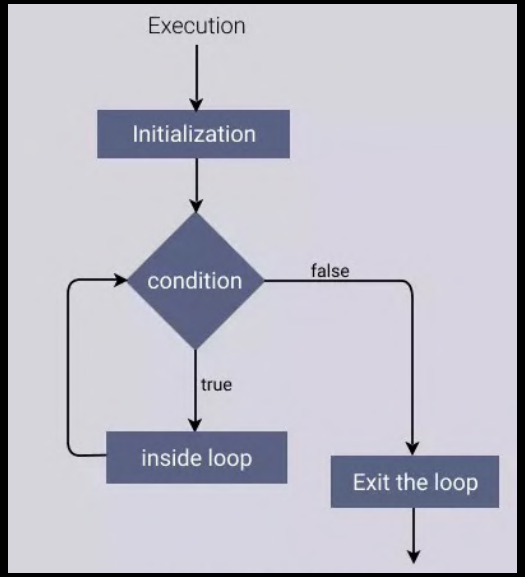


5.2 What is a Loop?



1. Code that runs multiple times based on a condition.
2. Repeated execution of code.
3. Loops automate repetitive tasks.
4. Types of Loops: while, for, while, do-while.
5. Iterations: Number of times the loop runs.

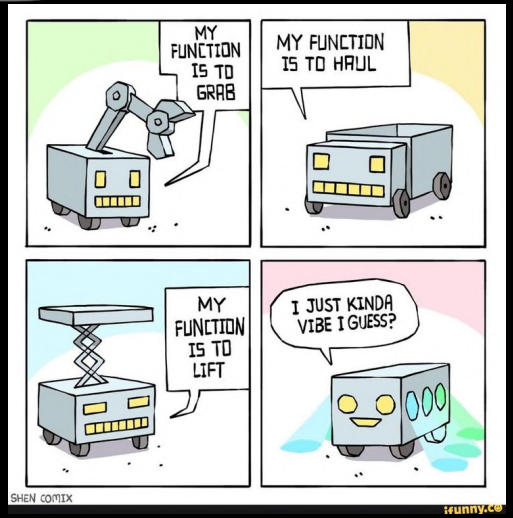
5.2 While Loop




```
while (condition) {
  // Body of the loop
}
```

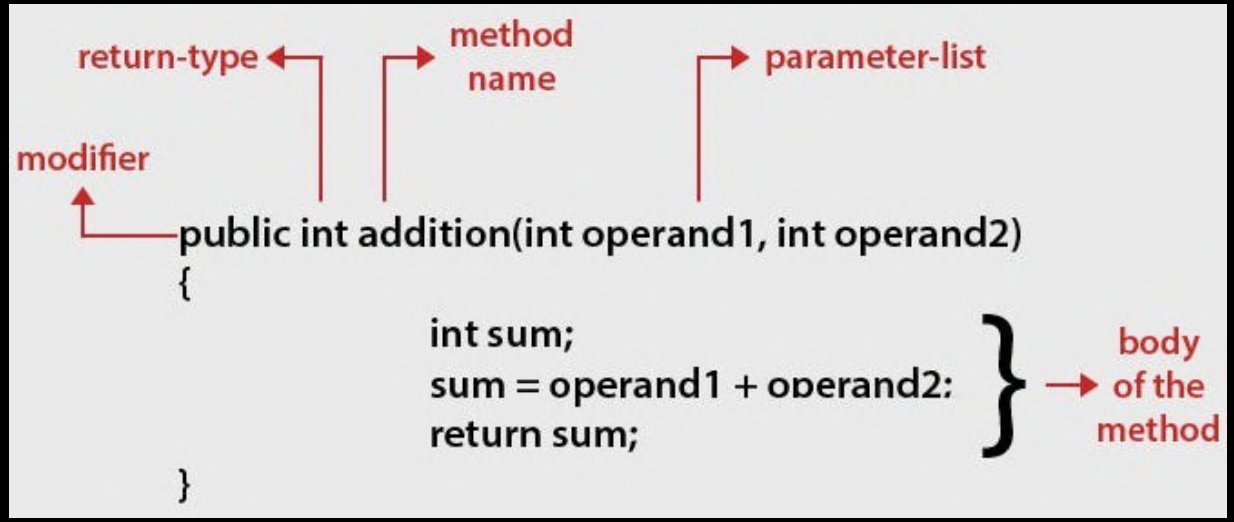
1. Iterations: Number of times the loop runs.
2. Used for non-standard conditions.
3. Repeating a block of code while a condition is true.
4. Remember: Always include an update to avoid infinite loops.

5.3 What are Functions / Methods?



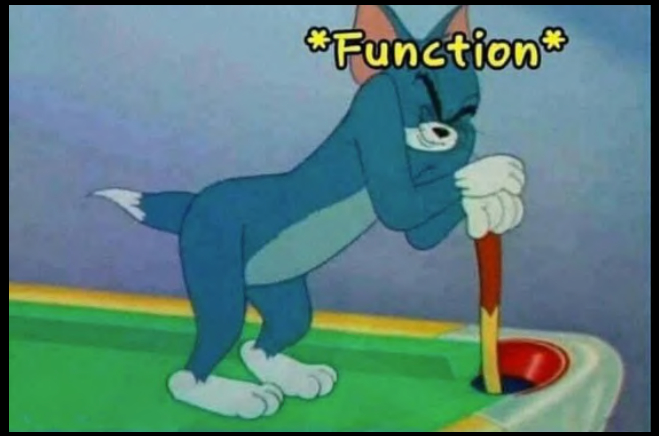
1. Definition: Blocks of reusable code.
2. DRY Principle: "Don't Repeat Yourself" it Encourages code reusability.
3. Usage: Organizes code and performs specific tasks.
4. Naming Rules: Same as variable names: camelCase
5. Example: ”Beta Gas band kar de"

5.3 Method Syntax



1. Follows same rules as variable names.
2. Use () to contain parameters.
3. Invoke by using the function name followed by ().
4. Fundamental for code organization and reusability.

5.4 Return statement



1. Sends a value back from a function.
2. Example: "Ek glass paani laao"
3. What Can Be Returned: Value, variable, calculation, etc.
4. Return ends the function immediately.
5. Function calls make code jump around.
6. Prefer returning values over using global variables.

5.5 Arguments vs Parameters

Parameter

Function

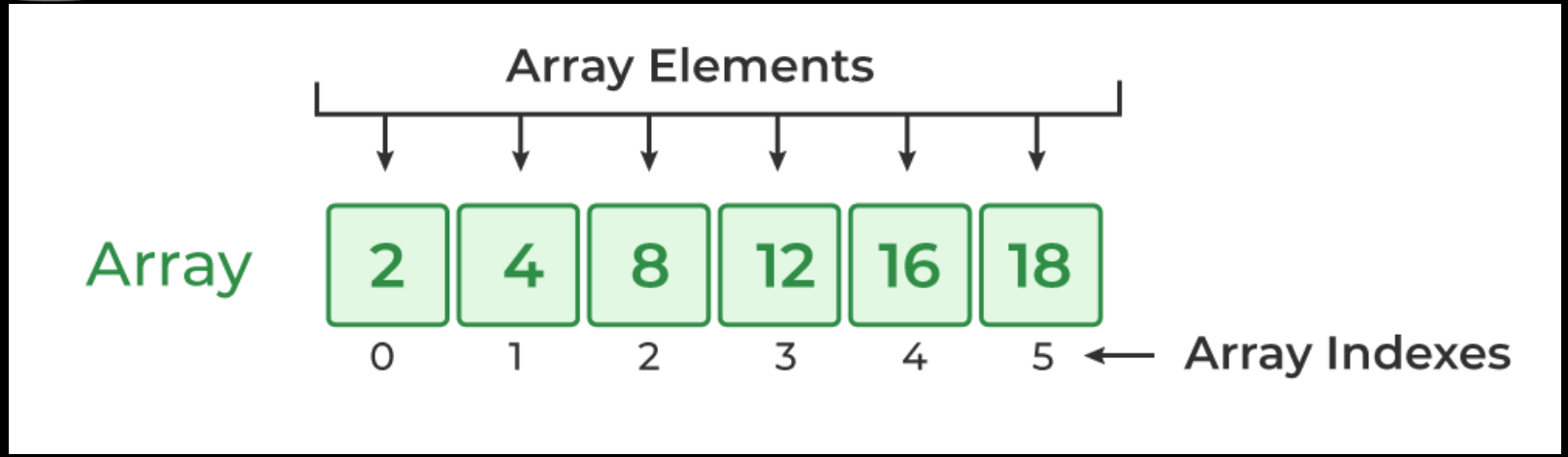
Return

1. Input values that a function takes.
2. Parameters put value into function, while return gets value out.
3. Example: "Ek packet dahi laao"
4. Naming Convention: Same as variable names.
5. Parameter vs Argument
6. Examples: System.out.print, Scanner.nextInt(), are functions we have already used
7. Multiple Parameters: Functions can take more than one.
8. Default Value: Can set a default value for a parameter.

CHALLENGE

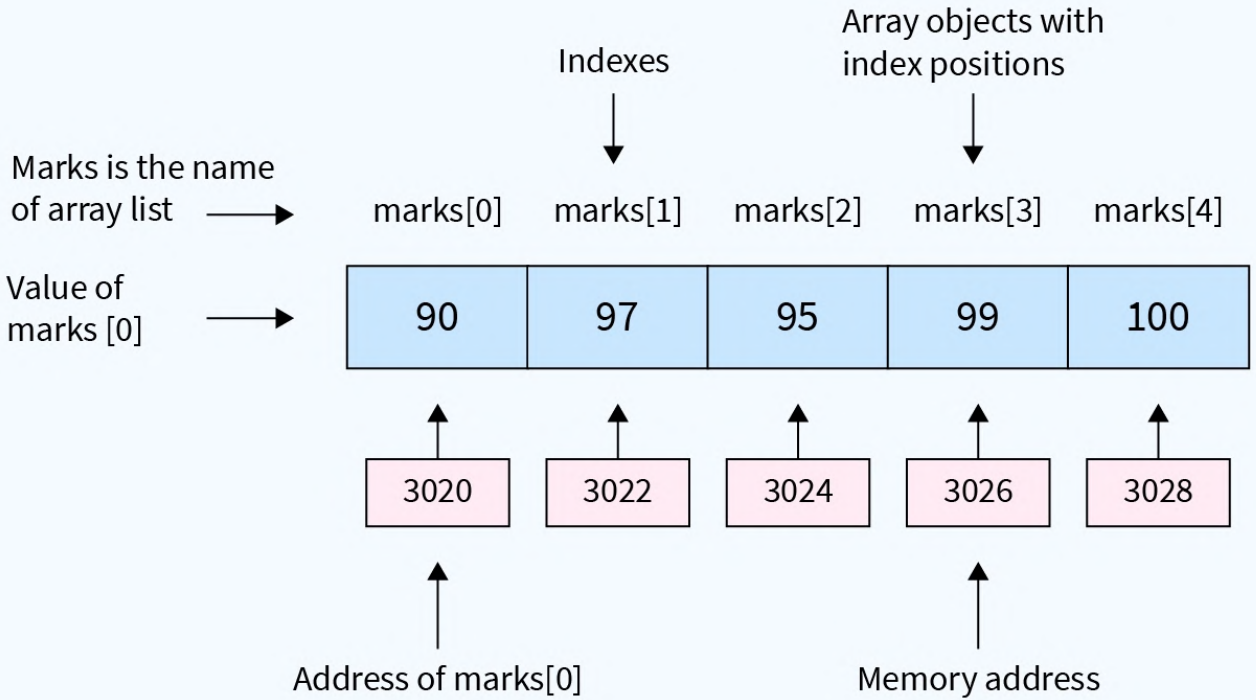
28. Develop a program that prints the multiplication table for a given number.
29. Create a program to sum all odd numbers from 1 to a specified number N.
30. Write a function that calculates the factorial of a given number.
31. Create a program that computes the sum of the digits of an integer.
32. Create a program to find the Least Common Multiple (LCM) of two numbers.
33. Create a program to find the Greatest Common Divisor (GCD) of two integers.
34. Create a program to check whether a given number is prime.
35. Create a program to reverse the digits of a number.
36. Create a program to print the Fibonacci series up to a certain number.
37. Create a program to check if a number is an Armstrong number.
38. Create a program to verify if a number is a palindrome.
39. Create a program that print patterns:

5.6 What is an Array?

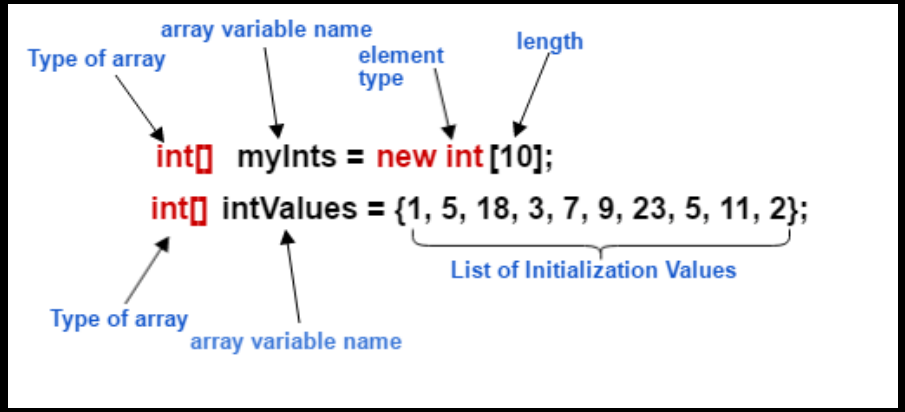


1. An Array is just a list of values.
2. Index: Starts with 0.
3. Arrays are used for storing multiple values in a single variable.

5.6 Array Memory



5.6 Array Syntax



1. Use {} to create a new array, {} brackets enclose list of values
2. Arrays can be saved to a variable.
3. Accessing Values: Use [] with index.
4. Syntax Rules:
   - Brackets start and end the array.
   - Values separated by commas.
   - Can span multiple lines.

5.6 Array Length

Array length = Array's Last Index + 1

1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7

Array's last index = 7

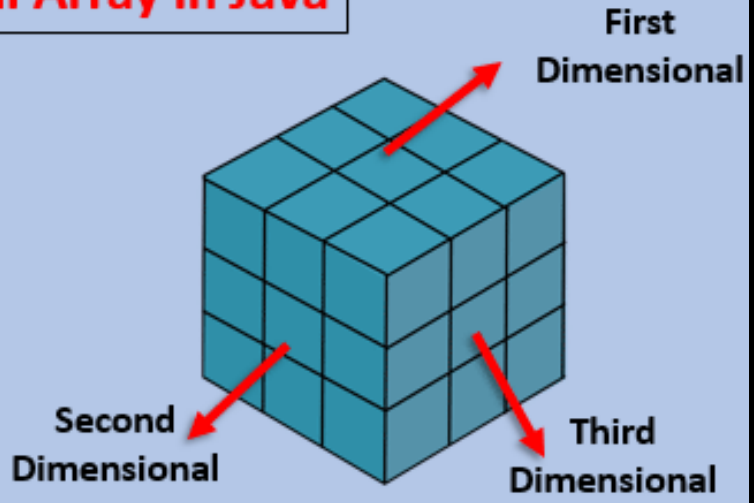
Arraylength = 7 + 1 = 8

5.7 2D Arrays

Types of Multidimensional Array in Java

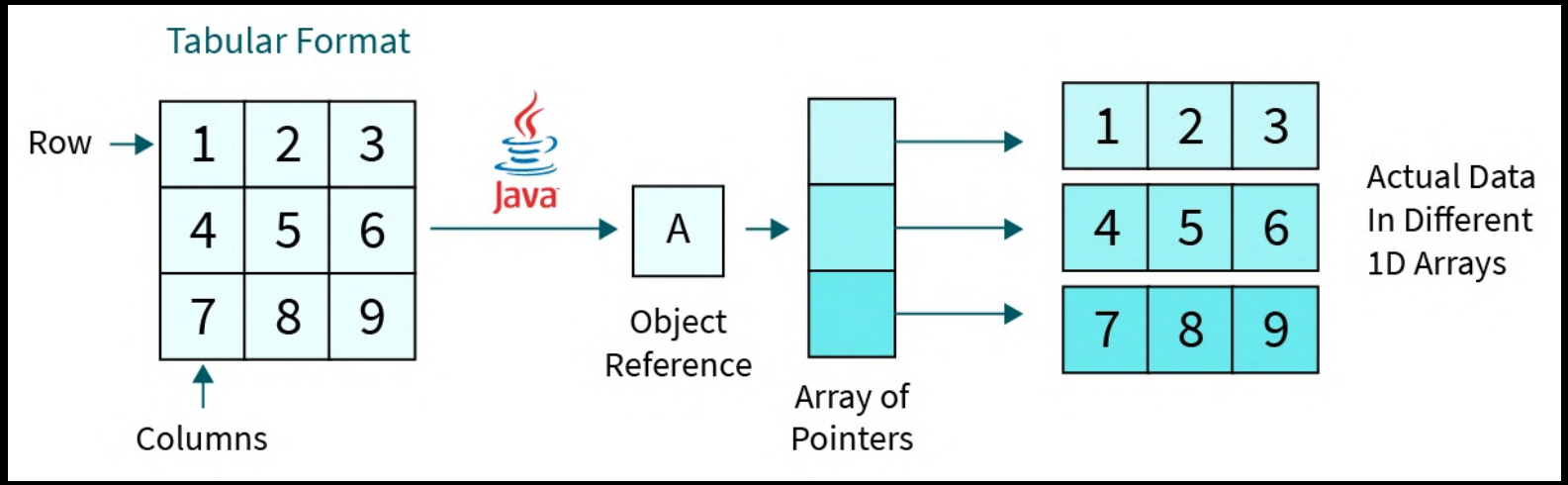
	Column 0	Column 1	Column 2
Row 0	X[0][0]	X[0][1]	X[0][2]
Row 1	X[1][0]	X[1][1]	X[1][2]
Row 2	X[2][0]	X[2][1]	X[2][2]

2D-Array



3D-Array

5.7 2D Arrays


Tabular Format
Row
1 2 3
4 5 6
7 8 9
Columns
Java
A
Object Reference
Array of Pointers
1 2 3
4 5 6
7 8 9
Actual Data In Different 1D Arrays

5.7 2D Arrays

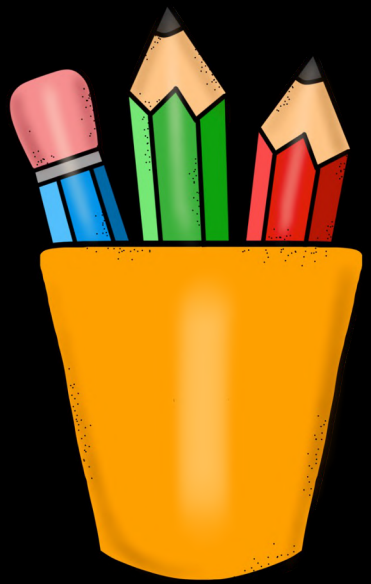
```
int[][] numArr = new int[2][3];
int[][] inArray = {{1, 2, 5}, {8, 9, 4}};

numArr[0][0] = 5;
```

CHALLENGE

40. Create a program to find the sum and average of all elements in an array.
41. Create a program to find number of occurrences of an element in an array.
42. Create a program to find the maximum and minimum element in an array.
43. Create a program to check if the given array is sorted.
44. Create a program to return a new array deleting a specific element.
45. Create a program to reverse an array.
46. Create a program to check is the array is palindrome or not.
47. Create a program to merge two sorted arrays.
48. Create a program to search an element in a 2-D array.
49. Create a program to do sum and average of all elements in a 2-D array.
50. Create a program to find the sum of two diagonal elements.

Revision

1. Comments
2. While Loop
3. Methods
4. Return statement
5. Arguments
6. Arrays
7. 2D Arrays



Practice Exercise

While Loop, Methods & Arrays

Answer in True/False:

1. A 'while' loop in Java continues to execute as long as its condition is true.
2. The body of a 'while' loop will execute at least once, regardless of the condition.
3. A 'while' loop cannot be used for iterating over an array.
4. Infinite loops are not possible with 'while' loops in Java.
5. A method in Java can return more than one value at a time.
6. It's mandatory for every Java method to have a return type.
7. The size of an array in Java can be modified after it is created.
8. Arrays in Java can contain elements of different data types.
9. An array is a reference type in Java.
10. Java arrays are zero-indexed, meaning the first element has an index of 0.

Practice Exercise

While Loop, Methods & Arrays

Answer in True/False:

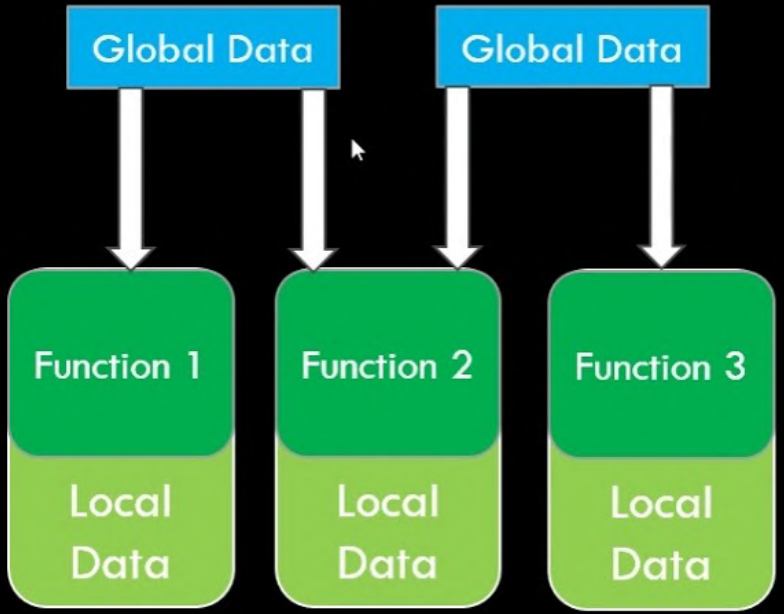
1. A 'while' loop in Java continues to execute as long as its condition is true.	True
2. The body of a 'while' loop will execute at least once, regardless of the condition.	False
3. A 'while' loop cannot be used for iterating over an array.	False
4. Infinite loops are not possible with 'while' loops in Java.	False
5. A method in Java can return more than one value at a time.	False
6. It's mandatory for every Java method to have a return type.	True
7. The size of an array in Java can be modified after it is created.	False
8. Arrays in Java can contain elements of different data types.	False
9. An array is a reference type in Java.	True
10. Java arrays are zero-indexed, meaning the first element has an index of 0.	True

6 Classes & Objects

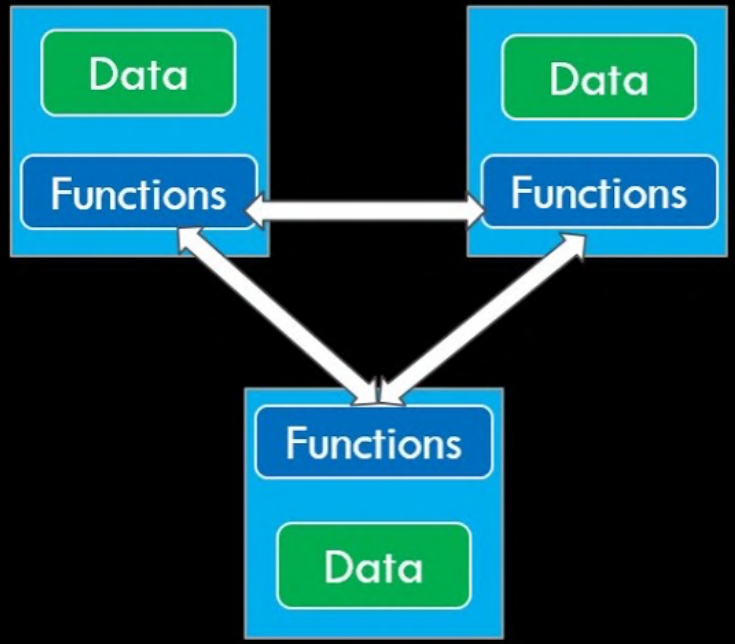
1. Process vs Object Oriented
2. Instance Variables and Methods
3. Declaring and Using Objects
4. Class vs Object
5. This & Static Keyword
6. Constructors & Code Blocks
7. Stack vs Heap Memory
8. Primitive vs Reference Types
9. Variable Scopes
10. Garbage Collection & Finalize

6.1 Process vs Object Oriented

Procedural Oriented Programming

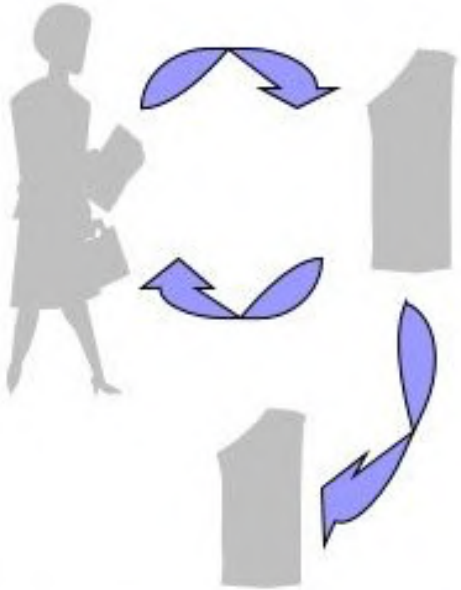


Object Oriented Programming



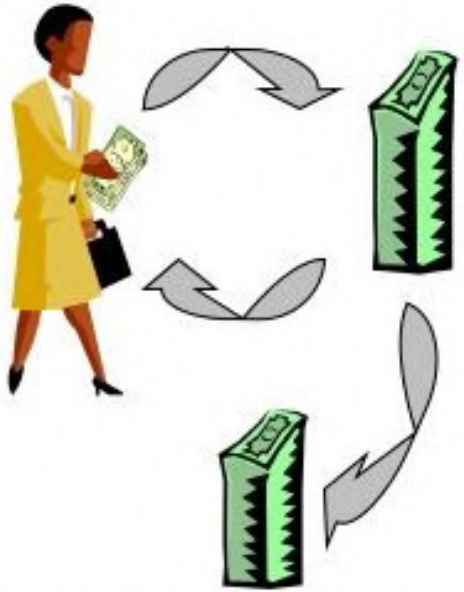
6.1 Process vs Object Oriented

- Procedural



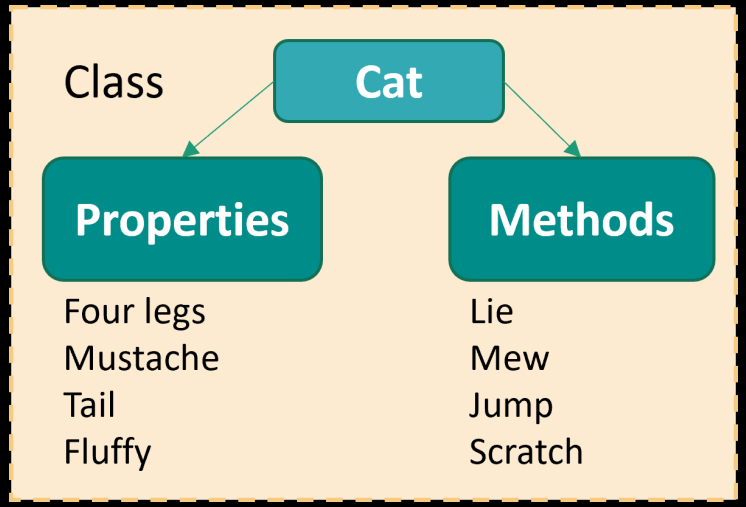
Withdraw, deposit, transfer

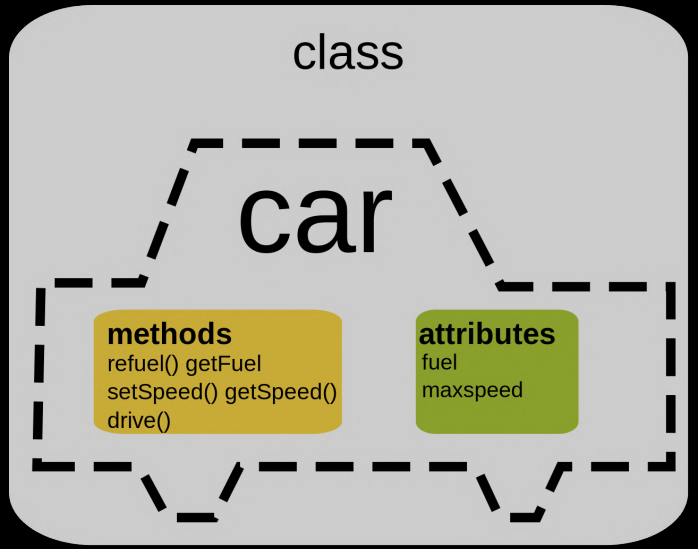
- Object Oriented



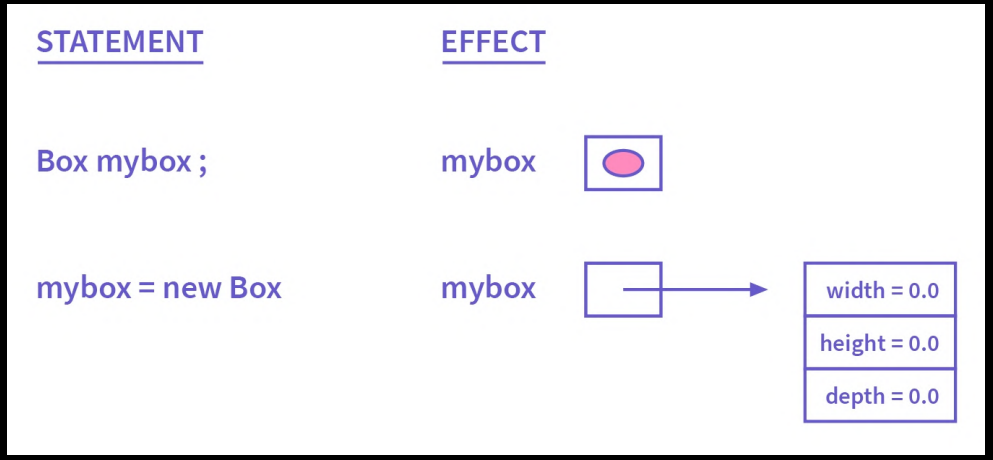
Customer, money, account

6.2 Instance Variables and Methods



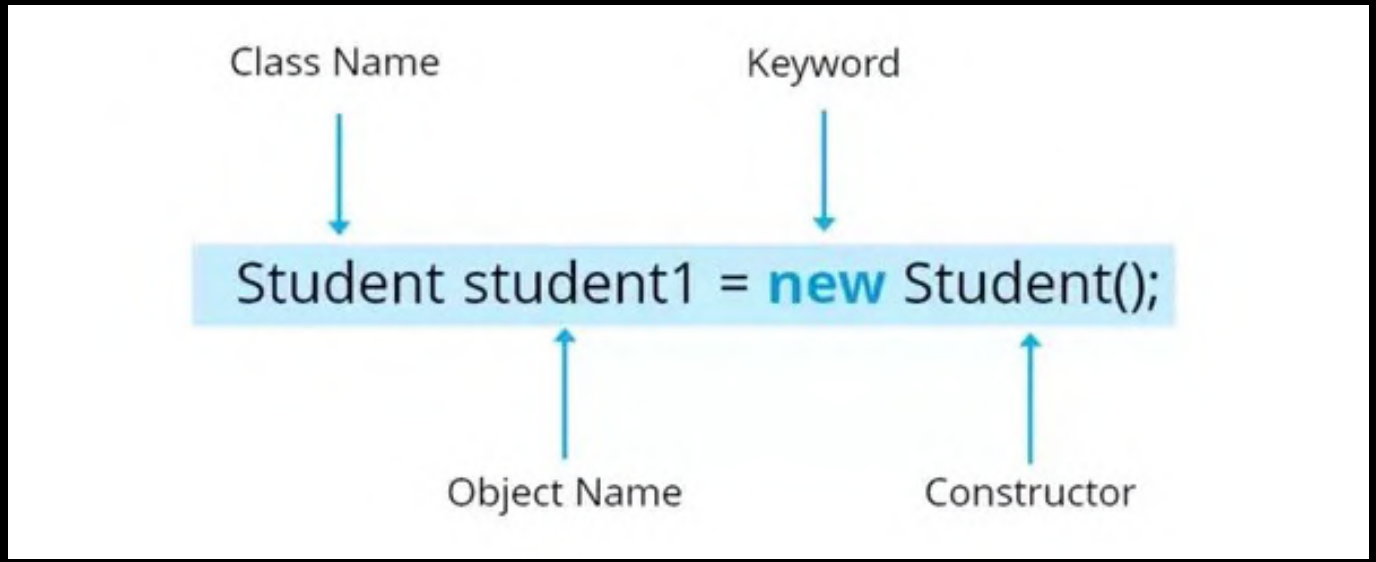


6.3 Declaring Objects

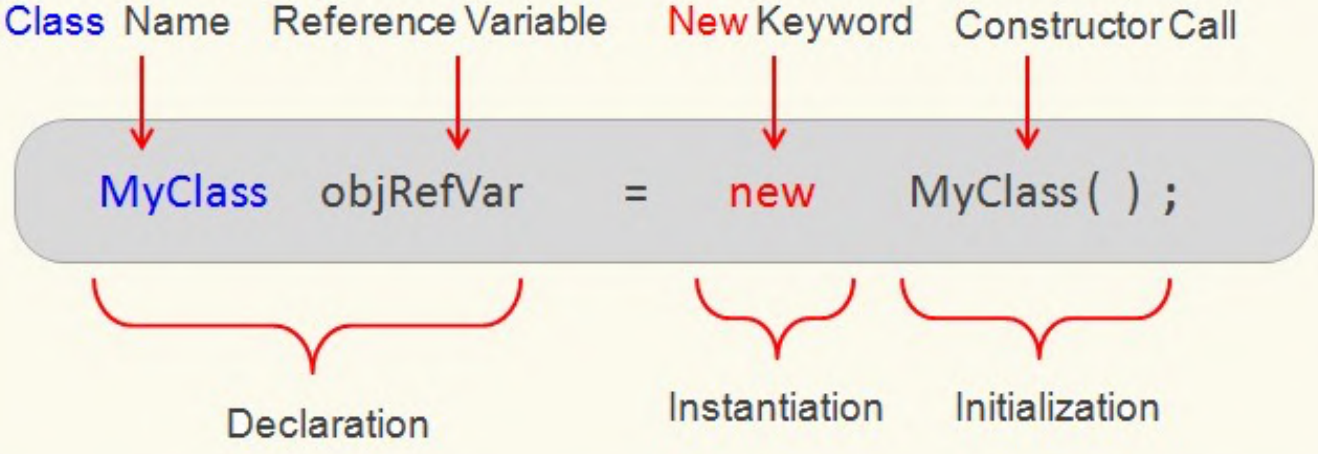


1. Object Creation: new instantiates a new object of a class.
2. Memory Allocation: Allocates memory for the object in the heap.
3. Constructor Invocation: Calls the class constructor to initialize the object.
4. Reference Return: Returns a reference to the created object.
5. Array Creation: Also used for creating arrays, like int[] arr = new int[5];.
6. Dynamic Allocation: Unlike static allocation, new allows for dynamic memory allocation, allocating memory at runtime.

6.3 Declaring Objects Syntax



6.3 Declaring Objects Syntax



objRefVar is a Reference Variable of MyClass Type .

new is a Java Keyword used To Instantiate a Class .

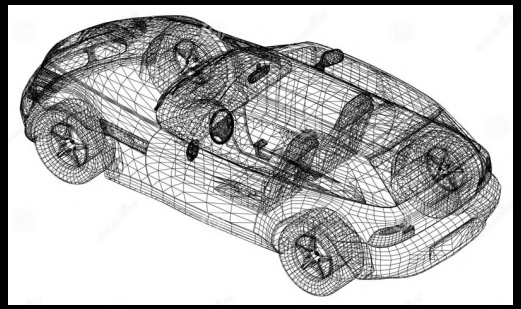
6.3 Using Objects

Access properties using . Operator like product.price

6.4 Class vs Object

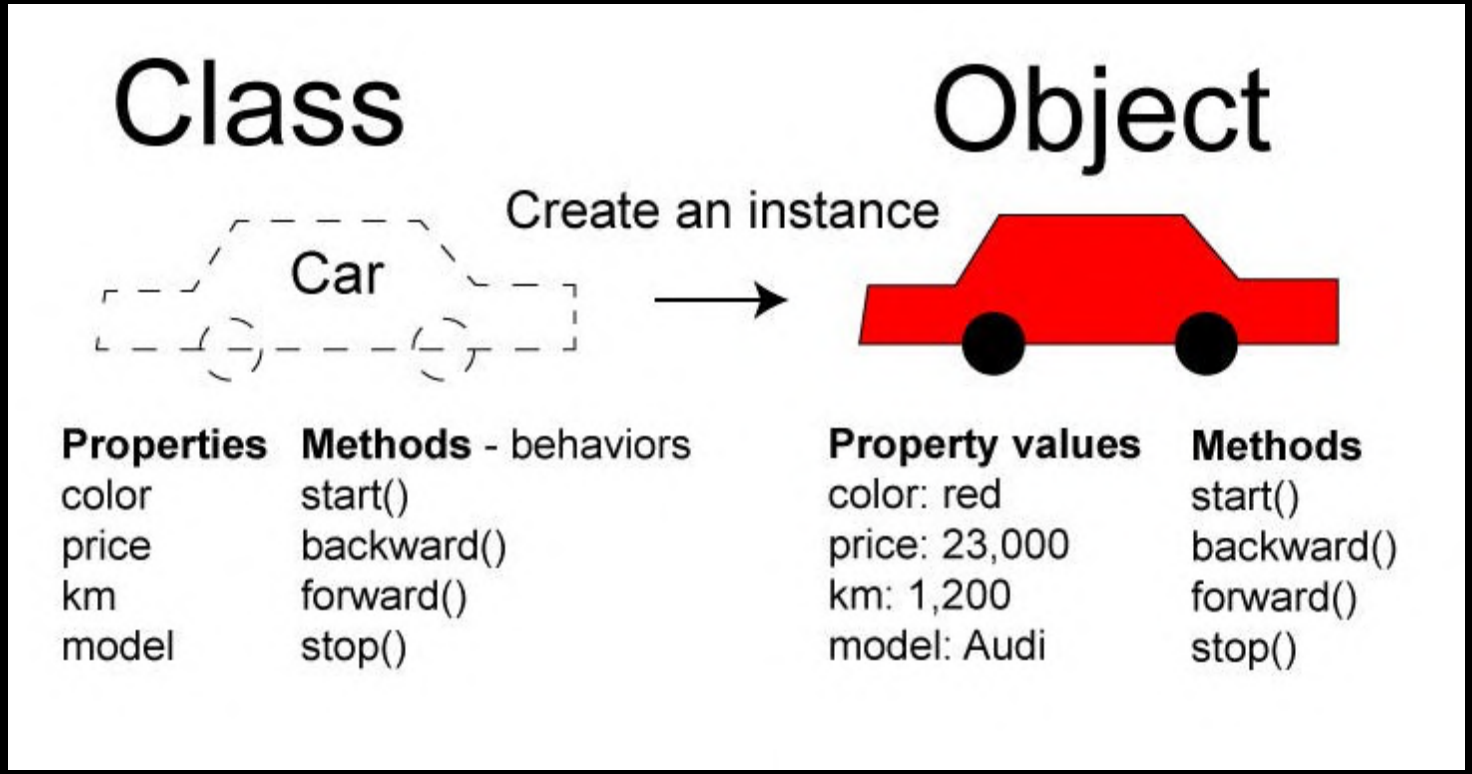
Class is a blueprint; Objects are real values in memory.

6.4 Class vs Object



Class is a blueprint; Objects are real values in memory.

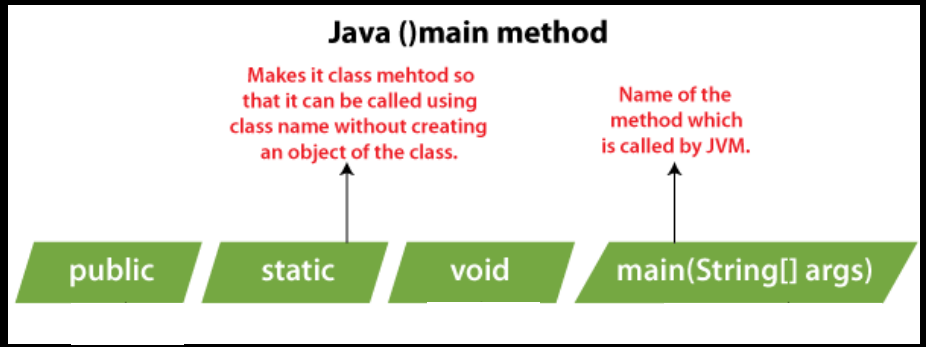
6.4 Class vs Object



6.5 This Keyword

1. Current Instance: Refers to the current class instance variable.
2. Constructor Call: Can be used to invoke a constructor of the same class (this()).
3. Method Call: Invokes a method of the current object.
4. Pass as Argument: Can be passed as an argument in the method call.
5. Return the Current Class Instance: Can return the current class instance from the method.

6.5 Static Keyword



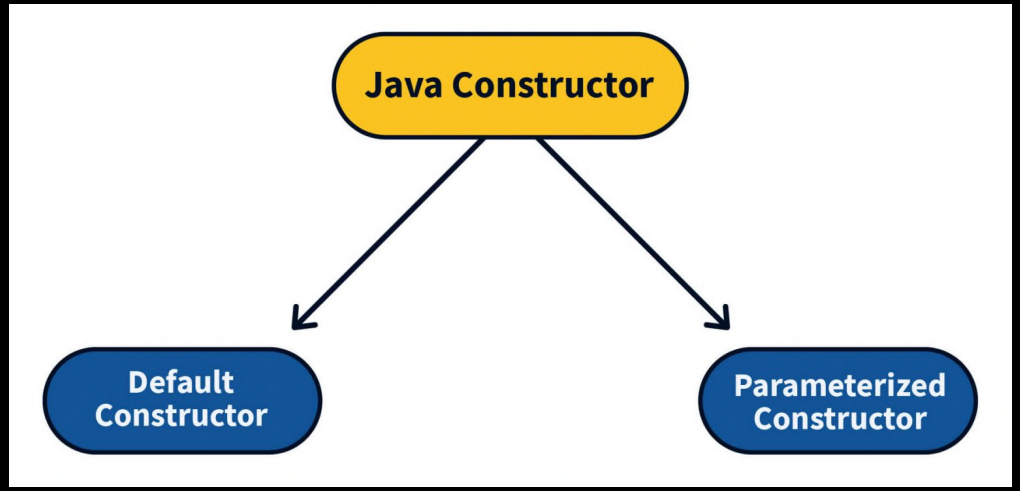
1. Static Variables: Belong to the class, not individual instances. Shared among all instances of the class.
2. Static Methods: Can be called without creating an object of the class. Can only directly access static variables and other static methods.
3. No Access to Non-static Members: Static methods and blocks cannot directly access non-static members (variables and methods) of the class.

6.6 Constructors

```java
public class Car {
    1 usage
    String color;
    1 usage
    float price;

    1 usage   new *
    Car() { // Default constructor
        color = "Black";
        price = 50000;
    }

    new *
    public static void main(String[] args) {
        Car swift = new Car();
    }
}
```

1. Purpose: Constructors initialize new objects and set initial states for the object's attributes.
2. Naming: A constructor must have the same name as the class in which it is declared.
3. No Return Type: Constructors do not have a return type, not even void.
4. Automatic Calls: A constructor is automatically called when an instance of a class is created.

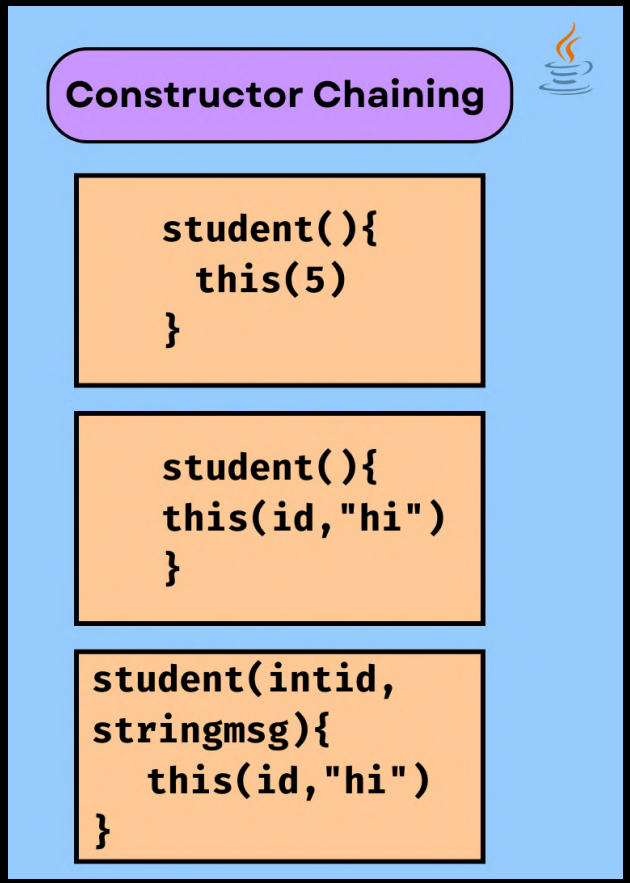
6.6 Constructors (Types)



1. Default Constructor: If no constructor is explicitly defined, Java provides a default constructor that initializes all member variables to default values.
2. Parameterized Constructors: Constructors can have parameters to pass values when creating an object, allowing for different initializations.

```
Car(String carColor, float currPrice) {
    color = carColor;
    price = currPrice;
}
```

6.6 Constructors (Chaining)



1. Within Same Class: Using this() to call another constructor in the same class.
2. First Statement: this() must be the first statement in a constructor.
3. No Loop: Constructor chaining can't form a loop; it must have a termination point.

6.6 Code Blocks

```
{
    System.out.println("This is a Initialization Block");
    color = "Black";
    price = 50000;
}
```

```
if (true) { // code block
    System.out.println("Code Block");
}
```

```
static {
    System.out.println("This is a Static Block");
}
```

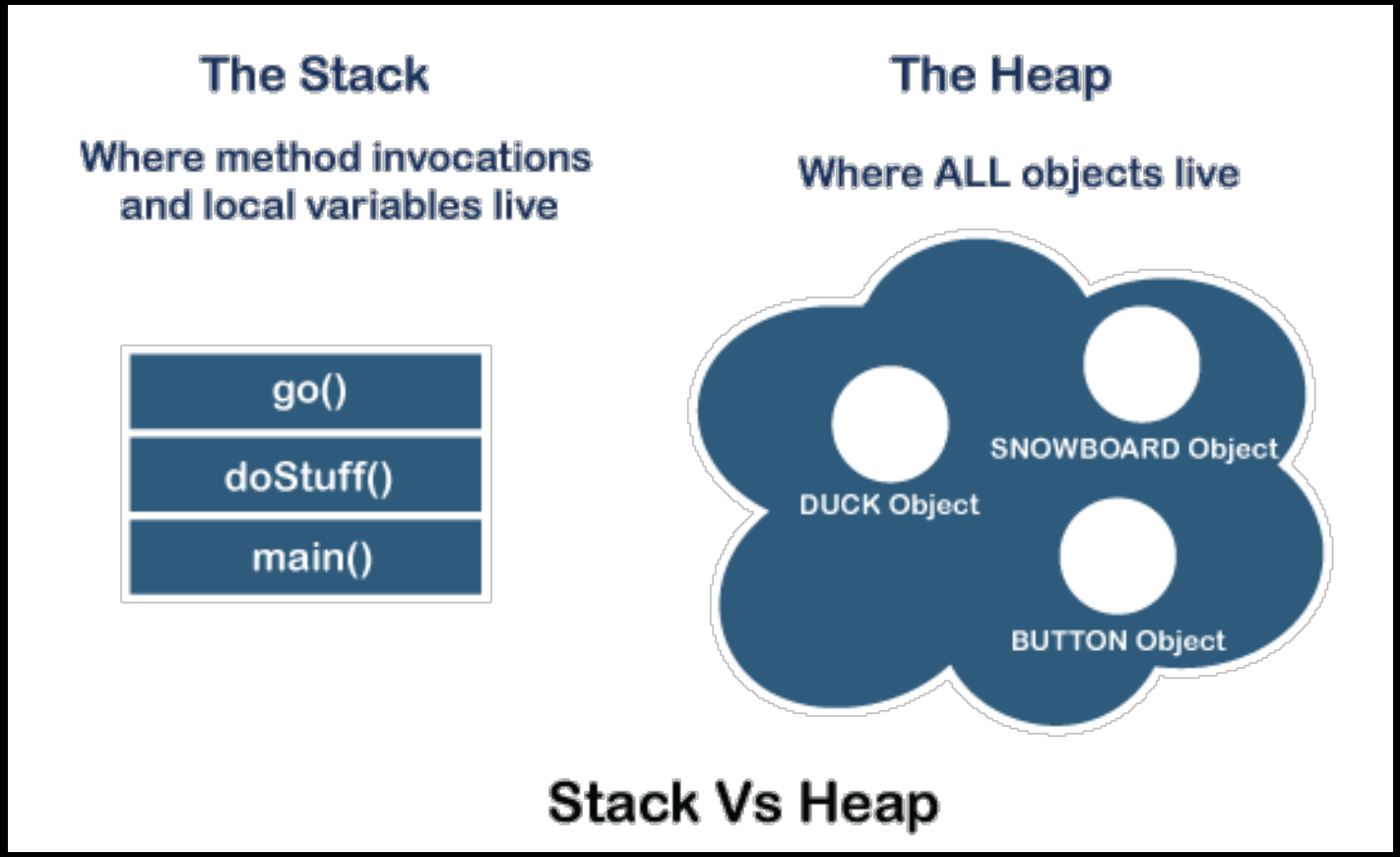
1. Scope: Code blocks {} determine the scope of variables.
2. Local Variables: Variables inside a block are not accessible outside it.
3. Initialization Block: Blocks without static run each time an instance is created.
4. Static Block: Blocks with static run once when the class is loaded.

CHALLENGE

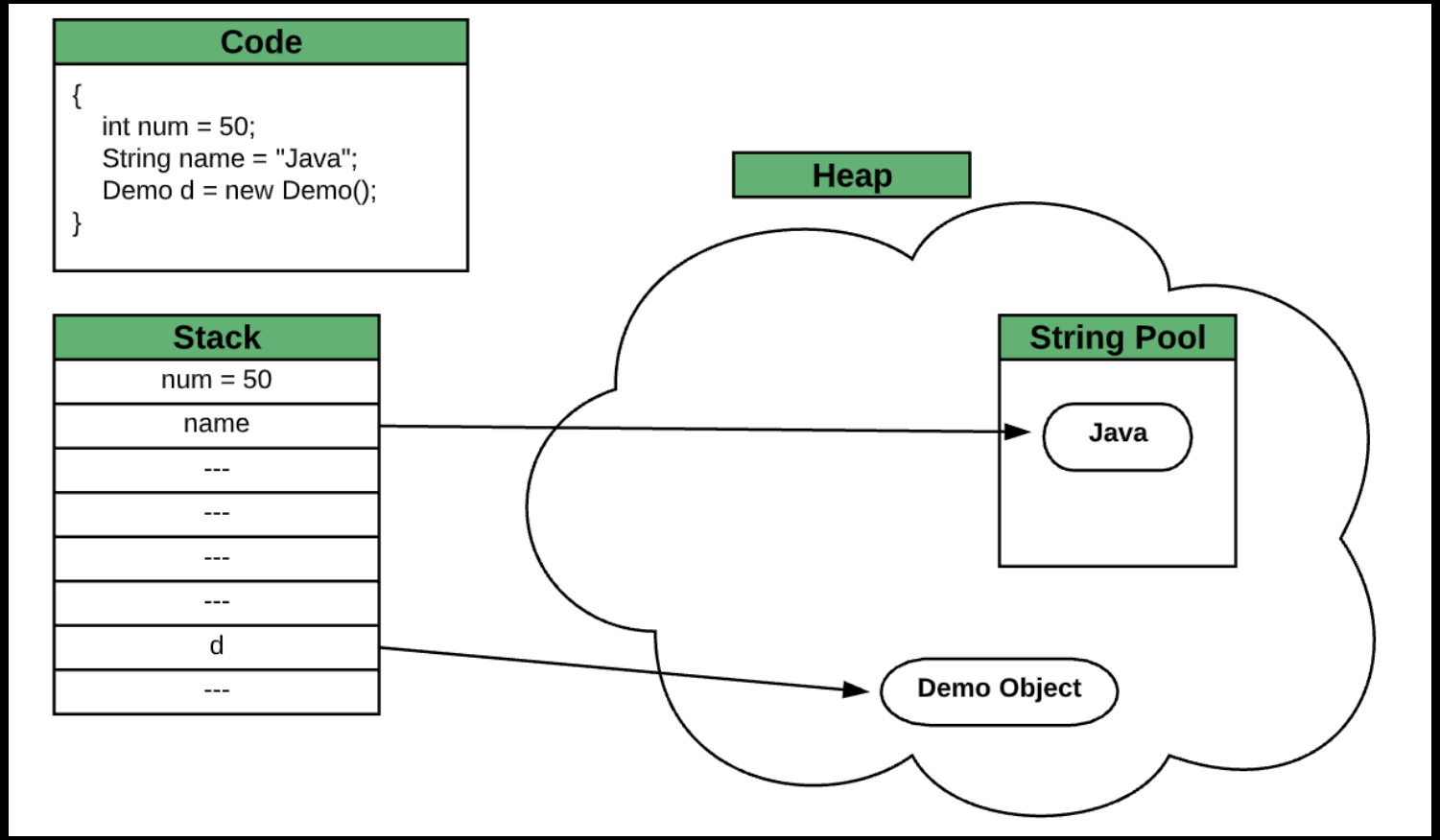
51. Create a Book class for a library system.
    - Instance variables: title, author, isbn.
    - Static variable: totalBooks, a counter for the total number of book instances.
    - Instance methods: borrowBook(), returnBook().
    - Static method: getTotalBooks(), to get the total number of books in the library.
52. Design a Course class.
    - Instance variables: courseName, enrolledStudents.
    - Static variable: maxCapacity, the maximum number of students for any course.
    - Instance methods: enrollStudent(String studentName), unenrollStudent(String studentName).
    - Static method: setMaxCapacity(int capacity), to set the maximum capacity for courses.

6.7 Stack vs Heap Memory

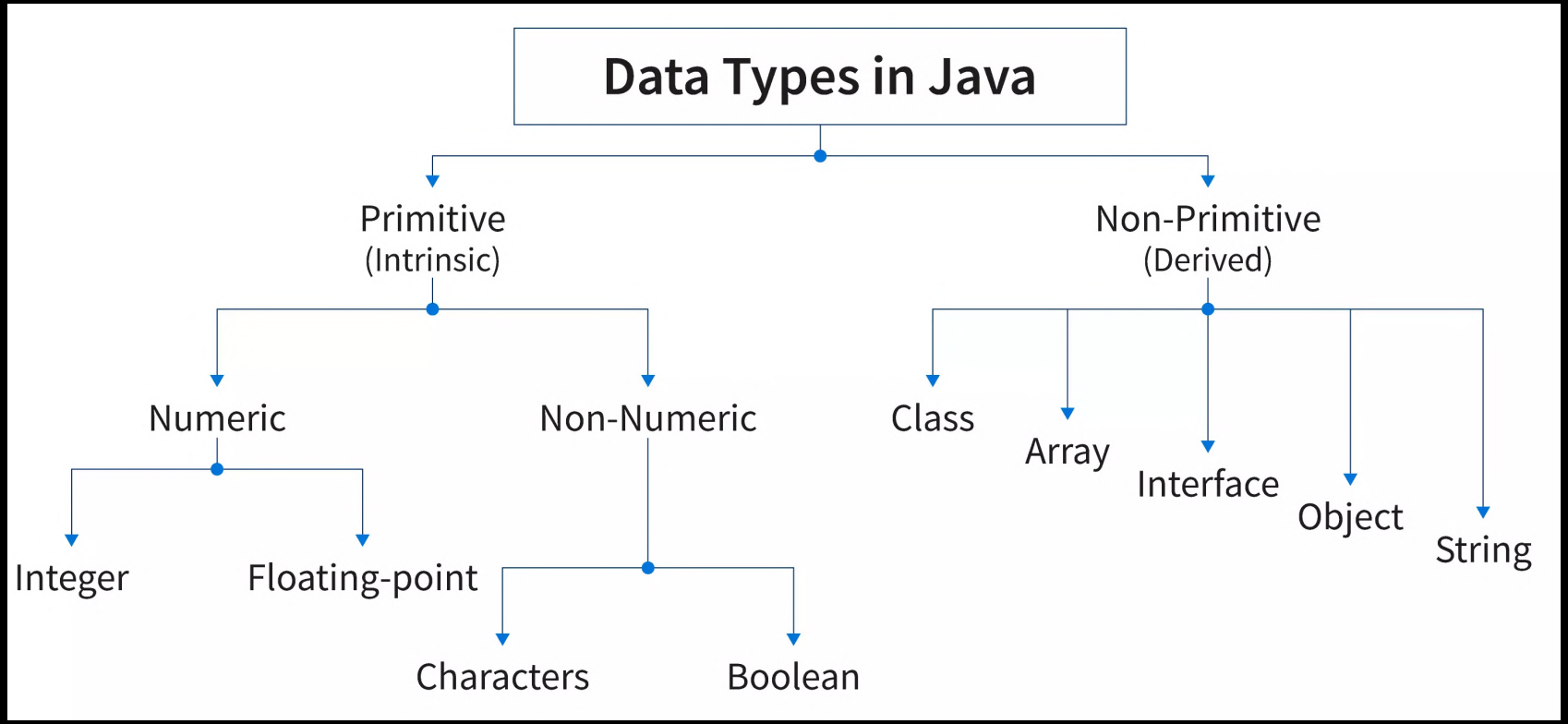
6.7 Stack vs Heap Memory



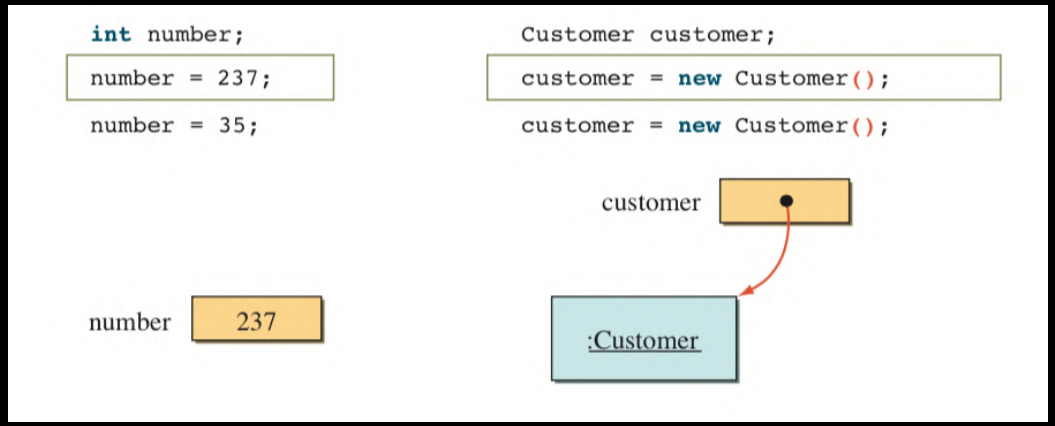
6.7 Stack vs Heap Memory



6.8 Primitive vs Reference Types



6.8 Primitive vs Reference Types



1. Memory: Primitives store actual values; reference types store addresses to objects.
2. Default Values: Primitives have specific defaults like 0 or false; reference types default to null.
3. Speed: Access to primitives is generally faster.
4. Storage Location: Primitives are stored in the stack; reference types are stored in the heap.
5. Comparison: Primitives compared by value; reference types compared by reference.

6.9 Variable Scopes

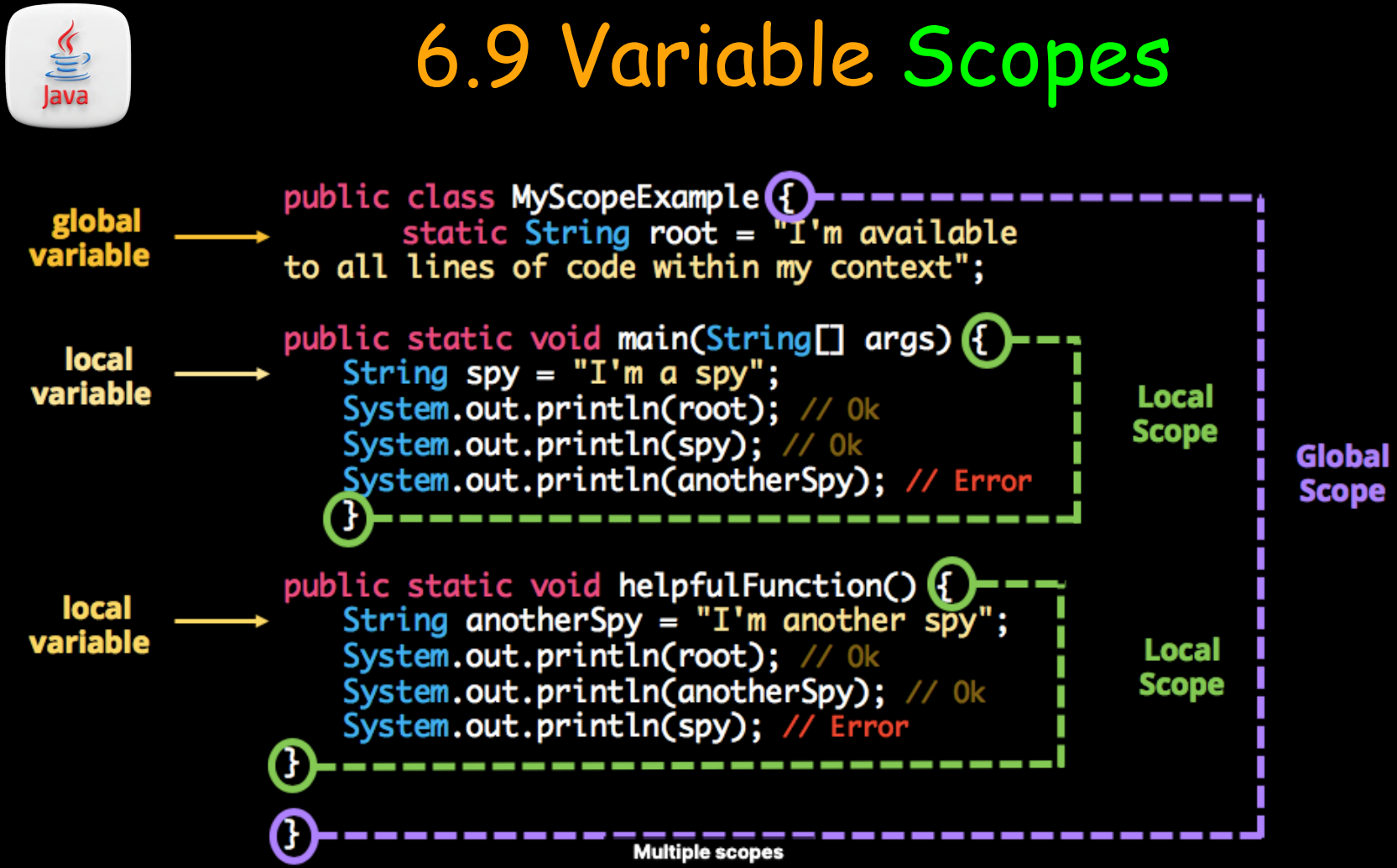
Instance vs Local Variables

```
class Athlete {
        public String   athleteName;
        public double   athleteSpeed;
        public int      athleteAge;

        public Athlete( name. speed, age ){
                this.athleteName = name;
                this.athleteSpeed = speed;
                this.atheleteAge = age;
        }
        public void athleteRun(){
                int speed = 100;
                System.out.println("Athlete runs at"+ speed +"Km/hr");
        }
}
```

Instance Variables

Local Variables

6.9 Variable Scopes

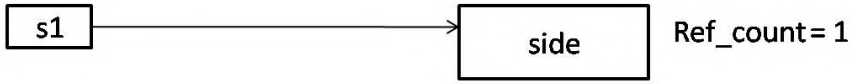


6.10 Garbage Collection & Finalize

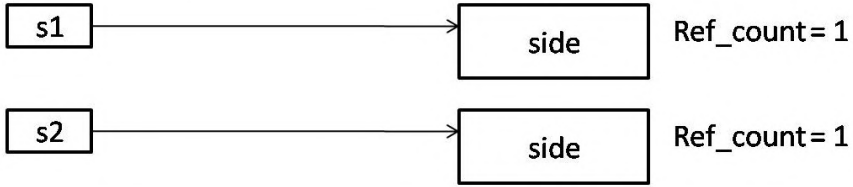
1. Automatic Process: Garbage collection is managed by the Java Virtual Machine (JVM), running in the background.
2. Object Eligibility: Objects that are no longer reachable, meaning no active references to them, are eligible for garbage collection.
3. No Manual Control: Unlike languages like C++, Java developers cannot explicitly deallocate memory. Garbage collection is automatic and non-deterministic.
4. Generational Collection: Java uses a generational garbage collection strategy, which divides memory into different regions (young, old, and permanent generations) based on object ages.
5. Heap Memory: Garbage collection occurs in the heap memory, where all Java objects reside.
6. Performance Impact: Garbage collection can affect application performance, particularly if it runs frequently or takes a long time to complete.

6.10 Garbage Collection & Finalize

Square s1 = new Square();

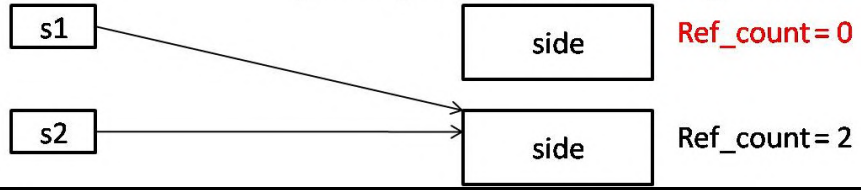


Square s2 = new Square();

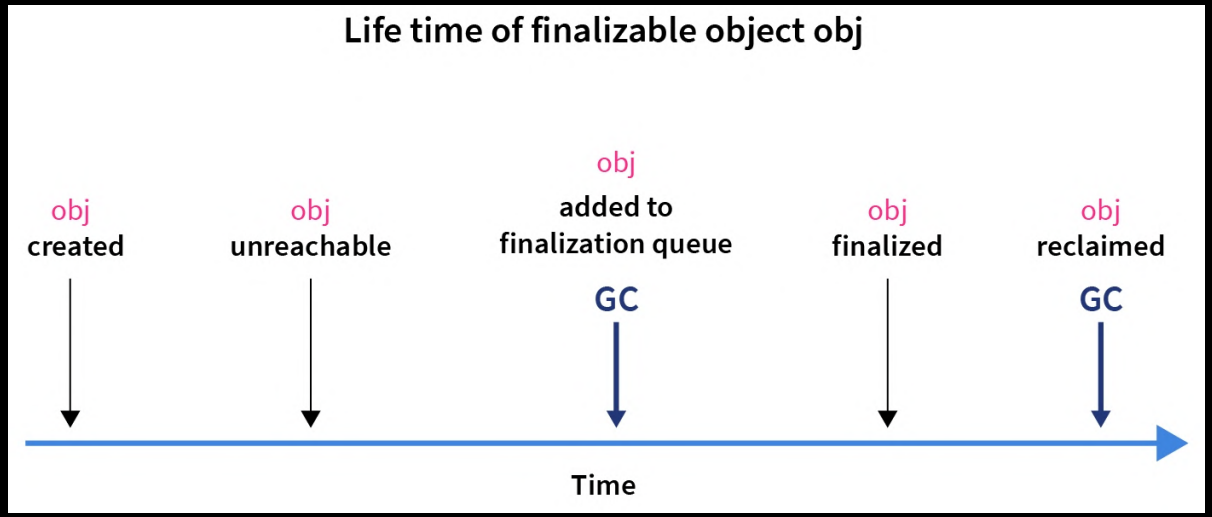


s1 = s2;

[This object is eligible for garbage collection]



6.10 Garbage Collection & Finalize



1. Finalization: Before an object is garbage collected, the finalize() method may be called, giving the object a chance to clean up resources. However, it's not guaranteed to run, and its usage is generally discouraged.
2. Optimization: Developers can optimize the process indirectly through code practices, like setting unnecessary object references to null.
3. System.gc() Call: While System.gc() suggests that the JVM performs garbage collection, it's not a guarantee.

Revision

1. Process vs Object Oriented
2. Instance Variables and Methods
3. Declaring and Using Objects
4. Class vs Object
5. This & Static Keyword
6. Constructors & Code Blocks
7. Stack vs Heap Memory
8. Primitive vs Reference Types
9. Variable Scopes
10. Garbage Collection & Finalize

Practice Exercise

Classes, Objects

Answer in True/False:

1. Object-oriented programming is mainly concerned with data rather than logic.
2. Instance methods in Java can be called without creating an object of the class.
3. A class in Java is a template that can be used to create objects.
4. Static methods belong to the class and can be called without an object.
5. The new keyword is used to declare an array in Java.
6. Variables declared within a method are called instance variables.
7. In Java, you can access instance variables through static methods directly.
8. Every object in Java has its own unique set of instance variables.
9. The static keyword in Java means that a particular member belongs to a type itself, rather than to instances of that type.
10. Variable scope in Java is determined at runtime.

Practice Exercise

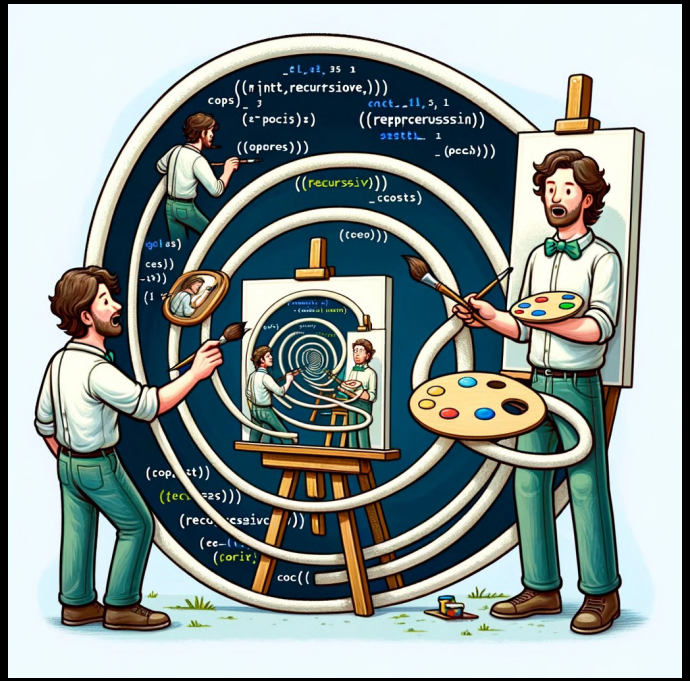
Classes, Objects

Answer in True/False:

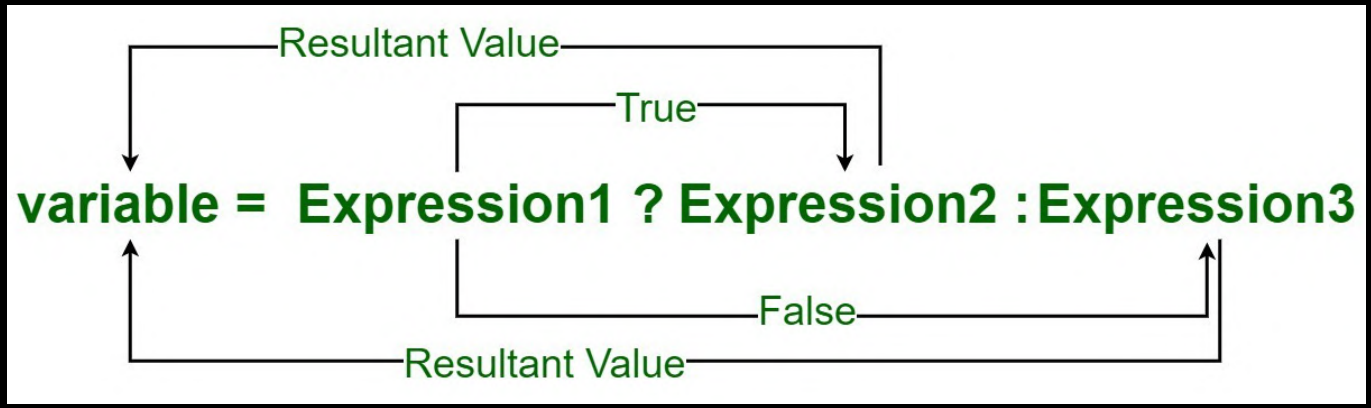
1. Object-oriented programming is mainly concerned with data rather than logic. — True
2. Instance methods in Java can be called without creating an object of the class. — False
3. A class in Java is a template that can be used to create objects. — True
4. Static methods belong to the class and can be called without an object. — True
5. The new keyword is used to declare an array in Java. — True
6. Variables declared within a method are called instance variables. — False
7. In Java, you can access instance variables through static methods directly. — False
8. Every object in Java has its own unique set of instance variables. — True
9. The static keyword in Java means that a particular member belongs to a type itself, rather than to instances of that type. — True
10. Variable scope in Java is determined at runtime. — False

7. Control Statements, Math & String

1. Ternary operator
2. Switch
3. Loops (Do-while, For, For each)
4. Using break & continue
5. Recursion
6. Random Numbers & Math class
7. Don't Learn Syntax
8. toString Method
9. String class
10. StringBuffer vs StringBuilder
11. Final keyword



7.1 Ternary operator



1. Syntax: condition ? expression1 : expression2
2. Condition: Boolean expression, evaluates to true or false.
3. Expressions: Both expressions must return compatible types.
4. Use Case: Suitable for simple conditional assignments.
5. Readability: Good for simple conditions, but can reduce clarity if overused.

7.2 Switch

```
switch (day) {
    case 1: System.out.println("Monday");
            break;
    case 2: System.out.println("Tuesday");
            break;
    case 3: System.out.println("Wednesday");
            break;
    case 4: System.out.println("Thursday");
            break;
    case 5: System.out.println("Friday");
            break;
    case 6: System.out.println("Saturday");
            break;
    case 7: System.out.println("Sunday");
            break;
    default: System.out.println("Invalid day.");
}
```

1. Multiple Cases: Handles multiple values for an expression efficiently.
2. Supported Types: Accepts byte, short, char, int, String, enums, and from Java 14, long, float, double.
3. Case Labels: Each case ends with a colon (:) and is followed by code.
4. Break Statement: Typically used to prevent fall-through between cases.
5. Default Case: Executes if no case matches; optional and doesn't require break.
6. Type Safety: Case label types must match the switch expression's type.

7.2 Switch

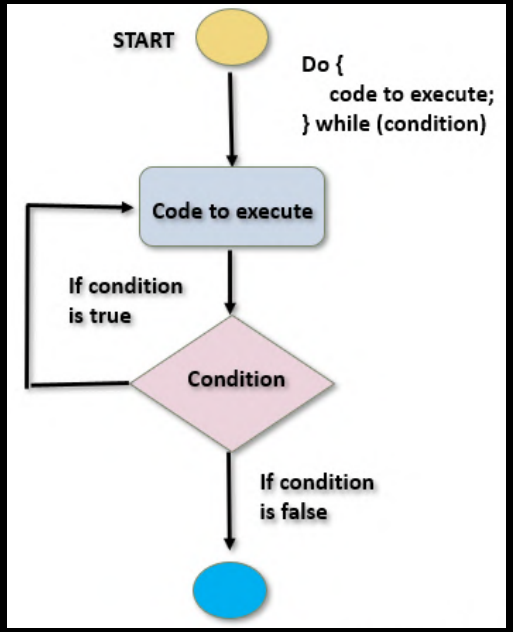
```
String output = switch (day) {
        case 1 -> "Monday";
        case 2 -> "Tuesday";
        case 3 -> "Wednesday";
        case 4 -> "Thursday";
        case 5 -> "Friday";
        case 6 -> "Saturday";
        case 7 -> "Sunday";
        default -> "Invalid";
};
System.out.println(output);
```

1. Enhanced Switch: Java 12 introduced enhancements like yield and multiple constants per case.
2. Switch Expression: From Java 14, switch can return a value using yield.

CHALLENGE

53. Create a program to find the minimum of two numbers.
54. Create a program to find if the given number is even or odd.
55. Create a program to calculate the absolute value of a given integer.
56. Create a program to Based on a student's score, categorize as "High", "Moderate", or "Low" using the ternary operator (e.g., High for scores > 80, Moderate for 50-80, Low for < 50).
57. Create a program to print the month of the year based on a number (1-12) input by the user.
58. Create a program to create a simple calculator that uses a switch statement to perform basic arithmetic operations like addition, subtraction, multiplication, and division.

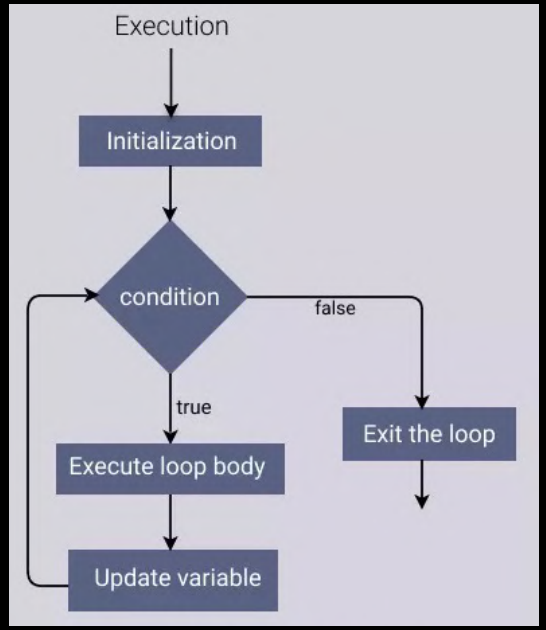
7.3 Loops (Do-while)




```do
do {
  // Body of the loop
}
while (condition);
```

1. Executes block first, then checks condition.
2. Guaranteed to run at least one iteration.
3. Unlike while, first iteration is unconditional.
4. Don't forget to update condition to avoid infinite loops.

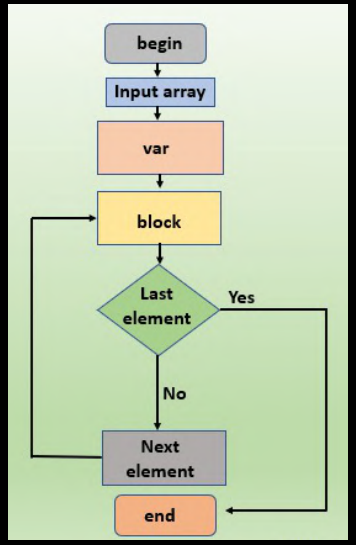
7.3 Loops (For)




```
for (initialisation; condition; update) {
  // Body of the loop
}
```

1. Standard loop for running code multiple times.
2. Generally preferred for counting iterations.

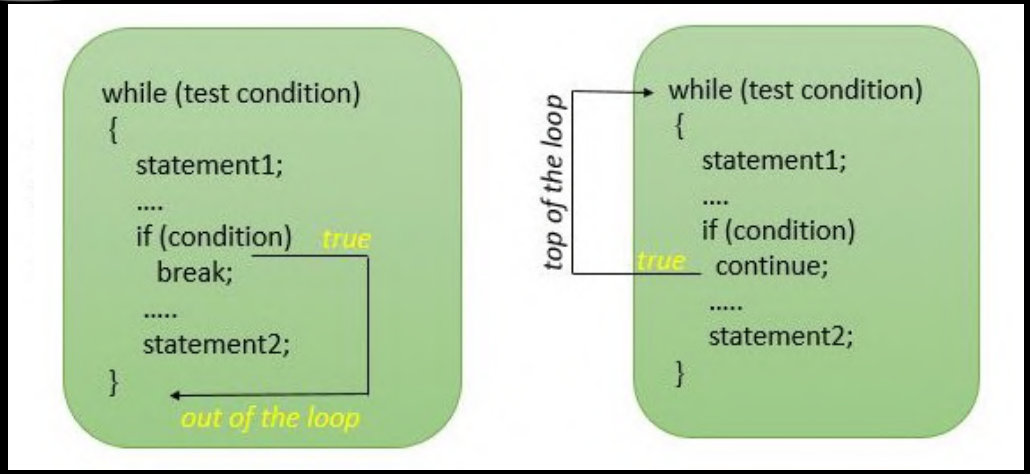
7.3 Loops (For each)




```java
String[] names = new String[] {
    "Ram", "Shyam", "Mohan", "Geeta",
    "Sita", "Sohan"
};
for (String name: names) {
    System.out.println(name);
}
```

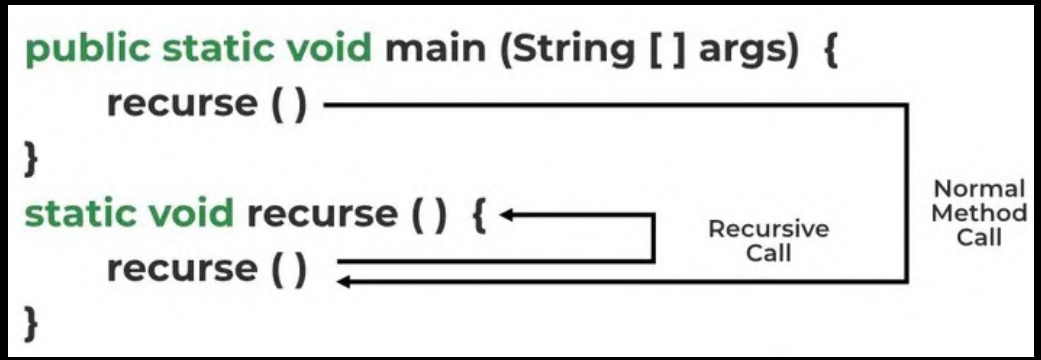
1. A method for array iteration, often preferred for readability.
2. Parameters: One for item, optional second for index.
3. Using return is similar to continue in traditional loops.
4. Not straightforward to break out of a forEach loop.
5. When you need to perform an action on each array element and don't need to break early.

7.4 Using break & continue



1. Break lets you stop a loop early, or break out of a loop
2. Continue is used to skip one iteration or the current iteration
3. In while loop remember to do the increment manually before using continue

7.5 Recursion

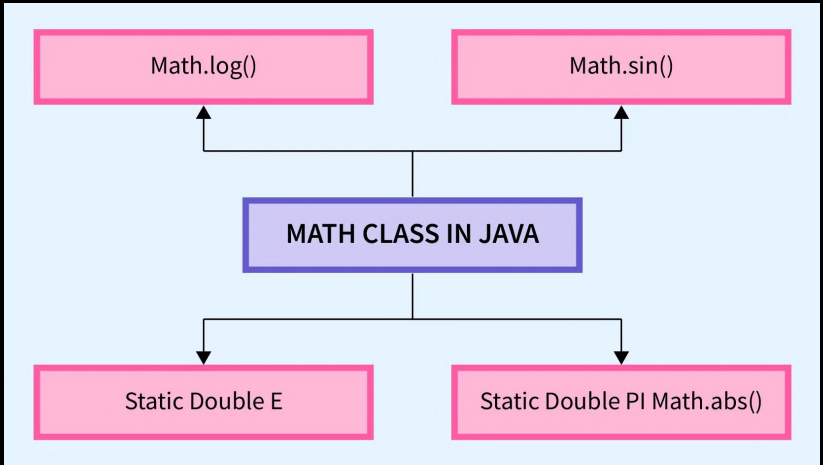


1. Self-Calling Function: Recursion is when a function calls itself.
2. Base Case: Essential to stop recursion and prevent infinite loops.
3. Recursive Case: The part where the function makes a recursive call.
4. Stack Overflow Risk: Excessive recursion can cause stack overflow errors.
5. Problem Solving: Ideal for problems divisible into similar, smaller problems.

CHALLENGE

59. Create a program using do-while to find password checker until a valid password is entered.
60. Create a program using do-while to implement a number guessing game.
61. Create a program using for loop multiplication table for a number.
62. Create a program using for to display if a number is prime or not.
63. Create a program using for-each to find the maximum value in an integer array.
64. Create a program using for-each to the occurrences of a specific element in an array.
65. Create a program using break to read inputs from the user in a loop and break the loop if a specific keyword (like "exit") is entered.
66. Create a program using continue to sum all positive numbers entered by the user; skip any negative numbers.
67. Create a program using continue to print only even numbers using continue for odd numbers.
68. Create a program using recursion to display the Fibonacci series upto a certain number.
69. Create a program using recursion to check if a string is a palindrome using recursion.

7.6 Random Numbers & Math class



Key Methods:

1. abs(): Absolute value.
2. ceil(): Rounds up.
3. floor(): Rounds down.
4. round(): Rounds to nearest integer.
5. max(), min(): Maximum and minimum of two numbers.
6. pow(): Power calculation.
7. sqrt(): Square root.
8. random(): Random number generation.
9. exp(), log(): Exponential and logarithmic functions.
10. Trigonometric functions: sin(), cos(), tan().

1. Static Class: Math methods are static and accessed directly.
2. Constants: Includes PI and E for π and the base of natural logarithms.

7.7 Don't Learn Syntax

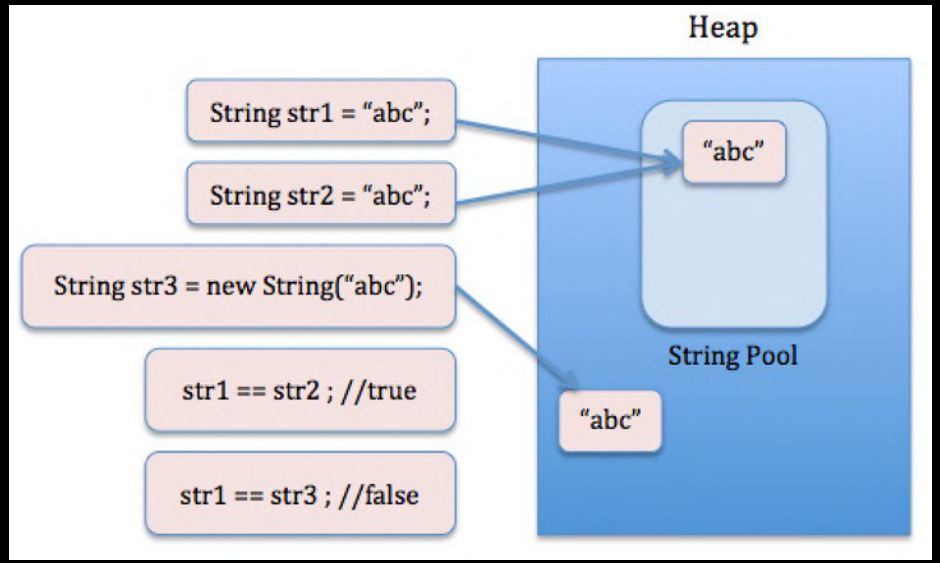
1. Google: Quick answers to coding problems.
2. Oracle: In-depth guides and documentation. https://docs.oracle.com/
3. ChatGPT: Real-time assistance for coding queries.
4. Focus: Understand concepts, not just syntax.

7.8 toString Method

```java
@Override
public String toString() {
    return "Car{" +
            "noOfWheels=" + noOfWheels +
            ", color='" + color + '\'' +
            ", maxSpeed=" + maxSpeed +
            ", currentFuelInLiters=" + currentFuelInLiters +
            ", noOfSeats=" + noOfSeats +
            '}';
}
```

1. Function: toString() provides a string representation of an object.
2. Inheritance: It's inherited from the Object class.
3. Default Format: By default, returns class name, "@", and hashcode.
4. Overriding: Commonly overridden in custom classes for meaningful output.
5. Implicit Call: Automatically called in string concatenation.

7.9 String class



1. Immutability: Once created, a String object's value cannot be changed. Modifications create new String objects.
2. String Pool: Java maintains a pool of strings for efficiency. When a new string is created, it's checked against the pool for a match to reuse.
3. Comparing: equals() method for value comparison, == operator checks reference equality.

7.9 String class

```
"Car{" +
 "noOfWheels=" + noOfWheels +
 ", color='" + color + '\'' +
 ", maxSpeed=" + maxSpeed +
 ", currentFuelInLiters=" + currentFuelInLiters +
 ", noOfSeats=" + noOfSeats +
 '}';
```

1. Concatenation: Strings can be concatenated using the + operator, but each concatenation creates a new String.
2. Methods: Provides methods like length(), substring(), equals(), compareTo(), indexOf(), for various operations.
3. Memory: Being immutable, strings can use more memory when frequently modified.

7.9 String class

Printf specifier	Data type
%s	String of text
%f	floating point value (float or double)
%e	Exponential, scientific notation of a float or double
%b	boolean true or false value
%c	Single character char
%d	Base 10 integer, such as a Java int, long, short or byte
%o	Octal number
%x	Hexadecimal number
%%	Percentage sign
%n	New line, aka carriage-return
%tY	Year to four digits
%tT	Time in format of HH:MM:SS (ie 21:46:30)

printf flag	Purpose
–	Aligns the formatted *printf* output to the left
+	The output includes a negative or positive sign
(	Places negative numbers in parenthesis
0	The formatted *printf* output is zero padded
,	The formatted output includes grouping separators
<space>	A blank space adds a minus sign for negative numbers and a leading space when positive

```
% [flags] [width] [.precision] specifier-character
```

7.9 String class

Pattern	Data	*Printf* Output
`'%s'`	`Java`	`'Java'`
`'%15s'`	`Java`	`'          Java'`
`'%-15s '`	`Java`	`'Java           '`
`'%-15S '`	`Java`	`'JAVA           '`

7.9 String class

Pattern	Data	*Printf* output
'%d'	123,457,890	'123457890'
'%,15d'	123,457,890	' 123,457,890'
'%+,15d'	123457890	' +123,457,890'
'%-+,15d'	123457890	'+123,457,890 '
'%0,15d'	123457890	'0000123,457,890'

7.10 StringBuffer vs StringBuilder

```
StringBuilder sentence = new StringBuilder("This is a sentence.");
sentence.append("Added word.");
System.out.println(sentence.toString()); //This is a sentence.Added word.
```

Parameter	String	StringBuilder	StringBuffer
mutability	immutable	mutable	mutable
Storage	String constant pool	heap	heap
Thread safety	Not used in the threaded environment as it is immutable	Not thread-safe so it is used in a single-threaded environment	Thread-safe so it is used in a multi-threaded environment
Speed	Comparably slowest	Comparably fastest	Faster than String but Slower than StringBuilder

7.11 Final keyword

```java
public class Main {
    public final String name = "John";

    public void setName(String name) {
        this.name = name; // cannot reassign final variables
    }
}
```

1. Variable: When applied to a variable, it becomes a constant, meaning its value cannot be changed once initialized.
2. Efficiency: Using final can lead to performance optimization, as the compiler can make certain assumptions about final elements.
3. Null Safety: A final variable must be initialized before the constructor completes, reducing null pointer errors.
4. Immutable Objects: Helps in creating immutable objects in combination with private fields and no setter methods.

CHALLENGE

70. Define a Student class with fields like name and age, and use toString to print student details.
71. Concatenate and Convert: Take two strings, concatenate them, and convert the result to uppercase.
72. Calculate the area and circumference of a circle for a given radius using Math.PI
73. Simulate a dice roll using Math.random() and display the outcome (1 to 6).
74. Create a number guessing game where the program selects a random number, and the user has to guess it.
75. Take an array of words and concatenate them into a single string using StringBuilder.
76. Create an object with final fields and a constructor to initialize them.

Revision

1. Ternary operator
2. Switch
3. Loops (Do-while, For, For each)
4. Using break & continue
5. Recursion
6. Random Numbers & Math class
7. Don't Learn Syntax
8. toString Method
9. String class
10. StringBuffer vs StringBuilder
11. Final keyword

Practice Exercise

Control Statements, Math & String

Answer in True/False:

1. The ternary operator in Java can replace certain types of if-else statements.
2. A switch statement in Java can only test for equality with constants.
3. The do-while loop will execute at least once even if the condition is false.
4. A for-each loop in Java is used to iterate over arrays.
5. The break statement skips the current iteration.
6. The continue statement exits the loop immediately, regardless of the condition.
7. In Java, recursion is a process where a method can call itself directly.
8. Random numbers generated by Math.random() are inclusive of 0 and 1.
9. The String class is mutable, meaning it can be changed after it's created.
10. StringBuilder is faster than StringBuffer since it is not synchronized.
11. A final variable in Java can be assigned a value once and only once.

Practice Exercise

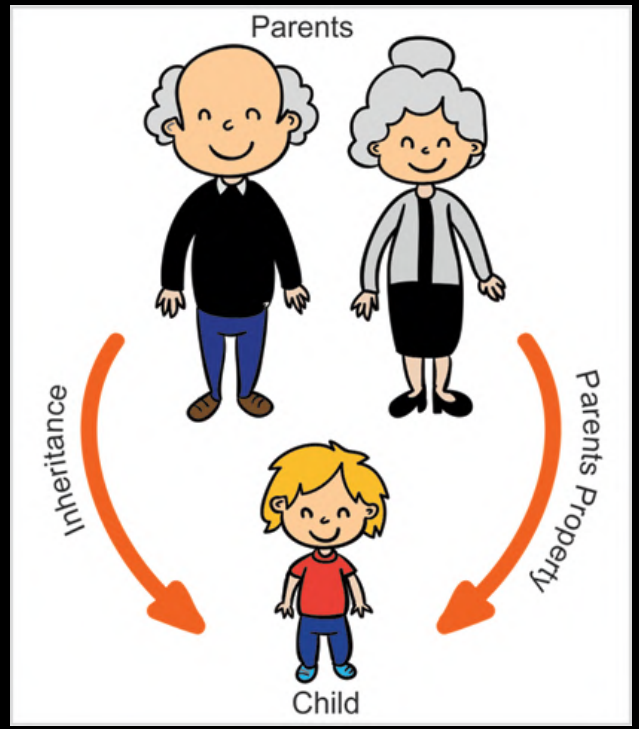
Control Statements, Math & String

Answer in True/False:

1. The ternary operator in Java can replace certain types of if-else statements. — True
2. A switch statement in Java can only test for equality with constants. — True
3. The do-while loop will execute at least once even if the condition is false. — True
4. A for-each loop in Java is used to iterate over arrays. — True
5. The break statement skips the current iteration. — False
6. The continue statement exits the loop immediately, regardless of the condition. — False
7. In Java, recursion is a process where a method can call itself directly. — True
8. Random numbers generated by Math.random() are inclusive of 0 and 1. — False
9. The String class is mutable, meaning it can be changed after it's created. — False
10. StringBuilder is faster than StringBuffer since it is not synchronized. — True
11. A final variable in Java can be assigned a value once and only once. — True

8 Encapsulation & Inheritance

1. Intro to OOPs Principle
2. What is Encapsulation
3. Import & Packages
4. Access Modifiers
5. Getter and Setter
6. What is Inheritance
7. Types of Inheritance
8. Object class
9. Equals and Hash Code
10. Nested and Inner Classes

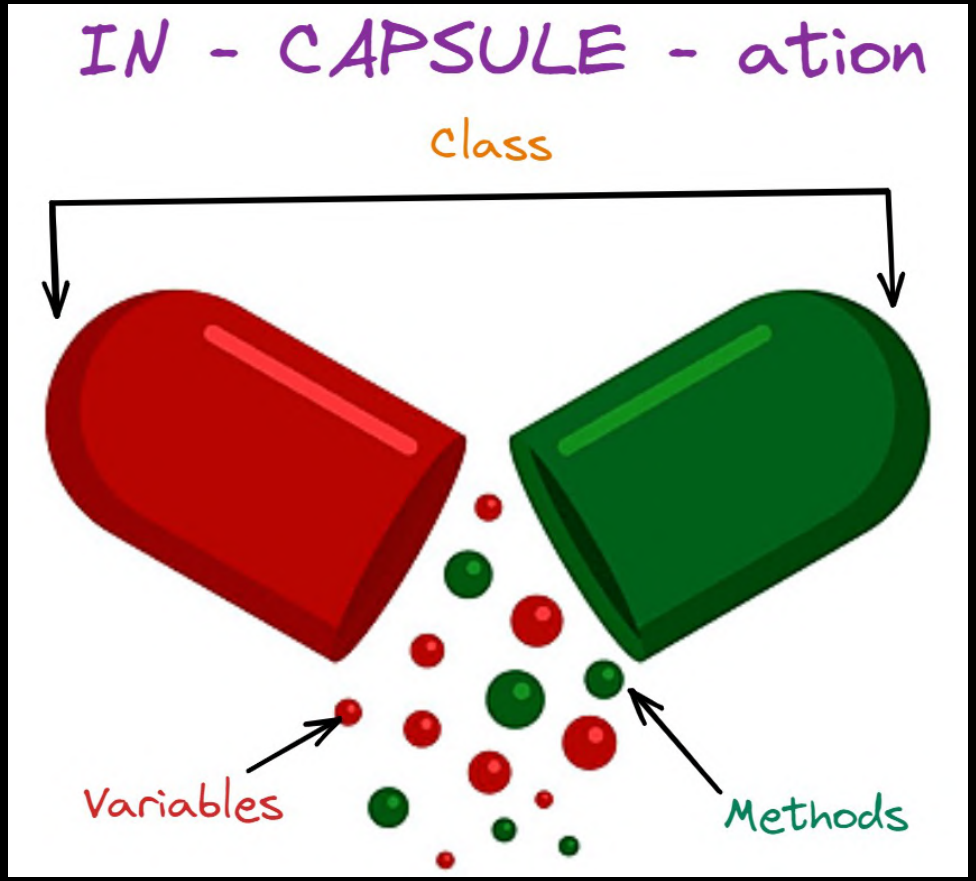


8.1 Intro to OOPs Principle

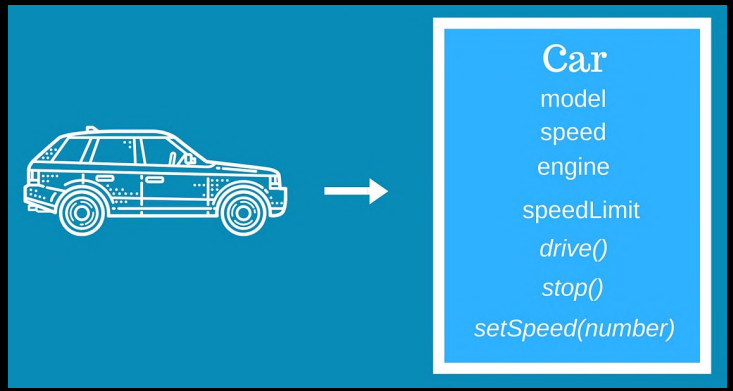
OOP Principles

Encapsulation	Abstraction	Inheritance	Polymorphism
When an object only exposes the selected information.	Hides complex details to reduce complexity.	Entities can inherit attributes from other entities.	Entities can have more than one form.

8.2 What is Encapsulation



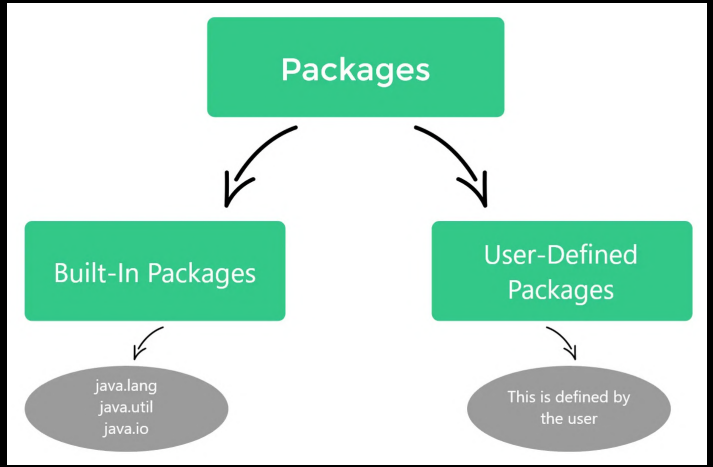
8.2 What is Encapsulation



1. Data Hiding: Encapsulation hides internal data, allowing access only through methods.
2. Access Modifiers: Uses private, public, protected to control access to class members.
3. Getter/Setter: Provides public methods for controlled property access.
4. Maintains Integrity: Protects object state from external interference.
5. Enhances Modularity: Keeps classes separate and reduces coupling.

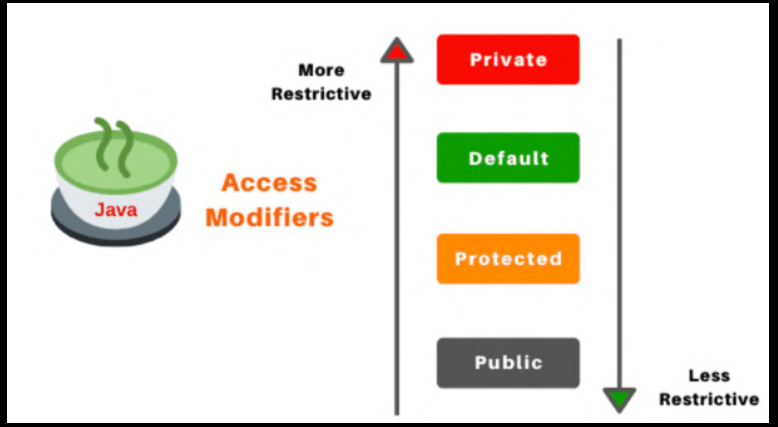
8.3 Import & Packages

1. Package Definition: A package in Java is a namespace that organizes classes and interfaces, preventing naming conflicts.
2. Package Declaration: Packages are declared at the beginning of a Java source file using the package keyword followed by the package name.
3. Import Statement: An import statement in Java is used to bring in classes or interfaces from other packages into the current file, making them accessible without using a fully qualified name.
4. Types of Import:
   - Single-Type Import: Imports a single class or interface from a package (e.g., import java.util.List;).
   - On-Demand Import: Imports all classes and interfaces from a package (e.g., import java.util.*;).
5. Avoiding Collisions: Packages help in avoiding name collisions by categorizing similar classes together.
6. Built-in Packages: Java comes with built-in packages like java.lang (automatically imported), java.util, java.io, etc.



8.4 Access Modifiers

1. Types: The four access modifiers are public, protected, default (no modifier), and private.
2. Public: The public modifier allows access from any other class. For classes, it means they can be accessed from any other package.
3. Protected: The protected modifier allows access within the same package and subclasses.
4. Default: If no access modifier is specified, it's the default, allowing access only within the same package.
5. Private: The private modifier restricts access to the defining class only.
6. Class-Level Access: Only public and default (no modifier) access modifiers are applicable for top-level classes.
7. Member-Level Access: Methods, constructors, and variables can use all four access modifiers to control visibility.



8.4 Access Modifiers

Access Specifiers in Java

		public	private	protected	default
Same Package	Class	YES	YES	YES	YES
	Sub class	YES	NO	YES	YES
	Non sub class	YES	NO	YES	YES
Different Package	Sub class	YES	NO	YES	NO
	Non sub class	YES	NO	NO	NO

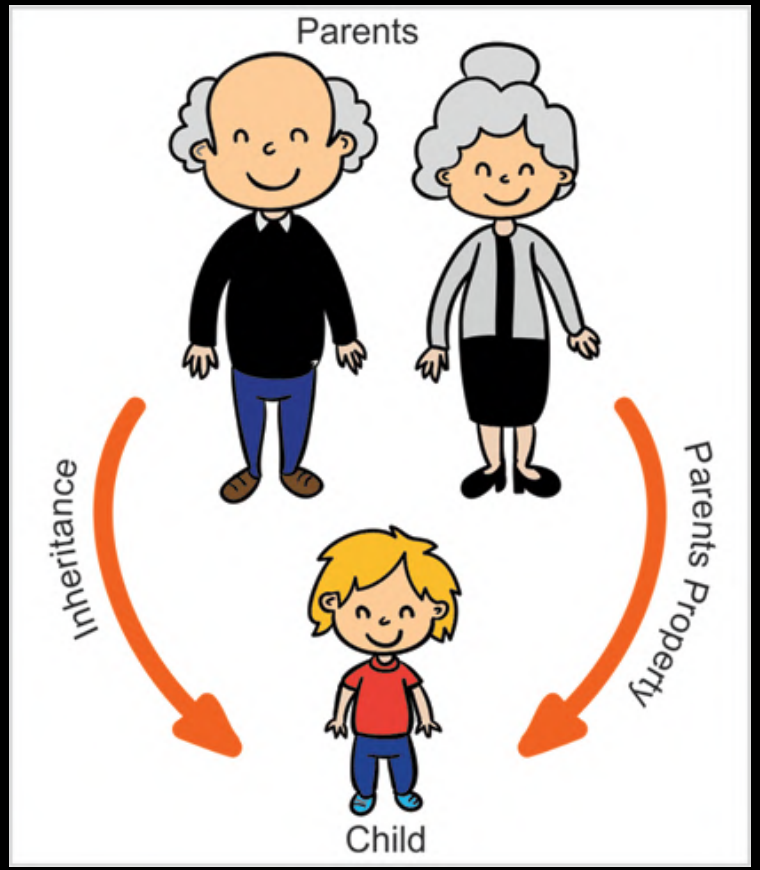
8.5 Getter and Setter

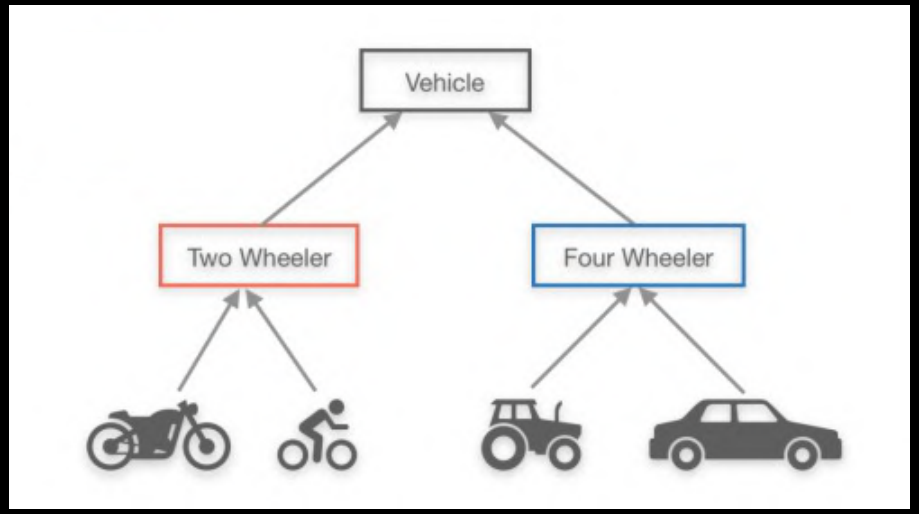
1. Getters: Retrieve private field values, typically named get<FieldName>.
2. Setters: Set or update private field values, usually named set<FieldName>.
3. Control and Validation: Offer controlled access and allow for validation logic.
4. Encapsulation: Facilitate read-only or write-only access to class fields.
5. Flexibility: Allow for internal changes without affecting external interfaces.

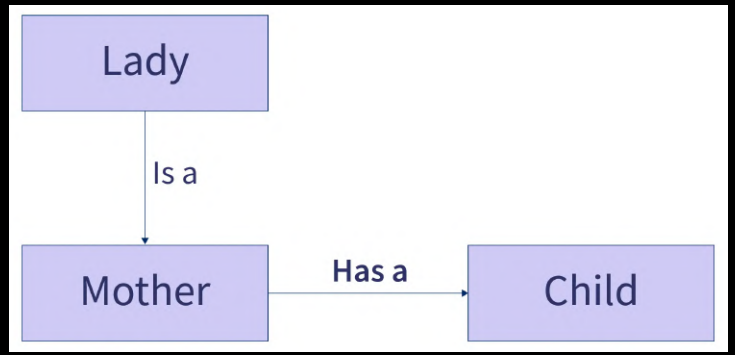
CHALLENGE

77. Create a simple application with at least two packages: com.example.geometry and com.example.utils. In the geometry package, define classes like Circle and Rectangle. In the utils package, create a Calculator class that can compute areas of these shapes.
78. Define a BankAccount class with private attributes like accountNumber, accountHolderName, and balance. Provide public methods to deposit and withdraw money, ensuring that these methods don't allow illegal operations like withdrawing more money than the current balance.
79. Define a class Employee with private attributes (like name, age, and salary), public methods to get and set these attributes, and a package-private method to displayEmployeeDetails. Create another class in the same package to test access to the displayEmployeeDetails method.

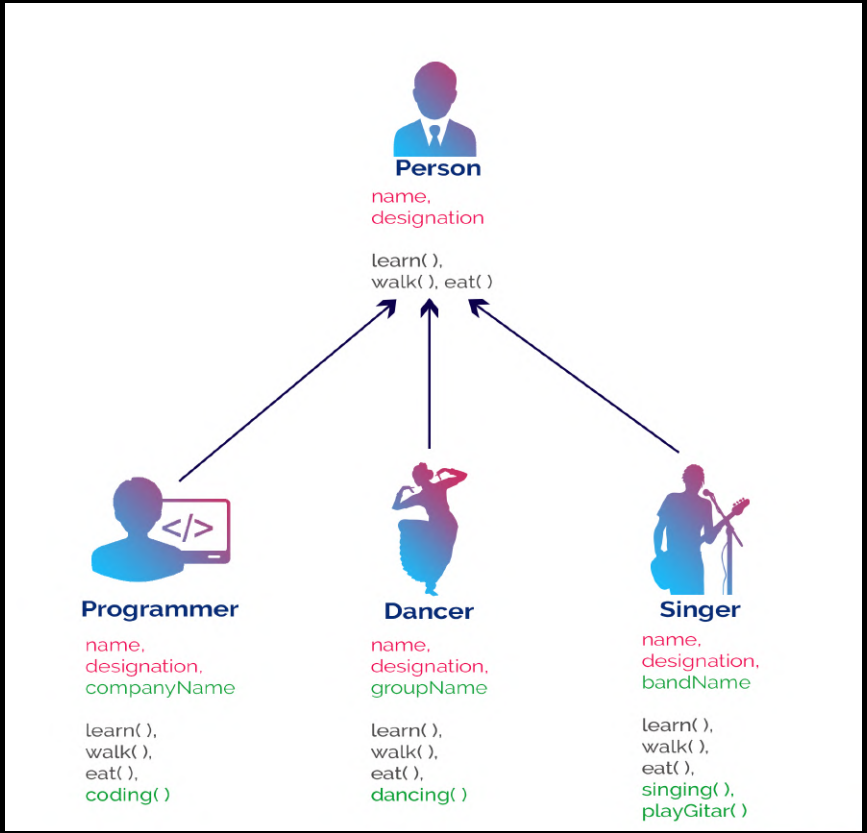
8.6 What is Inheritance





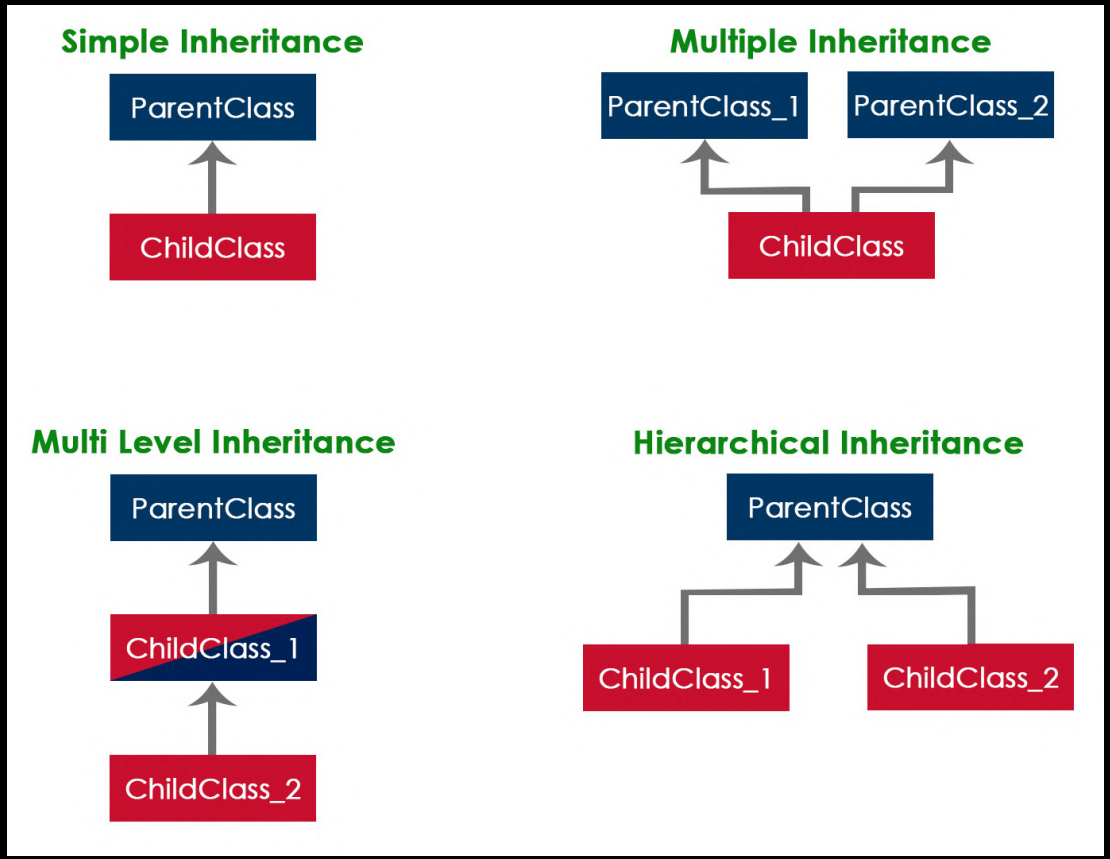


8.6 What is Inheritance

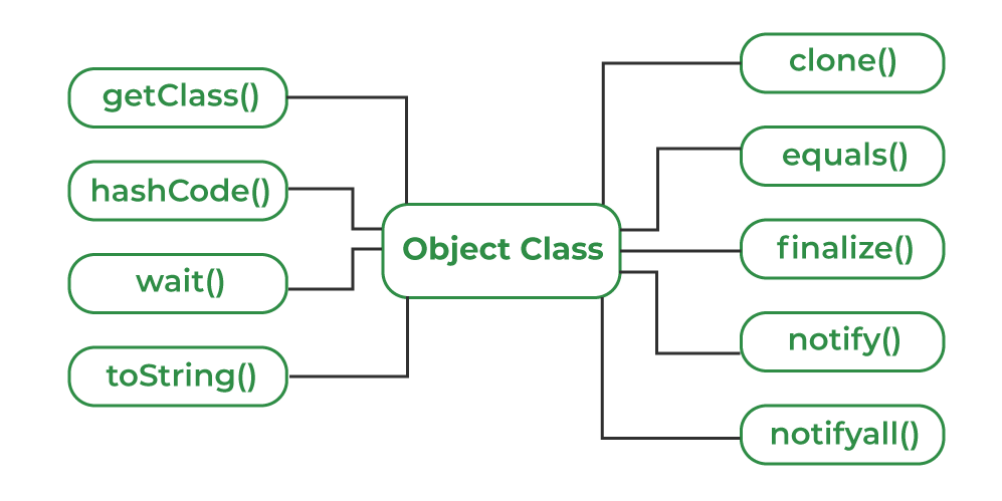


1. Inheritance allows a new class (subclass) to inherit features from an existing class (superclass).
2. Code Reusability: It enables subclasses to use methods and variables of the superclass, reducing code duplication.
3. Access Control: The protected access modifier is often used in inheritance to allow subclass access to superclass members.

8.7 Types of Inheritance

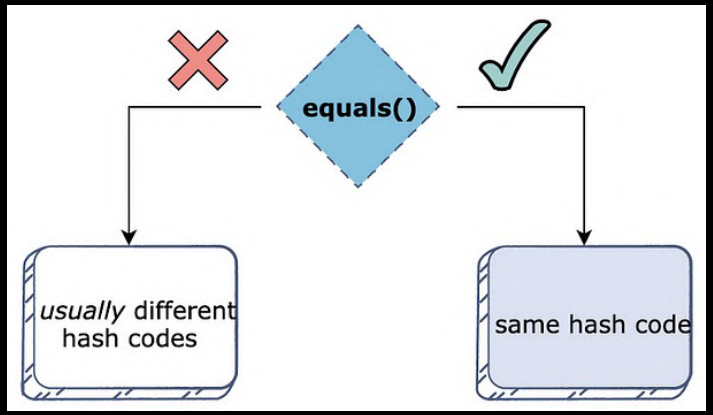


8.8 Object class



1. Root Class: The Object class is the parent class of all classes in Java, forming the top of the class hierarchy.
2. Default Methods: It provides fundamental methods like equals(), hashCode(), toString(), and getClass() that can be overridden by subclasses.
3. String Representation: The toString() method returns a string representation of the object, often overridden for more informative output.

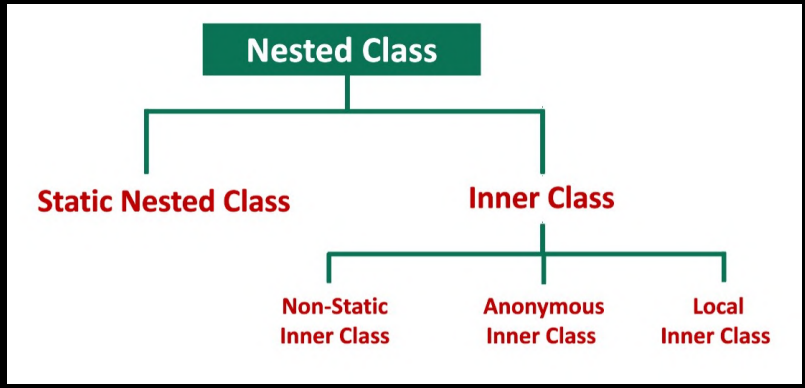
8.9 Equals and HashCode



1. equals() Method: Used for logical equality checks between objects. By default, it compares object references, but it's commonly overridden to compare object states.
2. hashCode() Method: Generates an integer hash code representing an object. It's crucial for the performance of hash-based collections like HashMap.
3. Equals-Hashcode Contract: If two objects are equal according to equals(), they must have the same hash code. However, two objects with the same hash code aren't necessarily equal.
4. Overriding Both: If equals() is overridden, hashCode() should also be overridden to maintain consistency between these methods.

8.10 Nested and Inner Classes

1. Nested Classes: Classes defined within another class, divided into static (static nested classes) and non-static (inner classes).
2. Static Nested Classes: Act as static members of the outer class; can access outer class's static members but not its non-static members.
3. Inner Classes: Associated with an instance of the outer class; can access all members of the outer class, including private ones.
4. Local and Anonymous Inner Classes: Local inner classes are defined within a block (like a method) and are not visible outside it. Anonymous inner classes are nameless and used for single-use implementations.
5. Use Cases: Useful for logically grouping classes, improving encapsulation, and enhancing code readability.



CHALLENGE

80. Start with a base class LibraryItem that includes common attributes like itemID, title, and author, and methods like checkout() and returnItem(). Create subclasses such as Book, Magazine, and DVD, each inheriting from LibraryItem. Add unique attributes to each subclass, like ISBN for Book, issueNumber for Magazine, and duration for DVD.

81. Create a class Person with attributes name and age. Override equals() to compare Person objects based on their attributes. Override hashCode() consistent with the definition of equals().

82. Create a class ArrayOperations with a static nested class Statistics. Statistics could have methods like mean(), median(), which operate on an array.

Revision

1. Intro to OOPs Principle
2. What is Encapsulation
3. Import & Packages
4. Access Modifiers
5. Getter and Setter
6. What is Inheritance
7. Types of Inheritance
8. Object class
9. Equals and Hash Code
10. Nested and Inner Classes

Practice Exercise

Encapsulation & Inheritance

Answer in True/False

1. Encapsulation is the OOP principle that ensures an object's data can be changed by any method, without restrictions.
2. The import statement in Java can be used to bring a single class or an entire package into visibility.
3. The private access modifier means that the member is only accessible within its own class.
4. Setters are used to retrieve the value of a private variable from outside the class.
5. Inheritance in OOP can be used to create a general class that defines traits to be inherited by more specific subclasses.
6. The Object class in Java has protected access by default.
7. If two objects are equal according to their equals() method, then their hashCode() methods must return different integers.
8. Inner classes are defined within the scope of a method and cannot exist independently of the method.
9. A static nested class can access the instance variables of its enclosing outer class.
10. Overriding the equals() method requires also overriding the hashCode() method to maintain consistency.

Practice Exercise

Encapsulation & Inheritance

Answer in True/False

1. Encapsulation is the OOP principle that ensures an object's data can be changed by any method, without restrictions. — False
2. The import statement in Java can be used to bring a single class or an entire package into visibility. — True
3. The private access modifier means that the member is only accessible within its own class. — True
4. Setters are used to retrieve the value of a private variable from outside the class. — False
5. Inheritance in OOP can be used to create a general class that defines traits to be inherited by more specific subclasses. — True
6. The Object class in Java has protected access by default. — False
7. If two objects are equal according to their equals() method, then their hashCode() methods must return different integers. — False
8. Inner classes are defined within the scope of a method and cannot exist independently of the method. — False
9. A static nested class can access the instance variables of its enclosing outer class. — False
10. Overriding the equals() method requires also overriding the hashCode() method to maintain consistency. — True

KG Coding

Some Other One shot Video Links:

- Complete HTML
- Complete CSS
- Complete JavaScript
- Complete React and Redux
- One shot University Exam Series

http://www.kgcoding.in/

Our YouTube Channels

KG Coding Android App

KG Coding

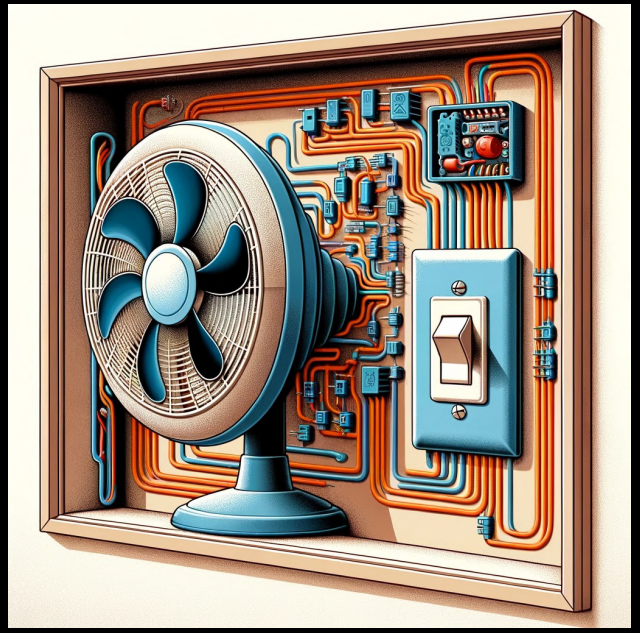
Knowledge GATE

KG Placement Prep

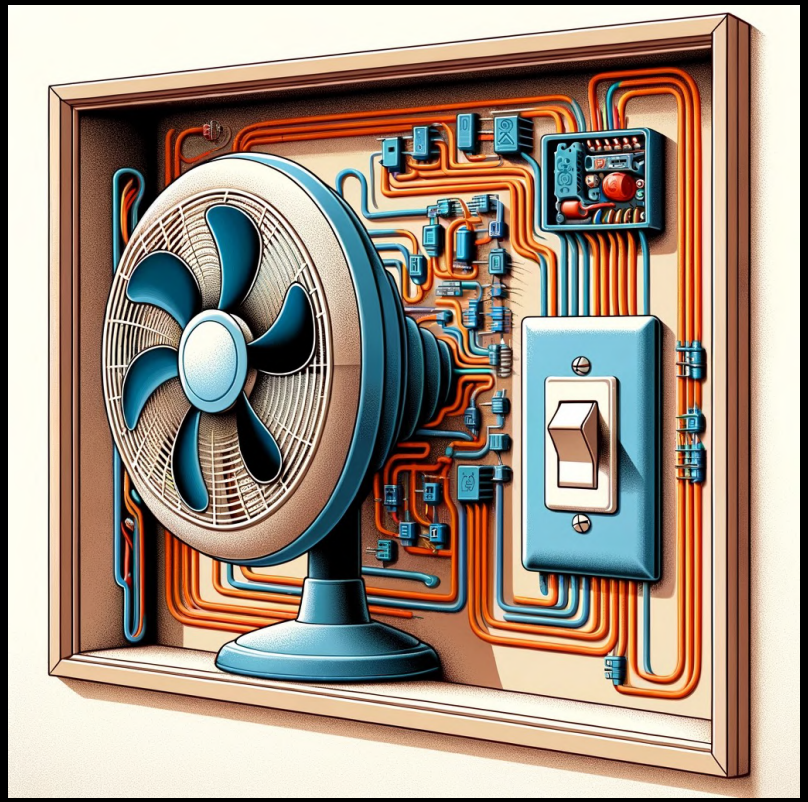
Sanchit Socket

9 Abstraction and Polymorphism

1. What is Abstraction
2. Abstract Keyword
3. Interfaces
4. What is Polymorphism
5. References and Objects
6. Method / Constructor Overloading
7. Super Keyword
8. Method / Constructor Overriding
9. Final keyword revisited
10. Pass by Value vs Pass by reference.

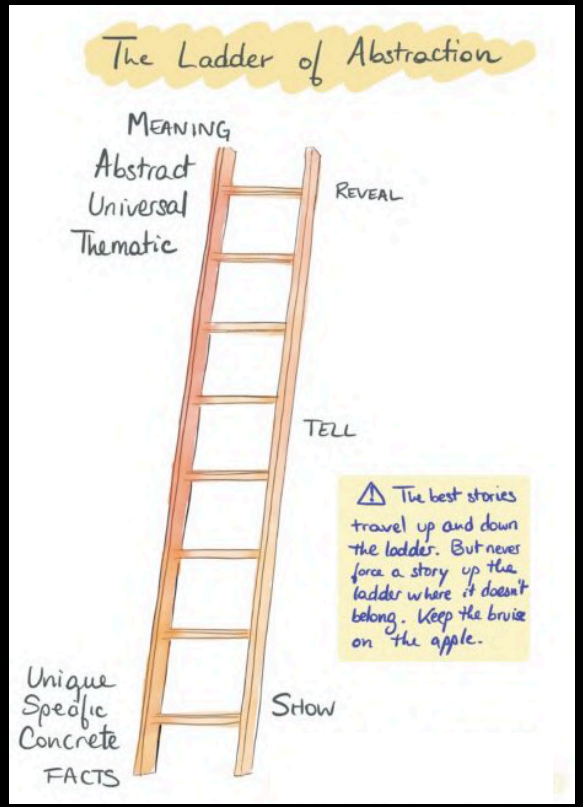


9.1 What is Abstraction



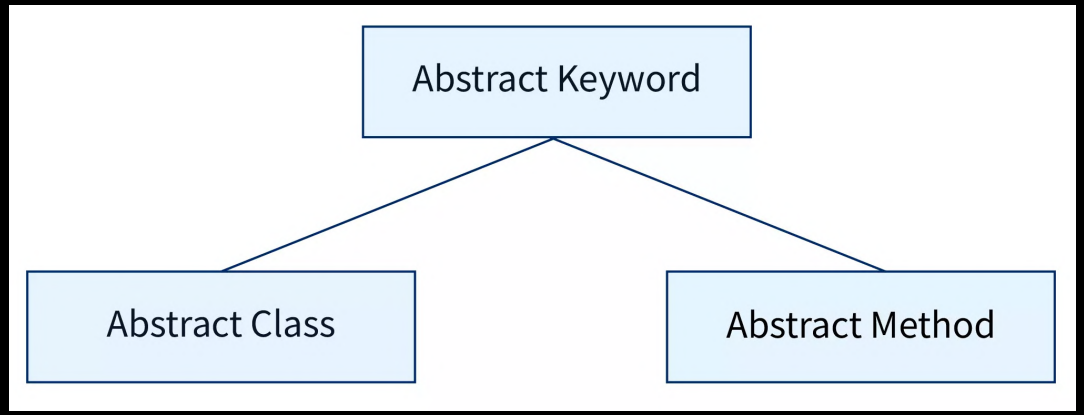
1. Core Principle: Abstraction hides complex implementation details, focusing only on essential features.
2. Focus on Functionality: Emphasizes what an object does, not how it does it, through clear interfaces.

9.1 What is Abstraction



1. Simplifies Complexity: It reduces complexity by showing only relevant information in class design.
2. Real-World Modelling: Abstraction allows creating objects that represent real-life entities with key attributes and behaviours.

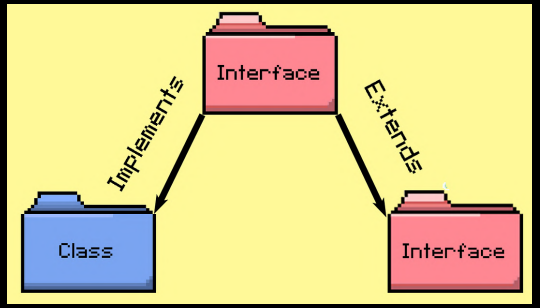
9.2 Abstract Keyword



1. Abstract Class: Used to declare non-instantiable abstract classes that serve as base classes.
2. Abstract Method: Defines methods without implementations, requiring subclasses to provide specific functionality.
3. Mandatory Implementation: Subclasses must implement all abstract methods of an abstract class.
4. Design Flexibility: Allows for flexible class design by defining a contract for subclasses.

9.3 Interfaces

```java
interface Pet {
    public void play();
}

class Dog extends Animal implements Pet{
    // Dog has its own implementation of run method
    public void run() { }

    public void bark() { }

    // Define the methods of the interface
    public void play() { }
}
```



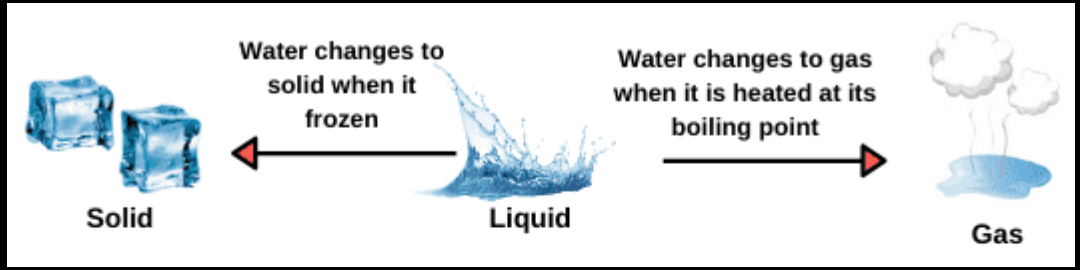
1. Interfaces primarily declare abstract methods for implementation by classes.
2. A class can implement multiple interfaces, allowing for more flexible designs.
3. Interfaces can have default methods with implementation and static methods.
4. Interface methods are inherently public and abstract, except for default and static methods.

CHALLENGE

83. Create an abstract class Shape with an abstract method calculateArea(). Implement two subclasses: Circle and Square. Each subclass should have relevant attributes (like radius for Circle, side for Square) and their own implementation of the calculateArea() method.

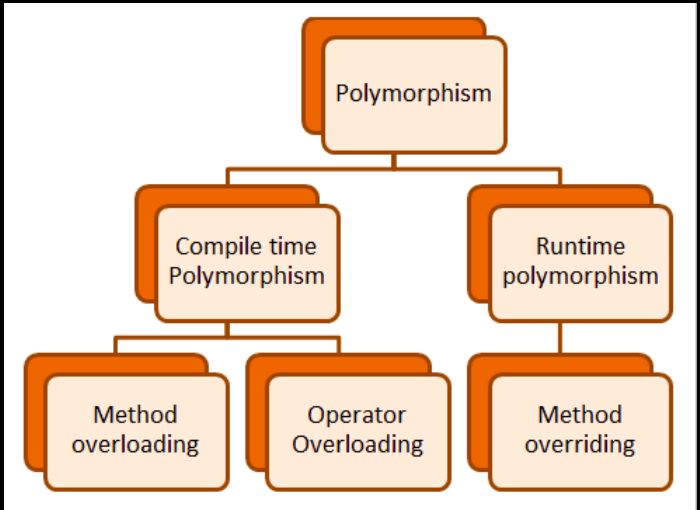
84. Create an interface Flyable with an abstract method fly(). Create an abstract class Bird that implements Flyable. Implement a subclass Eagle that extends Bird. Provide an implementation for the fly() method.

9.4 What is Polymorphism



1. Polymorphism is the ability of objects of different classes to respond to the same message (method call) in different ways.
2. Flexibility: Allows for writing more flexible and reusable code.
3. Simplicity: Enables developers to write more simple and readable code by using the same interface for different underlying data types.

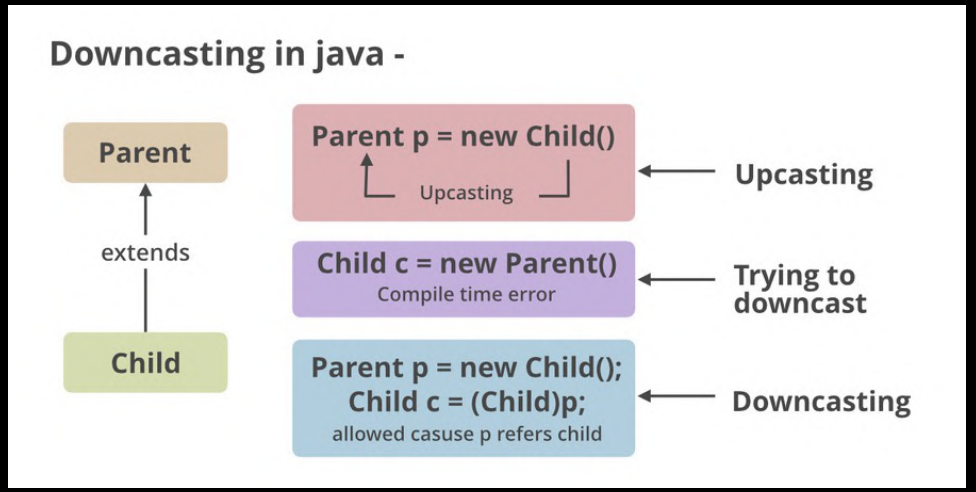
9.4 What is Polymorphism



1. Compile-Time Polymorphism: Achieved through method overloading or operator overloading.
2. Run-Time Polymorphism: Achieved through method overriding, where a subclass provides a specific implementation of a method already defined in its superclass.

9.5 References and Objects

1. Upcasting:
   - Converts subclass to superclass reference.
   - Automatic and safe.
   - Access only to superclass methods.
2. Downcasting:
   - Converts superclass to subclass reference.
   - Manual and risky, needs instanceof check.
   - Access to subclass-specific methods.
3. Usage:
   - Upcasting for generalization in methods.
   - Downcasting for specific subclass behaviors.



9.6 Method / Constructor Overloading

```java
// add() method is overloaded
public static int add(int a,int b) {
    return a + b; //adds integers
}

public static String add(String s1,String s2) {
    return s1.concat(s2); //concats strings
}
```

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

Same Method Name, Different Parameter

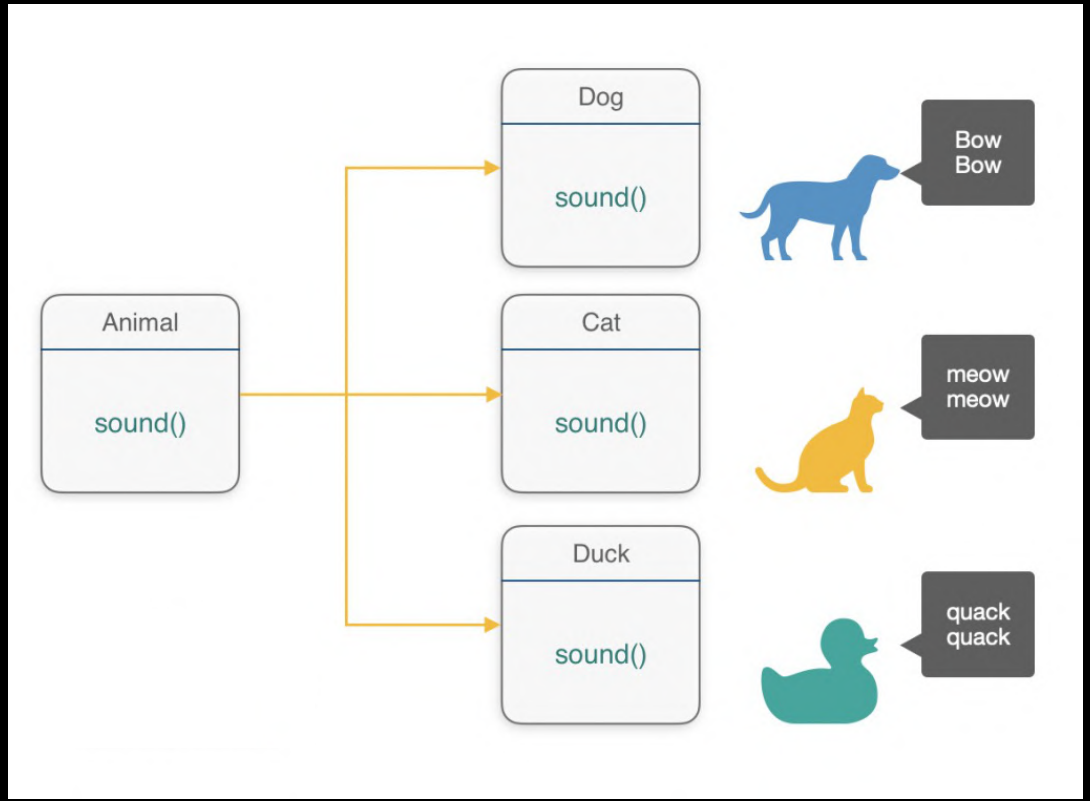
1. Method overloading occurs when multiple methods in the same class have the same name but different parameter lists.
2. Parameter Difference: Overloaded methods must differ in the number, type, or sequence of their parameters.
3. Return Type: Can vary between overloaded methods, but the return type alone does not distinguish them.
4. Compile-Time Polymorphism: It's a form of polymorphism that is resolved during compile time.

9.7 Super Keyword

1. Super can be used to refer immediate parent class instance varible.
2. Super can be used to invoke immediate parent class method.
3. Super () can be used to invoke immediate parent class constructor.

9.8 Method / Constructor Overriding

1. Method overriding occurs when a subclass provides a specific implementation for a method already defined in its superclass.
2. Run-Time Polymorphism: Overriding is a basis for runtime polymorphism, where the method call is determined by the object's type at runtime.
3. Superclass Reference: An overridden method can be called through a superclass reference holding a subclass object.



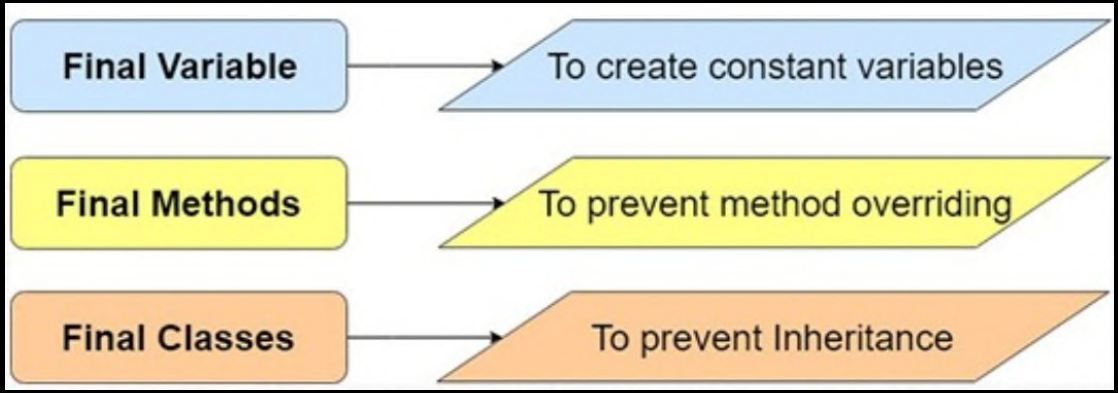
9.8 Method / Constructor Overriding

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

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

Same Method Name,
Same parameter

1. Same Signature: Overridden methods must have the same name, return type, and parameters as the method in the parent class.
2. Access Level: The access level cannot be more restrictive than the overridden method's access level.
3. **@Override Annotation**: This annotation is optional but helps to ensure that the method is correctly overridden.

9.9 Final keyword revisited



1. Variable becomes a constant, meaning its value cannot be changed once initialized.
2. Method: A final method cannot be overridden by subclasses.
3. Class: A final class cannot be subclassed, securing the class from being extended.
4. Efficiency: Using final can lead to performance optimization, as the compiler can make certain assumptions about final elements.
5. Null Safety: A final variable must be initialized before the constructor completes, reducing null pointer errors.
6. Immutable Objects: Helps in creating immutable objects in combination with private fields and no setter methods.

9.10 Pass by Value vs Pass by reference

pass by reference

cup =

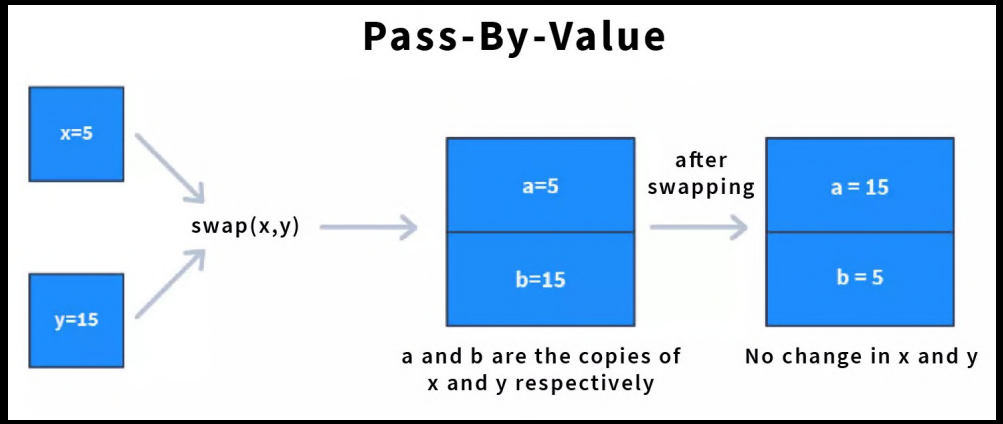
fillCup( )

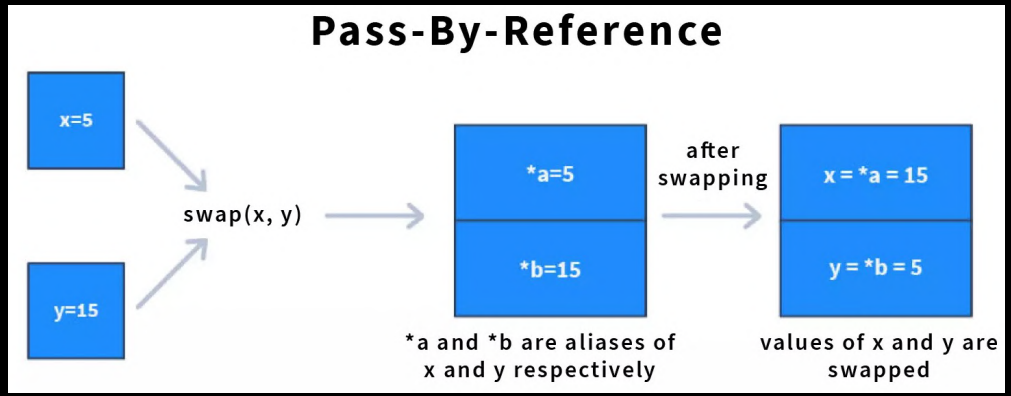
pass by value

cup =

fillCup( )

9.10 Pass by Value vs Pass by reference





1. Variable Pass by Value:
   - Java's default method.
   - Copies argument's value to function's parameter.
   - Changes in function don't affect original variable.
2. Objects and References:
   - Java passes the reference's value for objects.
   - Modifications to objects in methods affect originals.
3. Primitive Types:
   - Always passed by value.
   - In-function changes don't impact originals.

CHALLENGE

85. In a class Calculator, create multiple add() methods that overload each other and can sum two integers, three integers, or two doubles. Demonstrate how each can be called with different numbers of parameters.

86. Define a base class Vehicle with a method service() and a subclass Car that overrides service(). In Car's service(), provide a specific implementation that calls super.service() as well, to show how overriding works.

Revision

1. What is Abstraction
2. Abstract Keyword
3. Interfaces
4. What is Polymorphism
5. References and Objects
6. Method / Constructor Overloading
7. Super Keyword
8. Method / Constructor Overriding
9. Final keyword revisited
10. Pass by Value vs Pass by reference.

Practice Exercise

Abstraction and Polymorphism

Answer in True/False

1. Abstraction hides the internal implementation details and shows only the functionality to the users.
2. An abstract class must contain at least one abstract method.
3. When you pass an object to a method, Java copies the reference to the object.
4. Overloaded methods must have different return types.
5. The 'super' keyword can be used to access methods of the superclass that are hidden by the subclass.
6. Method overriding is used to change the default behaviour of a method in the superclass.
7. The final keyword in Java is used to define constants.
8. Java supports pass by reference for primitive types.
9. An interface in Java can contain instance fields (variables).
10. A subclass can override a protected method from its superclass with a public access modifier.

Practice Exercise

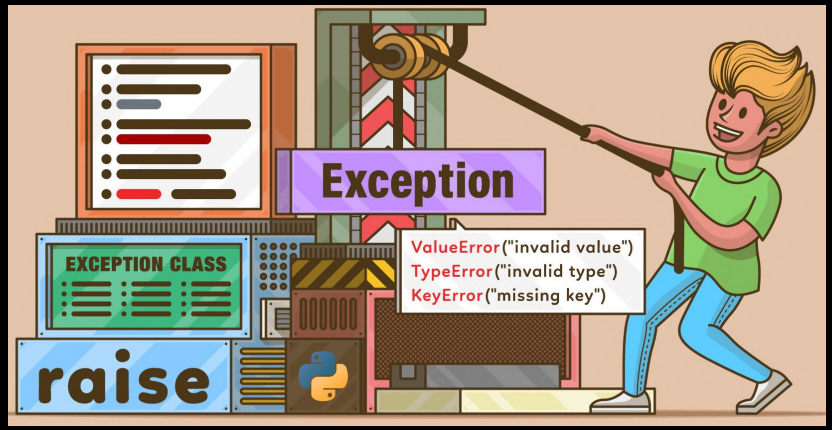
Abstraction and Polymorphism

Answer in True/False

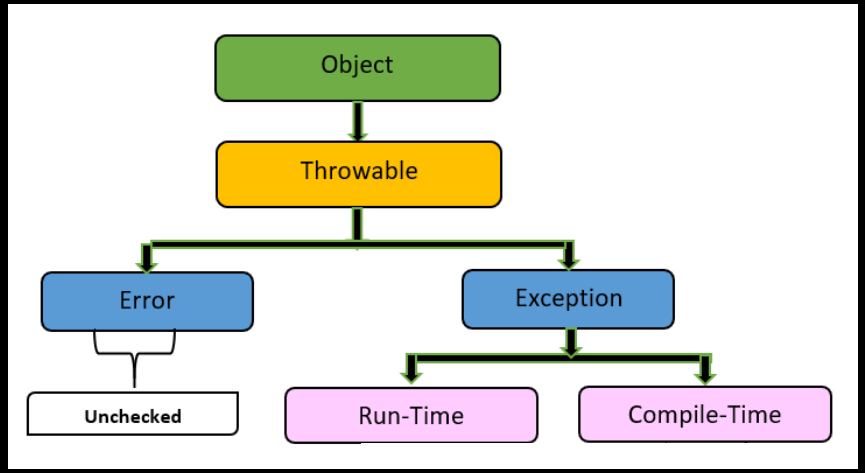
1. Abstraction hides the internal implementation details and shows only the functionality to the users. — True
2. An abstract class must contain at least one abstract method. — False
3. When you pass an object to a method, Java copies the reference to the object. — True
4. Overloaded methods must have different return types. — False
5. The 'super' keyword can be used to access methods of the superclass that are hidden by the subclass. — True
6. Method overriding is used to change the default behaviour of a method in the superclass. — True
7. The final keyword in Java is used to define constants. — True
8. Java supports pass by reference for primitive types. — False
9. An interface in Java can contain instance fields (variables). — False
10. A subclass can override a protected method from its superclass with a public access modifier. — True

10. Exception & File Handling

1. What is an Exception
2. Try-Catch
3. Types of Exception
4. Throw and Throws
5. Finally Block
6. Custom Exceptions
7. FileWriter class
8. FileReader class



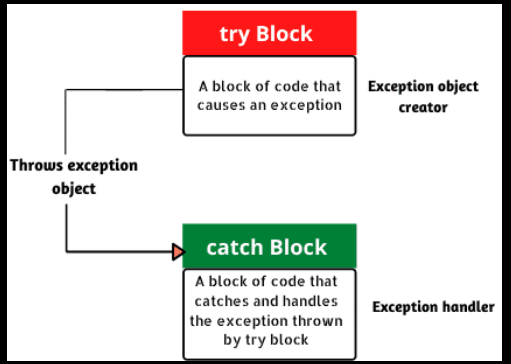
10.1 What is an Exception



1. In Java, an exception is a disruptive event that occurs during the execution of a program, interrupting its normal flow. It's an instance of a problem that arises while the program is running, such as arithmetic errors, null pointer accesses, or resource overflows.
2. Exceptions are objects in Java that encapsulate information about an error event, including its type and the state of the program when the error occurred.

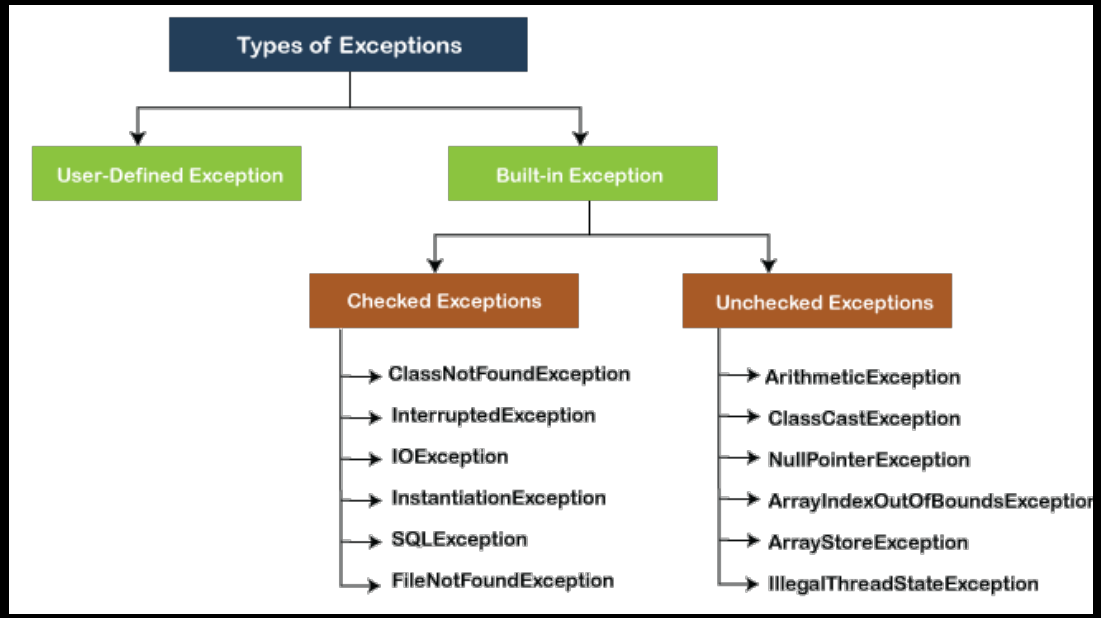
10.2 Try-Catch

```java
try{
    int num = 60/0;
    System.out.println(num);
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("Array index exception: " + e.getStackTrace());
} catch (NumberFormatException | ArithmeticException e) {
    System.out.println("Multiple exceptions");
} catch (Exception e) {
    System.out.println("Last exception");
}
```



1. Try Block: Contains code that is susceptible to exceptions.
2. Catch Block: Follows the try block and handles the exceptions thrown by the try block.
3. When an exception occurs in the try block, the control is transferred to the catch block, where the exception is handled.

10.3 Types of Exceptions



1. Checked Exceptions: These are exceptions that must be either caught or declared in the method.
2. Unchecked Exceptions: These are exceptions that do not need to be explicitly handled.

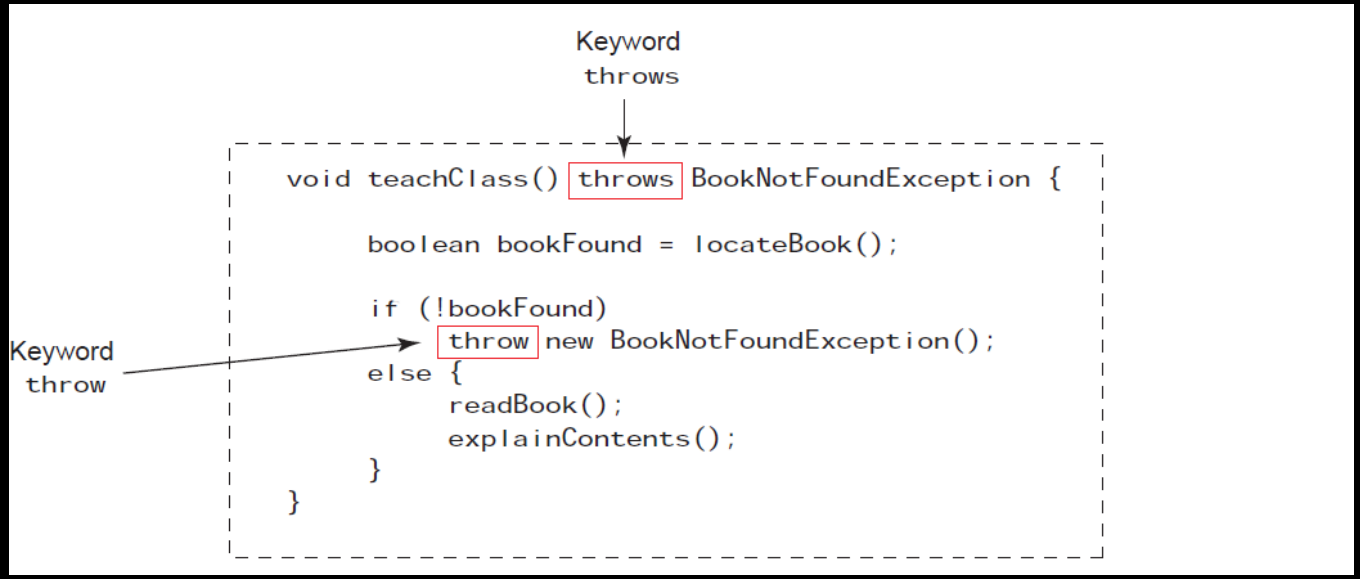
10.4 Throw and Throws

```
public void printName(String name)
        throws IllegalArgumentException {
    if (name.contains("-")) {
        throw new IllegalArgumentException("Name contains -");
    }
    System.out.println(name);
}
```

throws Keyword:

1. Declares that a method may throw one or more exceptions.
2. Used in the method signature to indicate that the method might throw exceptions of specified types.
3. A method declared with throws requires the calling method to handle or further declare the exception.

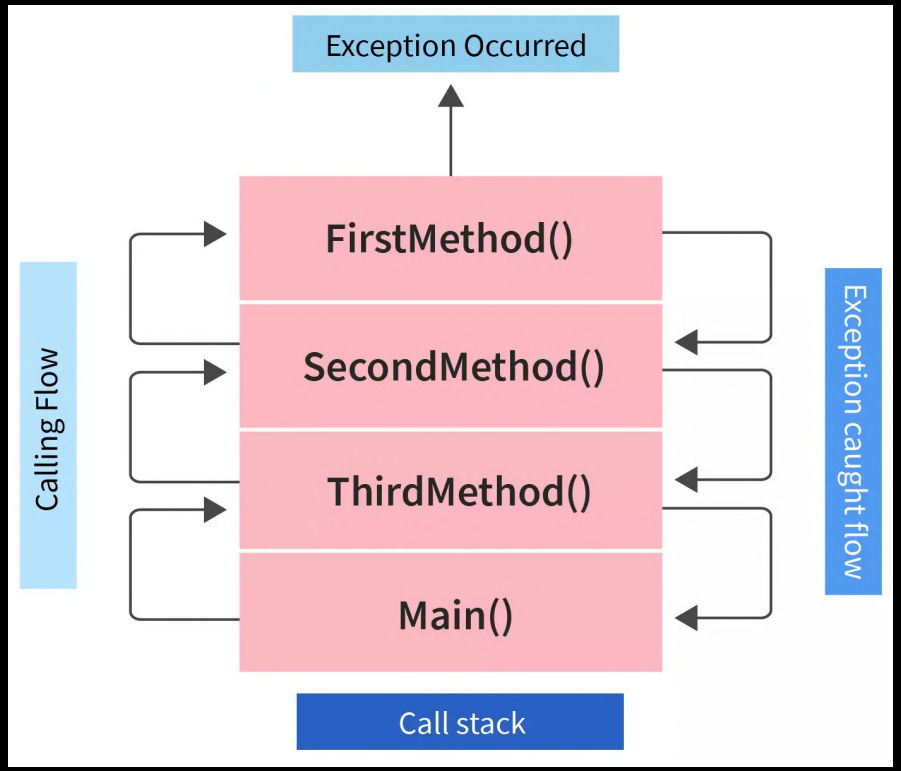
10.4 Throw and Throws



throw Keyword:

1. Used to explicitly throw an exception from any method or block of code.
2. You can throw either a new instance of an exception or an existing exception object using throw.
3. Example: throw new ArithmeticException("Division by zero");

10.4 Throw and Throws



10.4 Throw and Throws

throw	throws
Java **throw** keyword is used to explicitly throw an exception.	Java **throws** keyword is used to declare an exception.
Checked exception cannot be propagated using **throw**.	Checked exception can be propagated with **throws**.
If we see syntax wise, **throw** is followed by an instanceof Exception class Example : throw new NumberFormatException("The month entered, is invalid.");	If we see syntax wise, **throws** is followed by exception class names. Example : throws IOException,SQLException
The keyword **throw** is used inside method body.	**throws clause** is used in method declaration (signature).
By using **throw keyword** in java you cannot throw more than one exception. Example: throw new IOException("Connection failed!!")	By using **throws** you can declare multiple exceptions. Example: public void method() throws IOException,SQLException.

10.5 Finally Block

```
try {
    try {
        int result = 1 / 0;
    } catch (SomeException e) {
        System.out.println("Something caught");
    } finally {
        System.out.println("Not quite finally");
    }
} catch (ArithmeticException e) {
    System.out.println("ArithmeticException caught");
} finally {
    System.out.println("Finally");
}
```

1. Executes code after the try-catch blocks, used mainly for cleanup operations.
2. Always runs regardless of whether an exception is thrown or caught in the try-catch blocks.
3. Ideal for closing resources like files or database connections to prevent resource leaks.

10.6 Custom Exceptions

```
public class TemperatureException extends Exception {
    private double degrees;

    public TemperatureException(double degrees) {
        this.degrees = degrees;
    }

    public String getMessage() {
        return "The temperature (" + degrees
            + "C) isn't in the normal range.";
    }

    public double getDegrees() {
        return degrees;
    }
}
```

Override of Exception's getMessage() method

New method

1. Custom exceptions are user-defined exception classes that extend either Exception (for checked exceptions) or RuntimeException (for unchecked exceptions).
2. They are created to represent specific error conditions relevant to an application.

87. Arithmetic Exception Handling

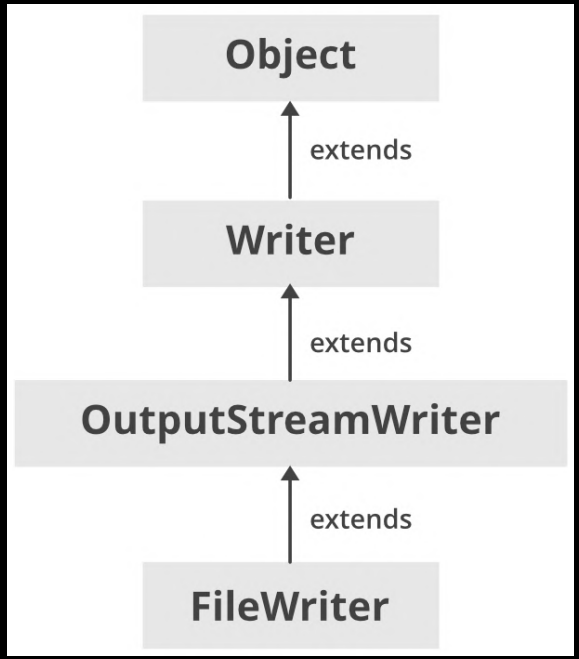
Write a program that asks the user to enter two integers and then divides the first by the second. The program should handle any arithmetic exceptions that may occur (like division by zero) and display an appropriate message.

Key Points:

- Use Scanner to read user input.
- Implement a try-catch block to handle ArithmeticException.
- Display a user-friendly message if division by zero occurs.

10.7 FileWriter class

1. FileWriter is used for writing streams of characters to files.
2. It's a character-based stream, which means it's best used for writing text rather than binary data.
3. Constructors:
   - FileWriter(String fileName): Creates a FileWriter object given the name of the file to write to.
   - FileWriter(File file): Creates a FileWriter object given a File object.
4. Common Methods:
   - write(int c): Writes a single character.
   - write(char[] cbuf): Writes an array of characters.
   - write(String str): Writes a string.
   - flush(): Flushes the stream, ensuring all data is written out.
   - close(): Closes the stream, releasing any associated system resources.



10.7 FileWriter class

```java
public class FileWriterExample {
    new *
    public static void main(String[] args) {
        // Define the filename
        String fileName = "example.txt";

        // Create a FileWriter object
        try (FileWriter writer = new FileWriter(fileName)) {
            // Write a string to the file
            writer.write( str: "Hello, this is a test.");

            // Optionally, you can flush the writer
            writer.flush();

            System.out.println("Successfully written to the file.");
        } catch (IOException e) {
            System.out.println("An error occurred.");
            e.printStackTrace();
        }
    }
}
```

10.8 FileReader class

1. The FileReader class is used for reading streams of characters from files.
2. It's a character-based stream, meaning it reads characters (as opposed to bytes). This makes it suitable for reading text files.
3. Constructors:
   - FileReader(String fileName): Creates a FileReader object to read from a file with the specified name.
   - FileReader(File file): Creates a FileReader object to read from the specified File object.
4. Common Methods:
   - read(): Reads a single character and returns it as an integer. Returns -1 if the end of the stream is reached.
   - read(char[] cbuf): Reads characters into an array and returns the number of characters read.

```java
public class FileReaderExample {
    new *
    public static void main(String[] args) {
        // Define the filename
        String fileName = "example.txt";

        // Create a FileReader object
        try (FileReader reader = new FileReader(fileName)) {
            int character;

            // Read and display characters one by one
            while ((character = reader.read()) != -1) {
                System.out.print((char) character);
            }
        } catch (IOException e) {
            System.out.println("An error occurred.");
            e.printStackTrace();
        }
    }
}
```

88. File Not Found Exception Handling

Write a program to read a filename from the user and display its content. The program should handle the situation where the file does not exist.

Key Points:

- Use Scanner to read the filename from the user.
- Use FileReader to read the file content.
- Implement a try-catch block to handle FileNotFoundException.
- Display a message informing the user if the file is not found.

Revision

1. What is an Exception
2. Try-Catch
3. Types of Exception
4. Throw and Throws
5. Finally Block
6. Custom Exceptions
7. FileWriter class
8. FileReader class

Practice Exercise

Exception & File Handling

Answer in True/False

1. A method that throws a checked exception must declare the exception using the throws keyword in its signature.
2. It is possible to handle more than one type of exception in a single catch block.
3. The finally block is always executed, even if a return statement is encountered in the try or catch block.
4. If both try and catch blocks have return statements, the finally block will not execute.
5. You can throw an exception of any type, including custom exceptions, using the throw keyword.
6. If a try block does not throw any exception, the catch block is executed for cleanup purposes.
7. Unchecked exceptions are a direct subclass of Throwable.
8. A catch block can exist independently of a try block.
9. You can nest try blocks within another try block.
10. The throw keyword is used to propagate an exception up the call stack.

Practice Exercise

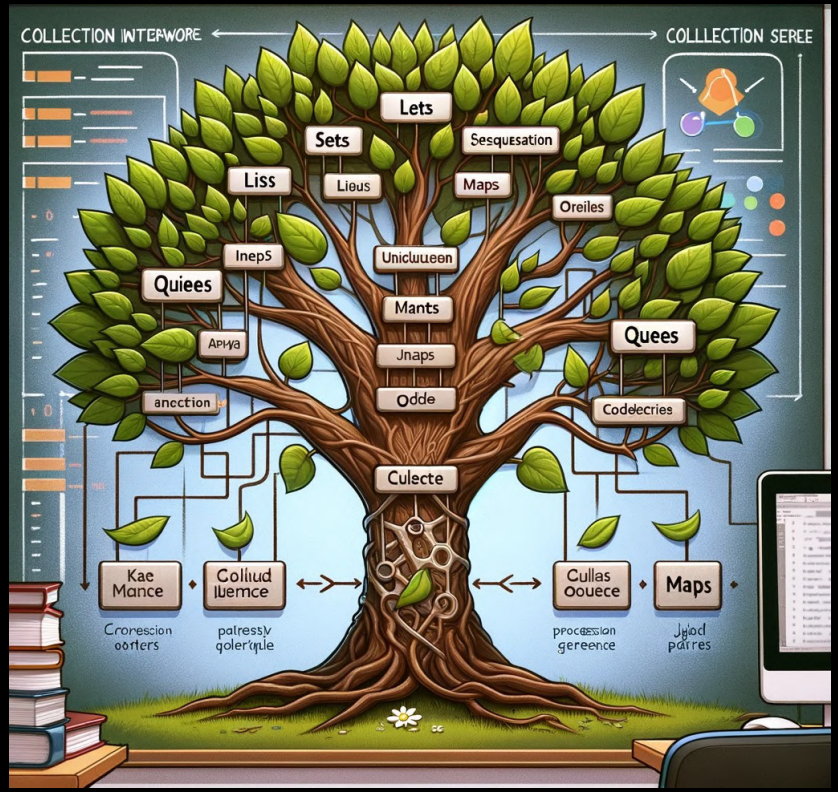
Exception & File Handling

Answer in True/False

1. A method that throws a checked exception must declare the exception using the throws keyword in its signature. — True
2. It is possible to handle more than one type of exception in a single catch block. — True
3. The finally block is always executed, even if a return statement is encountered in the try or catch block. — True
4. If both try and catch blocks have return statements, the finally block will not execute. — False
5. You can throw an exception of any type, including custom exceptions, using the throw keyword. — True
6. If a try block does not throw any exception, the catch block is executed for cleanup purposes. — False
7. Unchecked exceptions are a direct subclass of Throwable. — False
8. A catch block can exist independently of a try block. — False
9. You can nest try blocks within another try block. — True
10. The throw keyword is used to propagate an exception up the call stack. — True

11 Collections & Generics

1. Variable Arguments
2. Wrapper Classes & Autoboxing
3. Collections Library
4. List Interface
5. Queue Interface
6. Set Interface
7. Collections Class
8. Map Interface
9. Enums
10. Generics & Diamond Operators

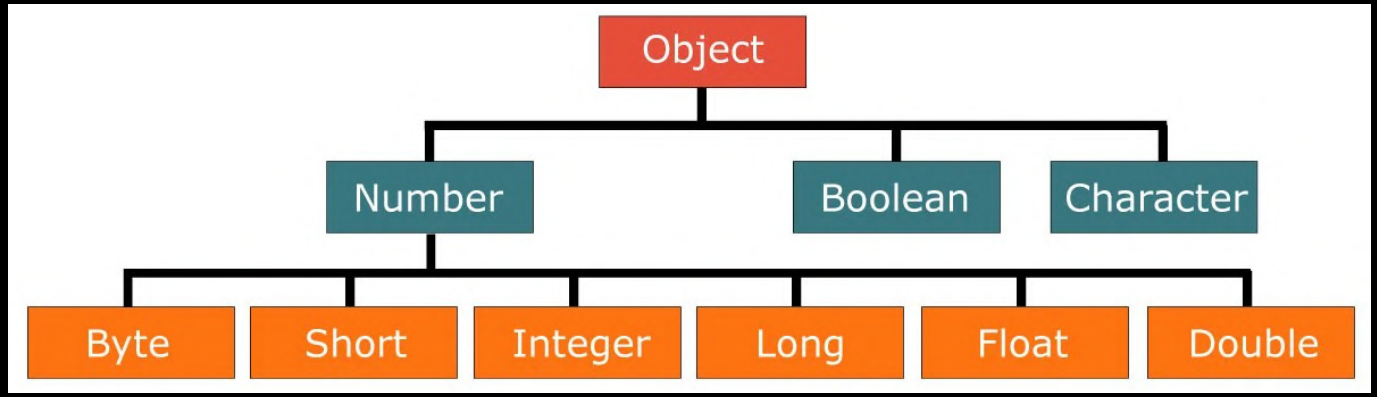


11.1 Variable Arguments

```java
public class MyClass {
    4 usages   new *
    static void printMany(String ...elements) {
        for (String element : elements) {
            System.out.printf(element);
        }
    }

    new *
    public static void main(String[] args) {
        printMany( ...elements: "one", "two", "three");
        printMany( ...elements: "one", "two");
        printMany();
        printMany(new String[]{"one", "two", "three"});
    }
}
```

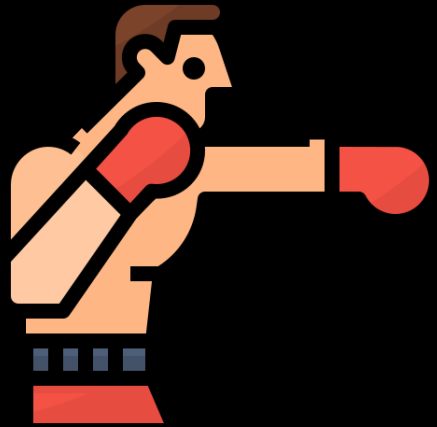
1. Java's varargs allow methods to accept any number of arguments.
2. Declared using an ellipsis (...), e.g., void method(int... nums).
3. Internal Handling: Treated as arrays, e.g., int... nums is int[] nums.
4. Placement: Must be the last in the method's parameters.
5. Usage: Call with varying argument counts, e.g., method(1, 2) or method().
6. Introduced in: Java 5.

11.2 Wrapper Classes



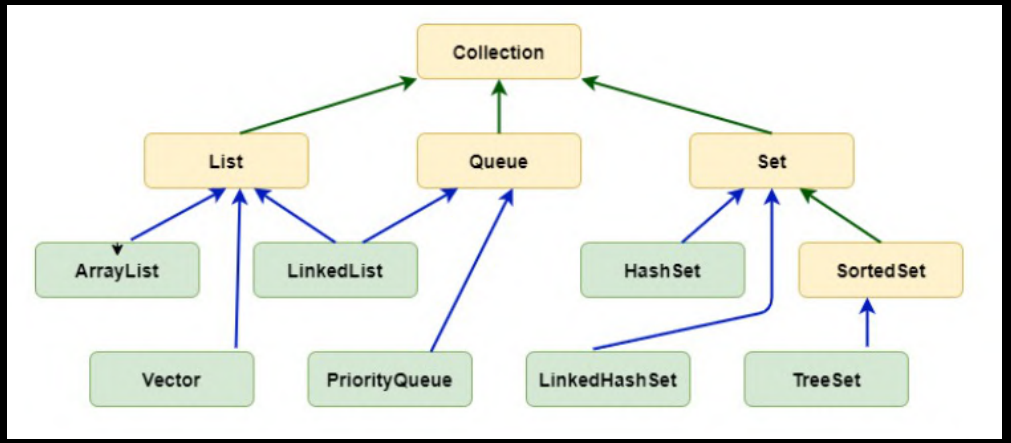
1. Provide a way to use primitive data types (int, char, boolean, etc.) as objects.
2. Automatic conversion between the primitive types and their corresponding wrapper classes.
3. Once created, the value of a wrapper object cannot be changed.
4. Utility Methods: Each wrapper class provides useful methods, like compareTo, valueOf, and parseXxx (e.g., parseInt for Integer).
5. Required for storing primitives in collection objects like ArrayList, HashMap, etc.
6. Allows assignment of null to primitive values when needed.

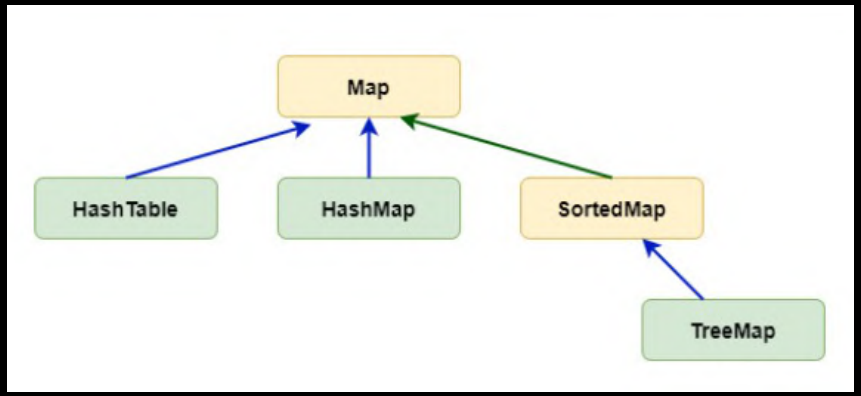
11.2 Autoboxing



1. Autoboxing: Automatic conversion of primitive types to their corresponding wrapper class objects.
2. Unboxing: Automatic conversion of wrapper class objects back to their respective primitive types.

11.3 Collections Library





1. Collection Interface: The root interface of the collection hierarchy. It declares basic operations like add, remove, clear, and size.
2. List Interface: An ordered collection. Lists can contain duplicate elements.
3. Set Interface: A collection that cannot contain duplicate elements.
4. Queue Interface: A collection used for holding elements in FIFO prior to processing.
5. Map Interface: Not a true Collection, but part of the Collections Framework. Maps store key-value pairs. Keys are unique, but different keys can map to the same value.

11.4 List Interface

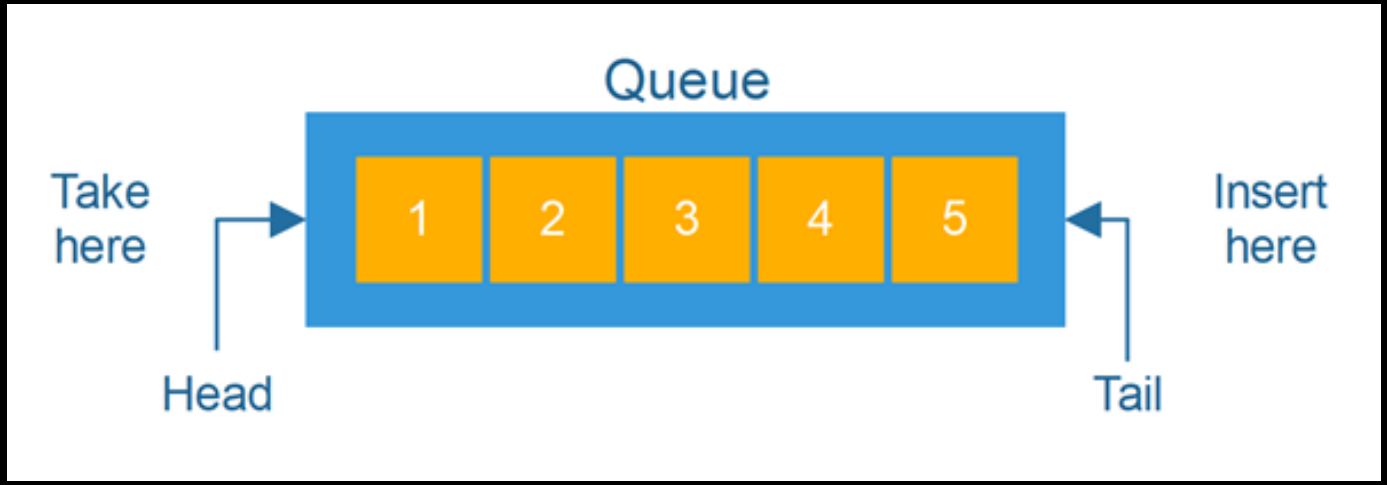
1. An ordered collection (also known as a sequence).
2. Allows duplicate elements.
3. Elements can be accessed by their integer index.
4. Maintains the insertion order of elements.
5. Performance: Offers fast random access and quick iteration.
6. Capacity: Grows automatically as elements are added.
7. Preferred over arrays when the size is dynamic or unknown.

11.4 List Interface

1. add(E e): Appends the specified element
2. add(int index, E element): Inserts at specified position
3. remove(Object o): Removes the first occurrence of the specified element
4. remove(int index): Removes the element at the specified position
5. get(int index): Returns the element at the specified position
6. set(int index, E element): Replaces the element at the specified position
7. size(): Returns the number of elements
8. clear(): Removes all of the elements
9. contains(Object o): Returns `true` if the list contains the specified element.
10. indexOf(Object o): Returns the index of the first occurrence, or -1 if the list does not contain the element.

```java
//List Creation
List<Integer> list = new ArrayList<Integer>();
//List Addition
list.add(1);
list.add(2);
//Printing the List
System.out.println(list);
```

11.5 Queue Interface



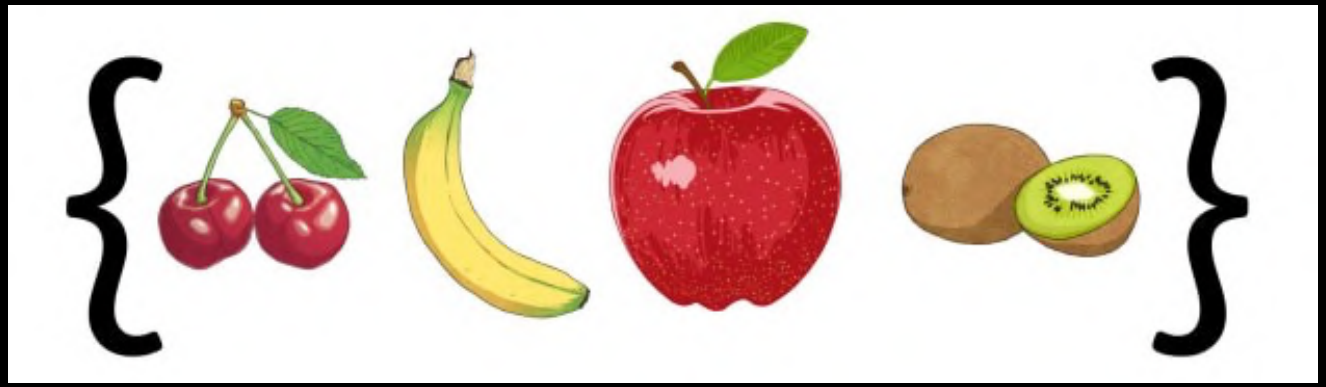
1. It's a collection designed for holding elements prior to processing.
2. Ordering: Typically, it orders elements in a FIFO (First-In-First-Out) manner.
3. End Points: Offers two ends - one for insertion (tail) and the other for removal (head).

11.5 Queue Interface

1. add(E e): Inserts the specified element into the queue. Throws an exception if the element cannot be added.
2. offer(E e): Inserts the specified element into the queue. Returns false if the element cannot be added.
3. remove(): Retrieves and removes the head of the queue. Throws an exception if the queue is empty.
4. poll(): Retrieves and removes the head of the queue, or returns null if the queue is empty.
5. element(): Retrieves, but does not remove, the head of the queue. Throws an exception if the queue is empty.
6. peek(): Retrieves, but does not remove, the head of the queue, or returns null if the queue is empty.

```
// Creating a Queue using LinkedList
Queue<String> queue = new LinkedList<>();

// Adding elements (offer method can also be used)
queue.add("First");
queue.add("Second");

// Displaying the head of the queue
System.out.println("Head of queue: " + queue.peek());

// Removing elements from the queue
while (!queue.isEmpty()) {
    System.out.println("Removed: " + queue.poll());
}
```

11.6 Set Interface



1. Unique Elements: Does not allow duplicate elements.
2. Unordered Collection: it does not guarantee any specific ordering of elements.
3. No Positional Access: Unlike lists, it doesn't support indexing-based access to elements.
4. Implementation: Common implementations include HashSet, LinkedHashSet, and TreeSet.

11.6 Set Interface

```java
// Creating a Set
Set<String> set = new HashSet<>();

// Adding elements
set.add("Apple");

// Attempting to add a duplicate element
boolean isAdded = set.add("Apple"); // This will be false
System.out.println("Apple added again: " + isAdded);

// Checking if a specific element is in the set
boolean containsOrange = set.contains("Orange");
System.out.println("Contains Orange: " + containsOrange);
```

1. add(E e): Adds the specified element to the set
2. remove(Object o): Removes the specified element from the set
3. contains(Object o): Checks if the set contains the specified element.
4. size(): Returns the number of elements in the set.
5. isEmpty(): Checks if the set is empty.

11.7 Collections Class

```java
// Creating a list
List<Integer> list = new ArrayList<>();
Collections.addAll(list, ...elements: 5, 1, 8, 3, 2);

// Sorting the list
Collections.sort(list);
System.out.println("Sorted list: " + list);

// Finding max and min in the list
int max = Collections.max(list);
int min = Collections.min(list);
System.out.println("Max: " + max + ", Min: " + min);
```

```java
// Reversing the list
Collections.reverse(list);
System.out.println("Reversed list: " + list);

// Searching in the list (List must be sorted)
Collections.sort(list);
int index = Collections.binarySearch(list, key: 3);
System.out.println("Index of 3: " + index);

// Creating an immutable list
List<Integer> unmodifiableList = Collections.unmodifiableList(list);
System.out.println("Unmodifiable list: " + unmodifiableList);
```

1. Offers methods like sort to sort lists.
2. Provides methods like binarySearch for searching sorted lists.
3. Allows reversing the order of elements in a list with reverse.
4. Can shuffle the elements of a list randomly using shuffle.
5. Creates unmodifiable collections using methods like unmodifiableList, etc.
6. Methods like singletonList, create immutable collections with a single element.
7. The copy method is used to copy all elements from one list to another.

CHALLENGE

89. Write a method concatenate Strings that takes variable arguments of String type and concatenates them into a single string.
90. Write a program that sorts a list of String objects in descending order using a custom Comparator.
91. Use the Collections class to count the frequency of a particular element in an ArrayList.
92. Write a method that swaps two elements in an ArrayList, given their indices.
93. Create a program that reverses the elements of a List and prints the reversed list.
94. Create a PriorityQueue of a custom class Student with attributes name and grade. Use a comparator to order by grade.
95. Write a program that takes a string and returns the number of unique characters using a Set.

11.8 Map Interface

1. Stores data as key-value pairs.
2. Each key can map to at most one value.
3. Keys are unique, but multiple keys can map to the same value.
4. It is part of the Collections Framework but does not extend the Collection interface.

11.8 Map Interface

```java
// Creating a Map
Map<String, Integer> map = new HashMap<>();

// Adding key-value pairs to the Map
map.put("Apple", 10);

// Accessing a value
Integer appleCount = map.get("Apple");
System.out.println("Apples count: " + appleCount);

// Checking if a key exists
if (map.containsKey("Banana")) {
    System.out.println("Banana is in the map");
}

// Removing a key-value pair
map.remove( key: "Orange");
```

1. put(K key, V value): Associates the specified value with the specified key in the map.
2. get(Object key): Returns the value to which the specified key is mapped, or null if the map contains no mapping for the key.
3. remove(Object key): Removes the mapping for a key from the map if it is present.
4. containsKey(Object key): Checks if the map contains a mapping for the specified key.
5. keySet(): Returns a Set view of the keys contained in the map.
6. values(): Returns a Collection view of the values contained in the map.

11.9 Enums

```java
enum TrafficSignal{
    RED("stop"), GREEN("start"), ORANGE("slow down");

    private String action;

    public String getAction(){
        return this.action;
    }
    private TrafficSignal(String action){
        this.action = action;
    }
}

// String to Enum using valueOf
TrafficSignal signal = TrafficSignal.valueOf("RED");
signal = TrafficSignal.valueOf("GREEN"); //OK
signal = TrafficSignal.valueOf("Green"); //Not Ok
```

1. Enums in Java: Special types for fixed sets of constants like days, colors.
2. Declaration: Use enum keyword, e.g., enum Color { RED, GREEN, BLUE; }.
3. Access: Access constants with dot syntax, e.g., Color.RED.
4. Features: Type-safe, readable, can have methods and fields.
5. Usage: Useful in switch statements and iterating with values() method.

11.10 Generics & Diamond Operators

```java
class SpecificClass {
    2 usages
    private String thing;
    no usages   new *
    public String getThing() {
        return thing;
    }
    no usages   new *
    public void setThing (String thing) {
        this.thing = thing;
    }
}
```

```java
class GenericClass<T> {
    2 usages
    private T thing;
    no usages   new *
    public T getThing() {
        return thing;
    }
    no usages   new *
    public void setThing(T thing) {
        this.thing = thing;
    }
}
```

1. They allow you to write flexible and reusable code by enabling types (classes and interfaces) to be parameters when defining classes, interfaces, and methods.
2. Generics provide compile-time type safety by allowing you to enforce that certain objects are of a specific type.
3. With generics, you don't need to cast objects because the type is known.
4. Generics are denoted by angle brackets <>, e.g., List<String> means a list of strings.
5. Diamond Operator: Introduced in Java 7, the diamond operator <> allows you to infer the type parameter from the context, simplifying instantiation of generic classes.

CHALLENGE

96. Create an enum called Day that represents the days of the week. Write a program that prints out all the days of the week from this enum.
97. Enhance the Day enum by adding an attribute that indicates whether it is a weekday or weekend. Add a method in the enum that returns whether it's a weekday or weekend, and write a program to print out each day along with its type.
98. Create a Map where the keys are country names (as String) and the values are their capitals (also String). Populate the map with at least five countries and their capitals. Write a program that prompts the user to enter a country name and then displays the corresponding capital, if it exists in the map.

Revision

1. Variable Arguments
2. Wrapper Classes & Autoboxing
3. Collections Library
4. List Interface
5. Queue Interface
6. Set Interface
7. Collections Class
8. Map Interface
9. Enums
10. Generics & Diamond Operators

Practice Exercise

Collections & Generics

Answer in True/False

1. Variable arguments can be used to pass an arbitrary number of values to a method.
2. Autoboxing is the process where primitive types are automatically converted into their corresponding wrapper class objects by the Java compiler.
3. Every class in the Collections Library implements the Collection interface.
4. Lists guarantee the order of insertion for the elements they contain.
5. Sets in Java inherently maintain the elements in sorted order.
6. In a Map, you can have multiple entries with the same value but not with the same key.
7. Enums in Java can have constructors, methods, variables, and can implement interfaces.
8. With the diamond operator, you do not need to specify the type on the right-hand side of a statement when initializing an object.
9. The List interface provides methods for inserting elements at a specific position in the list.
10. Wrapper classes in Java, such as Integer and Double, can be extended like any other class.

Practice Exercise

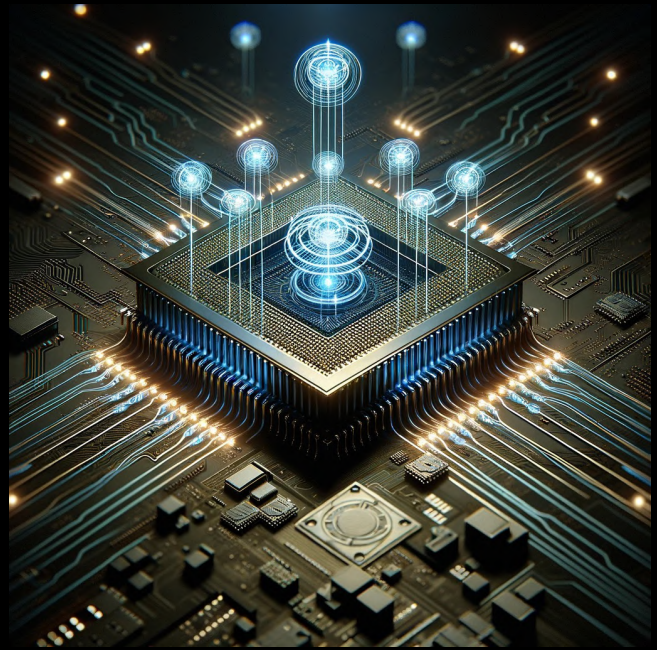
Collections & Generics

Answer in True/False

1. Variable arguments can be used to pass an arbitrary number of values to a method. — True
2. Autoboxing is the process where primitive types are automatically converted into their corresponding wrapper class objects by the Java compiler. — True
3. Every class in the Collections Library implements the Collection interface. — False
4. Lists guarantee the order of insertion for the elements they contain. — True
5. Sets in Java inherently maintain the elements in sorted order. — False
6. In a Map, you can have multiple entries with the same value but not with the same key. — True
7. Enums in Java can have constructors, methods, variables, and can implement interfaces. — True
8. With the diamond operator, you do not need to specify the type on the right-hand side of a statement when initializing an object. — True
9. The List interface provides methods for inserting elements at a specific position in the list. — True
10. Wrapper classes in Java, such as Integer and Double, can be extended like any other class. — False

12 Multi threading & Executor Service

1. Intro to Multi-threading
2. Creating a Thread
3. States of a Thread
4. Thread Priority
5. Join Method
6. Synchronize keyword
7. Thread Communication
8. Intro to Executor Service
9. Multiple Threads with Executor
10. Returning Futures



12.1 What is a Thread

1. What is a Thread: A thread in Java is a small part of a program that can run at the same time as other parts.
2. Purpose: Threads help a program do many things at once, like handling many users or doing different tasks simultaneously.
3. Creating Threads: You can make a thread by using the Thread class or the Runnable interface.
4. Using Threads: Use threads for tasks that can happen at the same time, like managing many requests or splitting up a big job.
5. Thread Talk: Threads can talk to each other using wait(), notify(), and notifyAll() to coordinate their work.

12.1 Need of Multi-threading

```java
// First Task
for (int i = 1; i <= 1000; i++) {
    System.out.printf("%d:* ", i);
}
System.out.println("\nFirst Task Done");

// Second Task
for (int i = 1; i <= 1000; i++) {
    System.out.printf("%d:& ", i);
}
System.out.println("\nSecond Task Done");

// Third Task
for (int i = 1; i <= 1000; i++) {
    System.out.printf("%d:$ ", i);
}
System.out.println("\nThird Task Done");
```

1. Tasks might be very important
2. Tasks are independent of each other
3. A Multi-core CPU is sitting idle most of the time
4. A big task can be divided into smaller parts
5. Making your code responsive

12.2 Creating a Thread

(Extending Thread Class)

```
// Step 1: Define a Class that Extends Thread
4 usages
public class PrintTask extends Thread {
    // Step 2: Override the run() Method
    no usages
    public void run() {
        // First Task
        for (int i = 1; i <= 1000; i++) {
            System.out.printf("%d:%c ", i, targetChar);
        }
        System.out.printf("\n%c Task Done\n", targetChar);
    }
    3 usages
    private final char targetChar;
    2 usages
    public PrintTask(char targetChar) {
        this.targetChar = targetChar;
    }
}
```

```
public static void main(String... args) {
    // Step 3: Create an Instance of Your Class
    PrintTask t1 = new PrintTask( targetChar: '*');
    t1.start(); // Start the first thread

    PrintTask t2 = new PrintTask( targetChar: '$');
    t2.start(); // Start the second thread
}
```

In the main method, two threads (t1 and t2) are created and started. They will execute independently and print their values.

12.2 Creating a Thread

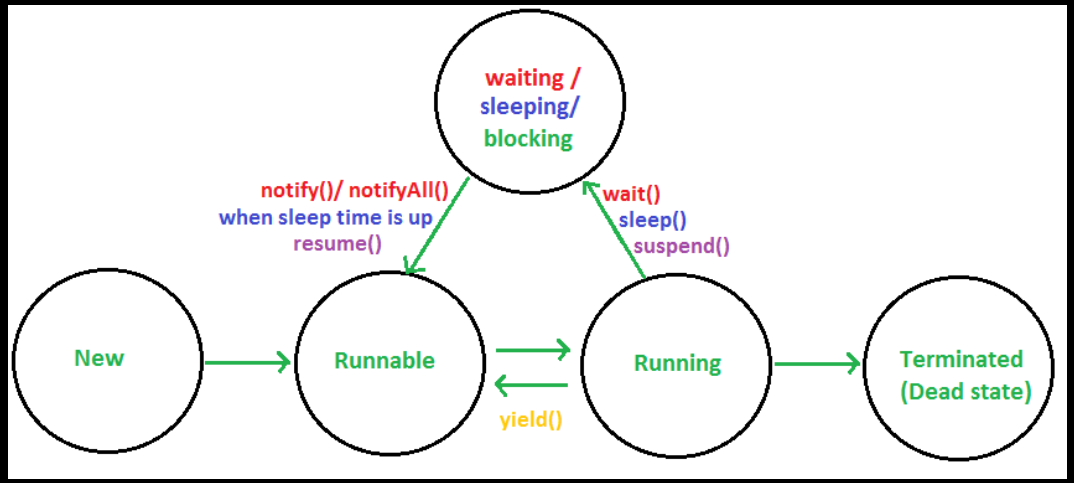
(Creating Runnables)

```
// Step 1: Define a Class that implements Runnable
4 usages
public class PrintRunnable implements Runnable {
    // Step 2: Override the run() Method
    no usages
    @Override
    public void run() {
        // First Task
        for (int i = 1; i <= 1000; i++) {
            System.out.printf("%d:%c ", i, targetChar);
        }
        System.out.printf("\n%c Task Done\n", targetChar);
    }
    3 usages
    private final char targetChar;
    2 usages
    public PrintRunnable(char targetChar) {
        this.targetChar = targetChar;
    }
}
```

```
public static void main(String... args) {
    // Step 3: Create an Instance of Your Class
    PrintRunnable t1 = new PrintRunnable( targetChar: '*');
    // Step4: Wrap your class with a thread
    new Thread(t1).start(); // Start the first thread

    PrintRunnable t2 = new PrintRunnable( targetChar: '$');
    new Thread(t1).start(); // Start the second thread
}
```

In the main method, two threads (t1 and t2) are created and started. They will execute independently and print their values.

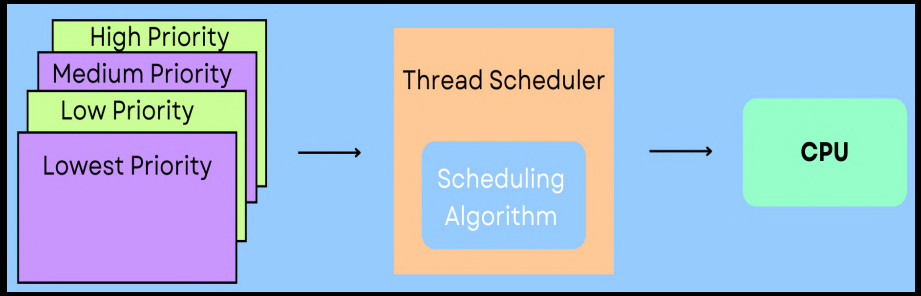
12.3 States of a Thread



1. New: Thread is created but not started.
2. Runnable: Thread is ready or running.
3. Running: Thread is actively executing tasks.
4. Blocked/Waiting: Thread is alive but not active because it's waiting for resources or other threads.
5. Terminated: Thread has finished or stopped running.

12.4 Thread Priority

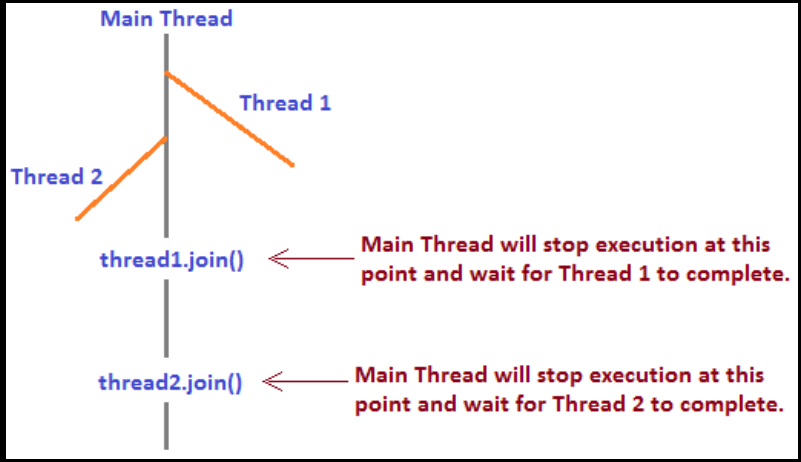
```java
class MyThread extends Thread {
    no usages
    public void run() {
        Thread current = Thread.currentThread();
        System.out.printf("Running thread name: %s\n",
                current.getName());
        System.out.printf("Running thread priority: %s\n",
                current.getPriority());
    }
}

public class ThreadPriority {
    public static void main(String args[]) {
        MyThread t1 = new MyThread();
        MyThread t2 = new MyThread();
        t1.setPriority(Thread.MIN_PRIORITY); // Setting priority to 1
        t2.setPriority(Thread.MAX_PRIORITY); // Setting priority to 10

        t1.setName("Thread-1");
        t2.setName("Thread-2");
        t1.start();
        t2.start();
    }
}
```



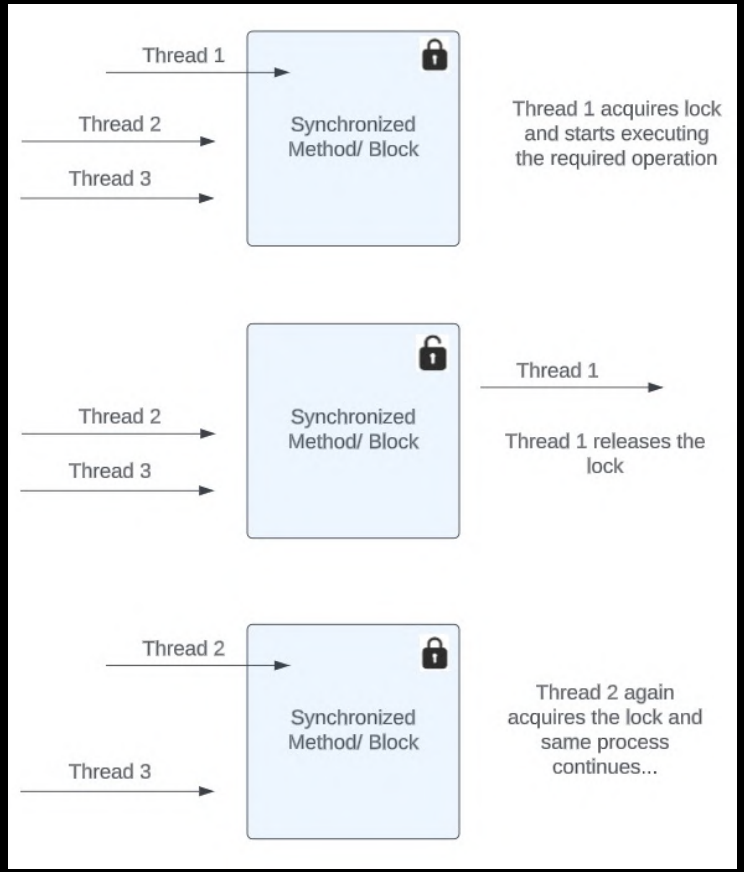
1. Priority Levels: Java threads have priority levels from 1 (lowest) to 10 (highest), with a default value of 5.
2. Influence on Execution: A thread's priority suggests the importance of a thread to the scheduler, though it doesn't guarantee the order of execution.
3. Set and Get Priority: Use setPriority(int) to change a thread's priority and getPriority() to retrieve it.

12.5 Join Method

1. Purpose of join: The join method is used to make the calling thread wait until the thread on which join has been called completes its execution.
2. Synchronization of Threads: join helps in synchronizing multiple threads, ensuring that a thread completes its execution before the next steps in the calling thread proceed.
3. Overloaded Versions: join comes in three versions:
   - join(): Waits indefinitely until the thread on which it's called finishes.
   - join(long millis): Waits for the thread to die for the specified milliseconds.
   - join(long millis, int nanos): Waits for the thread to die for the specified milliseconds plus nanoseconds.



12.6 Synchronize keyword



1. Mutual Exclusion: The synchronize keyword in Java ensures that only one thread can execute a block of code at a time, providing mutual exclusion and preventing race conditions.
2. Object Lock: When a thread enters a synchronized block or method, it acquires a lock on the object or class, depending on whether the method is an instance method or a static method.
3. Visibility: It ensures that changes made by one thread to shared data are visible to other threads.

12.6 Synchronize keyword

```
class Counter {
    2 usages
    private int count = 0;
    // Synchronized method to increment the counter
    1 usage
    public synchronized void increment() {
        count++;
    }
    // Method to get the current count
    1 usage
    public int getCount() {
        return count;
    }
}
4 usages
class SynchronizedThread extends Thread {
    2 usages
    private Counter counter;
    2 usages
    public SynchronizedThread(Counter counter) {
        this.counter = counter;
    }
    no usages
    public void run() {
        for (int i = 0; i < 1000; i++) {
            counter.increment();
        }
    }
}
```

```
public class SynchronizedExample {
    public static void main(String[] args) throws InterruptedException {
        Counter counter = new Counter();
        SynchronizedThread t1 = new SynchronizedThread(counter);
        SynchronizedThread t2 = new SynchronizedThread(counter);

        t1.start();
        t2.start();

        t1.join();
        t2.join();
        System.out.println("Final count is: " + counter.getCount());
    }
}
```

12.7 Thread Communication

1. sleep(long millis): Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
2. yield(): Causes the currently executing thread to pause and allow other threads to execute. It's a way of suggesting that other threads of the same priority can run.
3. wait(): Causes the current thread to wait until another thread invokes the notify() or notifyAll() method for this object. It releases the lock held by this thread.
4. notify(): Wakes up a single thread that is waiting on the object's monitor. If any threads are waiting, one is chosen to be awakened.
5. notifyAll(): Wakes up all threads that are waiting on the object's monitor.

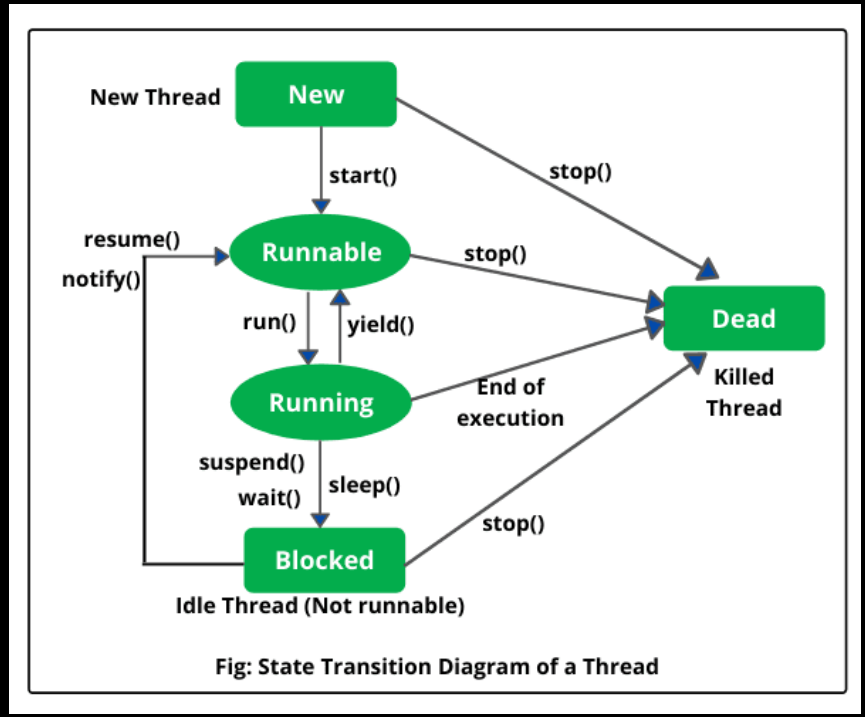
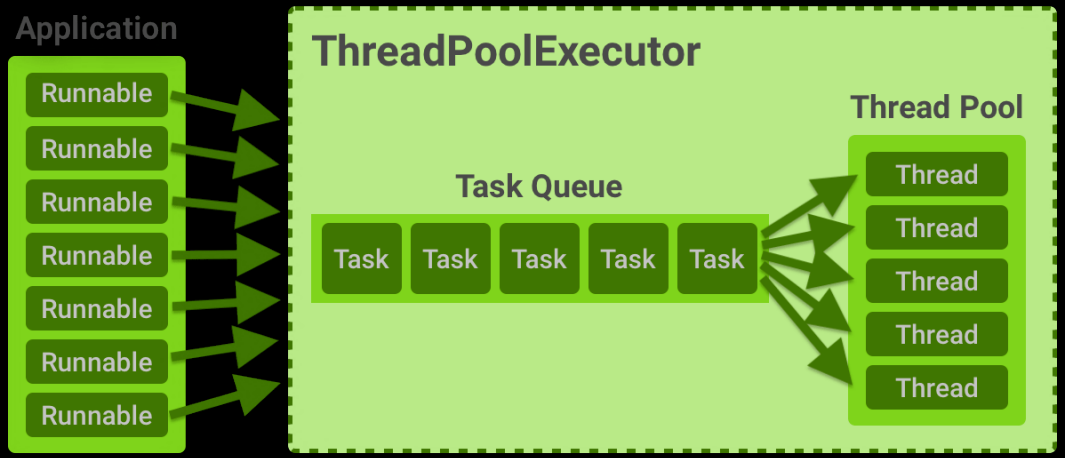


Fig: State Transition Diagram of a Thread

CHALLENGE

99. Write a program that creates two threads. Each thread should print "Hello from Thread X", where X is the number of the thread (1 or 2), ten times, then terminate.
100. Write a program that starts a thread and prints its state after each significant event (creation, starting, and termination). Use Thread.sleep() to simulate long-running tasks and Thread.getState() to print the thread's state.
101. Create three threads. Ensure that the second thread starts only after the first thread ends and the third thread starts only after the second thread ends using the join method. Each thread should print its start and end along with its name.
102. Simulate a traffic signal using threads. Create three threads representing three signals: RED, YELLOW, and GREEN. Each signal should be on for a certain time, then switch to the next signal in order. Use sleep for timing and synchronize to make sure only one signal is active at a time.

12.8 Intro to Executor Service



1. Purpose: ExecutorService is a framework provided by the Java Concurrency API to manage and execute submitted tasks without the need to manually manage thread life cycles.
2. Thread Pool Management: ExecutorService efficiently reuses a fixed pool of threads to execute tasks, thereby improving performance by reducing the overhead of thread creation, especially for short-lived asynchronous tasks.

12.8 Intro to Executor Service

```java
// Step 1: Define a Class that implements Runnable
5 usages
public class PrintRunnable implements Runnable {
    // Step 2: Override the run() Method
    no usages
    @Override
    public void run() {
        // First Task
        for (int i = 1; i <= 1000; i++) {
            System.out.printf("%d:%c ", i, targetChar);
        }
        System.out.printf("\n%c Task Done\n", targetChar);
    }
    3 usages
    private final char targetChar;
    3 usages
    public PrintRunnable(char targetChar) {
        this.targetChar = targetChar;
    }
}
```

```java
public class SingleThreadExecutorExample {
    public static void main(String[] args) {
        // Create a single-threaded executor
        ExecutorService executor = Executors.newSingleThreadExecutor();
        // Define a task (a Runnable)
        Runnable task = new PrintRunnable( targetChar: 'E');

        // Submit the task to the executor
        executor.submit(task);
        executor.shutdown();
    }
}
```

12.9 Multiple Threads with Executor

```java
public class PrintRunnable implements Runnable {
    // Step 2: Override the run() Method
    no usages
    @Override
    public void run() {
        // Task
        String threadName = Thread.currentThread().getName();
        System.out.printf("Executing task with char %c on" +
                "| Thread: %s \n", targetChar, threadName);
        for (int i = 1; i <= 1000; i++) {
            System.out.printf("%d:%c ", i, targetChar);
        }
        System.out.printf("\n%c Task Done\n", targetChar);
    }
    4 usages
    private final char targetChar;
    3 usages
    public PrintRunnable(char targetChar) {
        this.targetChar = targetChar;
    }
}
```

```java
public class MultiThreadExecutorExample {
    public static void main(String[] args) throws InterruptedException {
        // Create a single-threaded executor
        ExecutorService executor = Executors.newFixedThreadPool( nThreads: 3);

        // Define a task (a Runnable)
        for (int i = 0; i < 5; i++) {
            int taskNumber = i + 1;
            Runnable task = new PrintRunnable((char)taskNumber);
            // Submit the task to the executor
            executor.submit(task);
        }

        executor.shutdown();

        // Wait for all tasks to finish executing or timeout after 10 seconds
        if (!executor.awaitTermination( l: 10, TimeUnit.SECONDS)) {
            // Try to stop all actively executing tasks
            executor.shutdownNow();
        }
    }
}
```

12.10 Returning Futures

```java
// Create an executor with a fixed thread pool of 2 threads
ExecutorService executor = Executors.newFixedThreadPool( nThreads: 2);
// Submit the task to the executor and get a Future object
Future<String> future = executor.submit(task);

try {
    // Get the result from the Future object.
    String result = future.get();
    System.out.println("Result from future: " + result);
} catch (ExecutionException | InterruptedException e) {
    // Handle the interruption during the get
    Thread.currentThread().interrupt();
    System.out.println("Task was interrupted");
}

// Shut down the executor
executor.shutdown();
```

```java
// Define a callable task that returns a result
Callable<String> task = new Callable<String>() {
    @Override
    public String call() throws Exception {
        // Simulate task execution time
        TimeUnit.SECONDS.sleep( timeout: 1);
        return "Result from task";
    }
};
```

CHALLENGE

103. Write a program that creates a single-threaded executor service. Define and submit a simple Runnable task that prints numbers from 1 to 10. After submission, shut down the executor.
104. Create a fixed thread pool with a specified number of threads using Executors.newFixedThreadPool(int). Submit multiple tasks to this executor, where each task should print the current thread's name and sleep for a random time between 1 and 5 seconds. Finally, shut down the executor and handle proper termination using awaitTermination.
105. Write a program that uses an executor service to execute multiple Callable tasks. Each task should calculate and return the factorial of a number provided to it. Use Future objects to receive the results of the calculations. After all tasks are submitted, retrieve the results from the futures, print them, and ensure the executor service is shut down correctly.

Revision

1. Intro to Multi-threading
2. Creating a Thread
3. States of a Thread
4. Thread Priority
5. Join Method
6. Synchronize keyword
7. Thread Communication
8. Intro to Executor Service
9. Multiple Threads with Executor
10. Returning Futures

Practice Exercise

Multi threading & Executor Service

Answer in True/False

1. In Java, the main thread is the only user thread that is created automatically when a program starts.
2. The run() method is automatically called when a thread is started using the start() method.
3. A thread can be in both the Runnable and Running state at the same time.
4. Lower priority threads are executed first to ensure equal processing time for all threads.
5. The join() method causes the calling thread to immediately stop executing until the thread it joins with stops running.
6. Using the synchronized keyword can prevent two threads from executing a method simultaneously.
7. The notify() method wakes up all threads that are waiting on the object's monitor.
8. The ExecutorService must be explicitly shut down to terminate the threads it manages.
9. A Callable task cannot be submitted to an ExecutorService.
10. The Thread.yield() method forces a thread to stop its execution permanently.

Practice Exercise

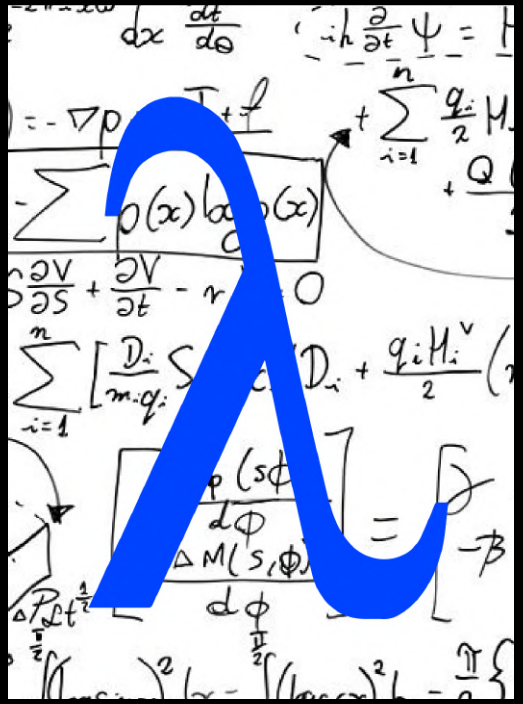
Multi threading & Executor Service

Answer in True/False

1. In Java, the main thread is the only user thread that is created automatically when a program starts. — True
2. The run() method is automatically called when a thread is started using the start() method. — True
3. A thread can be in both the Runnable and Running state at the same time. — False
4. Lower priority threads are executed first to ensure equal processing time for all threads. — False
5. The join() method causes the calling thread to immediately stop executing until the thread it joins with stops running. — True
6. Using the synchronized keyword can prevent two threads from executing a method simultaneously. — True
7. The notify() method wakes up all threads that are waiting on the object's monitor. — False
8. The ExecutorService must be explicitly shut down to terminate the threads it manages. — True
9. A Callable task cannot be submitted to an ExecutorService. — False
10. The Thread.yield() method forces a thread to stop its execution permanently. — False

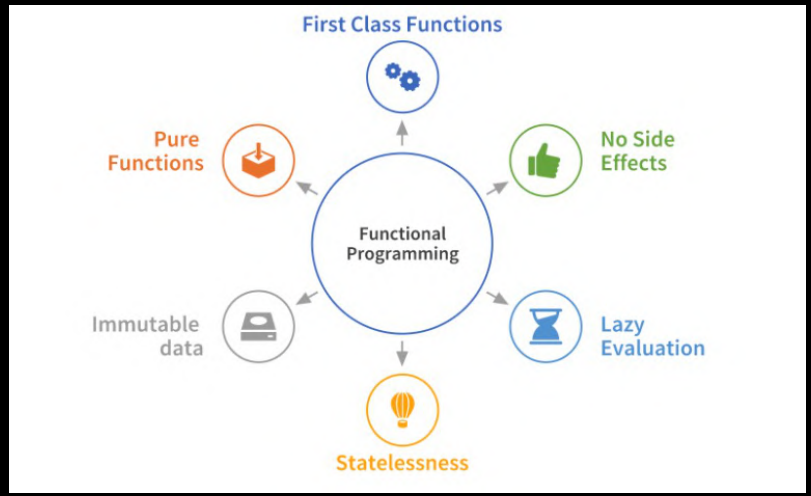
13 Functional Programming

1. What is Functional Programming
2. Lambda Expression
3. What is a Stream
4. Filtering & Reducing
5. Functional Interfaces
6. Method References
7. Functional vs Structural Programming
8. Optional Class
9. Intermediate vs Terminal Operations
10. Max, Min, Collect to List
11. Sort, Distinct, Map



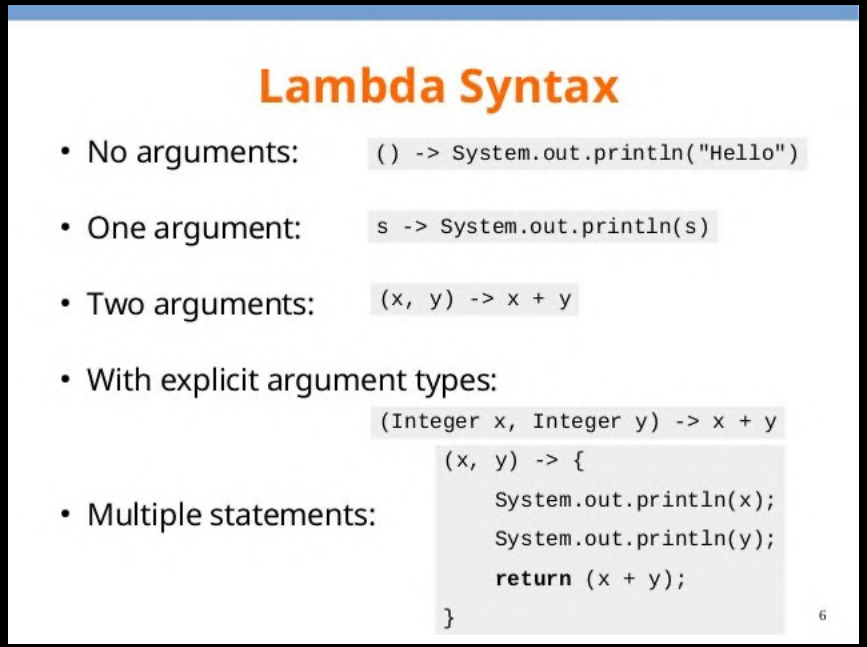
13.1 What is Functional Programming

1. Functional Programming: It's a way of writing programs where you use functions like small building blocks.
2. Functions as First-Class: Functions can be passed as arguments, returned from other functions, and assigned to variables.
3. Immutable Data: Once you create a piece of data, you don't change it.
4. Pure Functions: These are special functions that always give the same result for the same input with no Side-Effects.
5. Functional Interfaces: These are like templates for functions, making it easier to use them in different parts of your program.



13.2 Lambda Expression

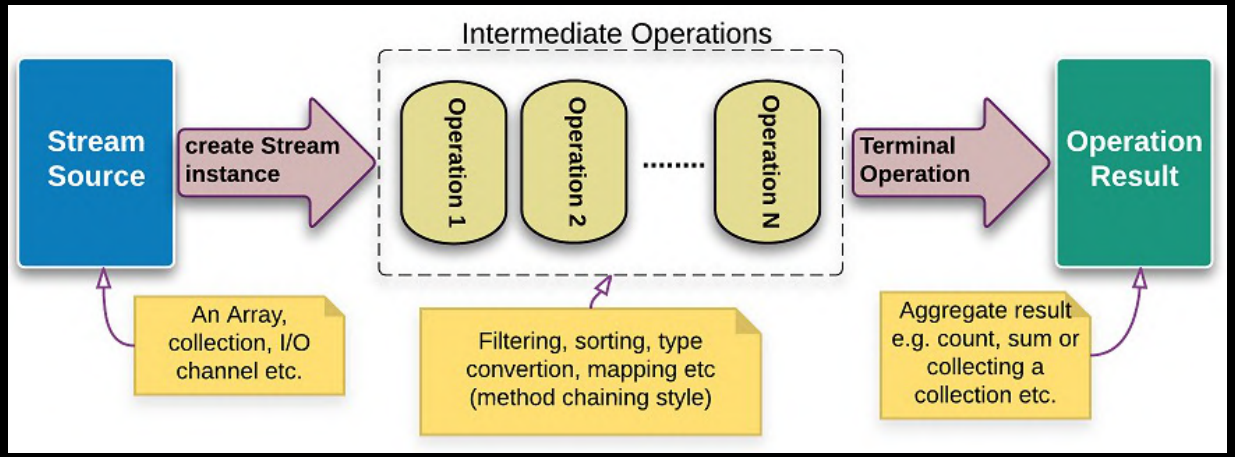
1. Shortcuts: Lambda expressions are quick, nameless functions for small tasks.
2. Syntax: Written as (parameters) -> {body}, linking inputs to actions.
3. Functional Interfaces: They work with interfaces that have only one method, making code concise.
4. Readability: They make code shorter and clearer, especially with collections.
5. Useful with Collections: Great for managing lists and sets, like filtering or sorting.



13.2 Lambda Expression

```
// Simple Addition:
(int a, int b) -> a + b;

// Check if a Number is Even:
(int number) -> number % 2 == 0;

// Print a Message:
(String message) -> System.out.println(message);

// Sort a List of Strings by Length:
(String s1, String s2) -> s1.length() - s2.length();

// Runnable with Lambda (No Parameters):
() -> System.out.println("Hello, World!");
```

13.3 What is a Stream



1. Element Sequence: Streams represent a sequence of elements
2. Functional Operations: Operations like map, filter, and reduce.
3. No Storage: Streams don't store data; they process it on-the-fly from sources like collections or arrays.
4. Efficiency: Stream operations can be lazy, processing elements only as needed, which is efficient for large data.
5. One-Time Use: Streams are consumable; once processed, they cannot be.
6. Parallel Capable: They support parallel processing, making operations faster by utilizing multiple threads.

13.4 Filtering & Reducing (Filter)

```
List<String> myList = List.of("apple", "banana", "cherry", "date");
myList.stream()
    .filter(s -> s.endsWith("e"))
    .forEach(s -> System.out.println(s));
```

1. Purpose: Used to filter elements of a stream based on a given predicate (a condition). Only elements that satisfy the condition are included in the resulting stream.
2. Lazy Operation: It's a lazy operation, meaning it's not executed until a terminal operation (like collect or forEach) is invoked on the stream.
3. Returns a Stream: filter itself returns a new stream with elements that match the predicate.

13.4 Filtering & Reducing (Reduce)

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
int sum = numbers.stream()
                 .reduce(0, (a, b) -> a + b);
// sum: 15
```

1. Purpose: Used to reduce the elements of a stream to a single value. It takes a binary operator as a parameter and applies it repeatedly, combining the elements of the stream.
2. Versatile: Can be used for summing, finding min or max, and combining elements in a myriad of ways.
3. Optional or Default Value: Without an identity value, reduce returns an Optional. With an identity value, it returns a default value if the stream is empty.

CHALLENGE

106. Write a lambda expression that takes two integers and returns their multiplication. Then, apply this lambda to a pair of numbers.
107. Convert an array of strings into a stream. Then, use the stream to print each string to the console.
108. Given a list of strings, use stream operations to filter out strings that have length of 10 or more and then concatenate the remaining strings.
109. Given a list of integers, use stream operations to filter odd numbers and print them.

13.5 Functional Interfaces

1. Single Abstract Method (SAM): A functional interface has only one abstract method. However, it can have multiple default or static methods.
2. Lambda Compatibility: They are intended to be used with lambda expressions, providing a target type for lambdas and method references.
3. @FunctionalInterface Annotation: While not mandatory, this annotation helps the compiler to identify the intention of making an interface functional and to generate an error if the annotated interface does not satisfy the conditions.
4. Common Examples: Predicate, Consumer, BinaryOperator, Runnable, Callable, Comparator, and user-defined interfaces can be functional if they have only one abstract method.

```
Predicate<Integer> isPositive = x -> x > 0;
// Output: true
System.out.println(isPositive.test(5));
// Output: false
System.out.println(isPositive.test(-5));

Consumer<String> print =
        message -> System.out.println(message);
// Output: Hello, World!
print.accept("Hello, World!");

BinaryOperator<Integer> multiply = (a, b) -> a * b;
// Output: 15
System.out.println(multiply.apply(5, 3));
```

13.6 Method References

Lambda Expression	Method Reference
`s -> s.toLowerCase()`	`String::toLowerCase`
`s.toLowerCase()`	`String::toLowerCase`
`(a, b) -> a.compareTo(b)`	For Integers it will be `Integer::compareTo` and for strings it is `String::compareTo`
`(a, b) -> Person.compareByAge(a, b)`	`Person::compareByAge`

Syntax:

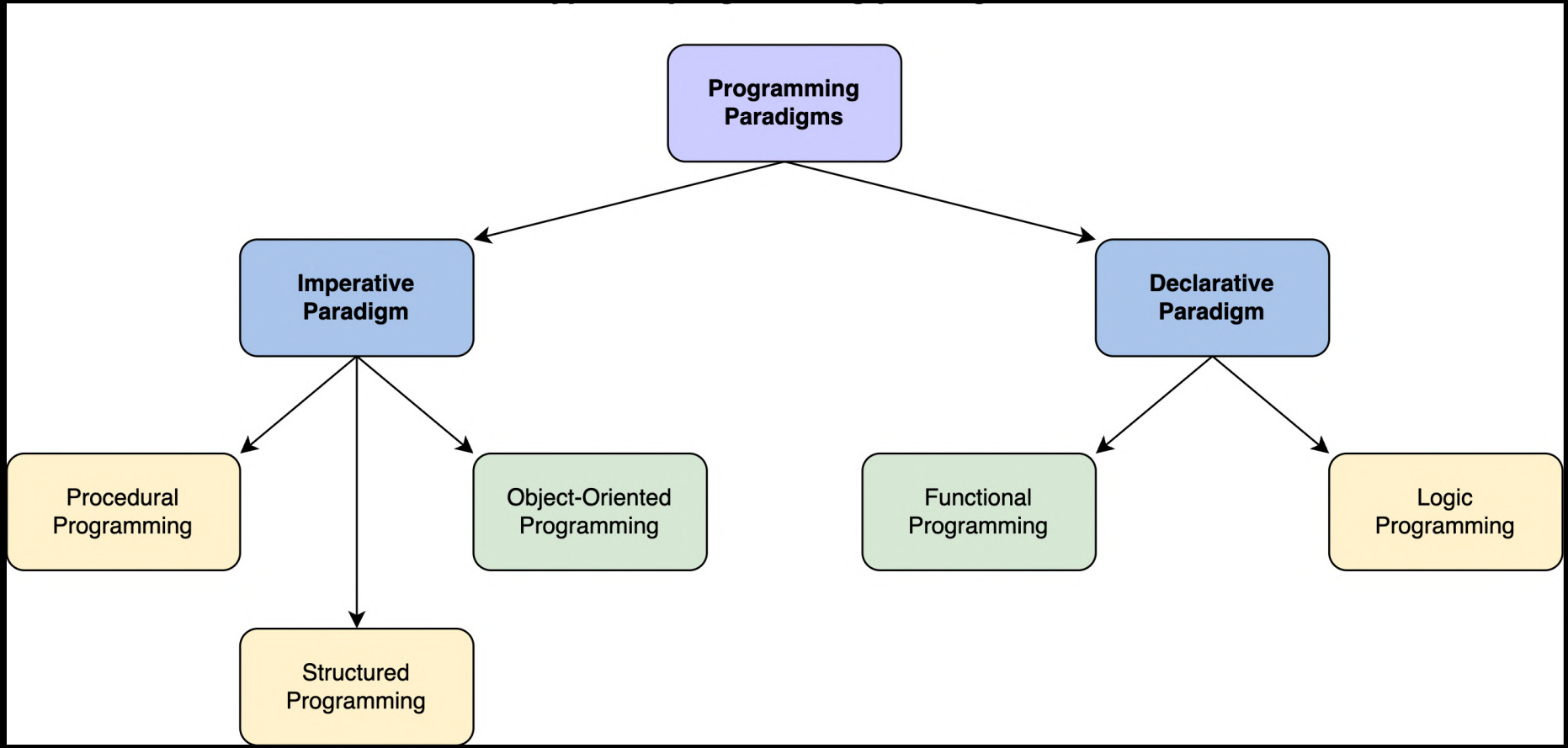
- Static Method References: ClassName::staticMethodName
- Instance Method: instance::instanceMethodName
- Instance Method Particular Class: ClassName::methodName
- Constructor References: ClassName::new.

1. Purpose Syntax & Usage: A method reference is described using :: (double colon) syntax. For example, System.out::println refers to the println method of the System.out object.
2. Functional Interfaces: They are used with functional interfaces.
3. Benefit: They make your code more readable and concise
4. Limitation: They can only be used for methods that fit the parameters and return type.

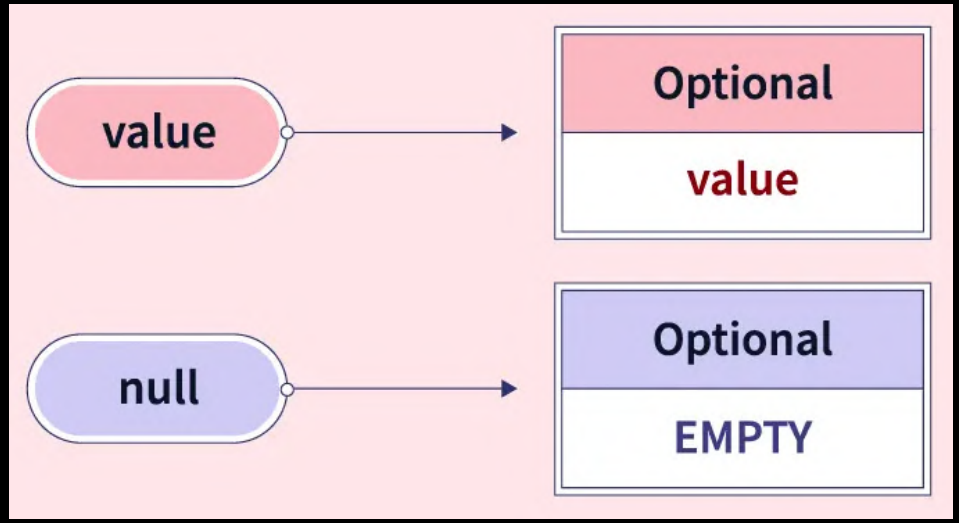
13.7 Functional vs Structural Programming

Imperative Programming	Declarative Programming
1. Computation	
You describe the step-by-step instructions for how an executed program achieves the desired results.	You set the conditions that trigger the program execution to produce the desired results.
2. Readability and complexity	
With the emphasis on the control flow, you can often follow the step-by-step process fairly easily. However, as you add more code, it can become longer and more complex	Step-by-step processes are eschewed. You'll discover that this paradigm is less complex and requires less code, making it easier to read.
3. Customization	
A straightforward way to customize and edit code and structure is offered. You have complete control and can easily adapt the structure of your program to your needs.	Customizing the source code is more difficult because of complicated syntax and the paradigm's dependence on implementing a pre-configured algorithm.
4. Optimization	
Adding extensions and making upgrades are supported, but doing so is significantly more challenging than with declarative programming, making it harder to optimize.	You can easily optimize code because an algorithm controls the implementation. Furthermore, you can add extensions and make upgrades.
5. Structure	
The code structure can be long and complex. The code itself specifies how it should run and in what order.	The code structure is concise and precise, and it lacks detail.

13.7 Functional vs Structural Programming



13.8 Optional Class



1. Creating Optional Objects: Optional.empty(), Optional.of(), Optional.ofNullable()
2. Checking Value Presence: isPresent() and ifPresent()
3. Default Values: orElse() and orElseGet()
4. Value Transformation: map()
5. Throwing Exception: orElseThrow()

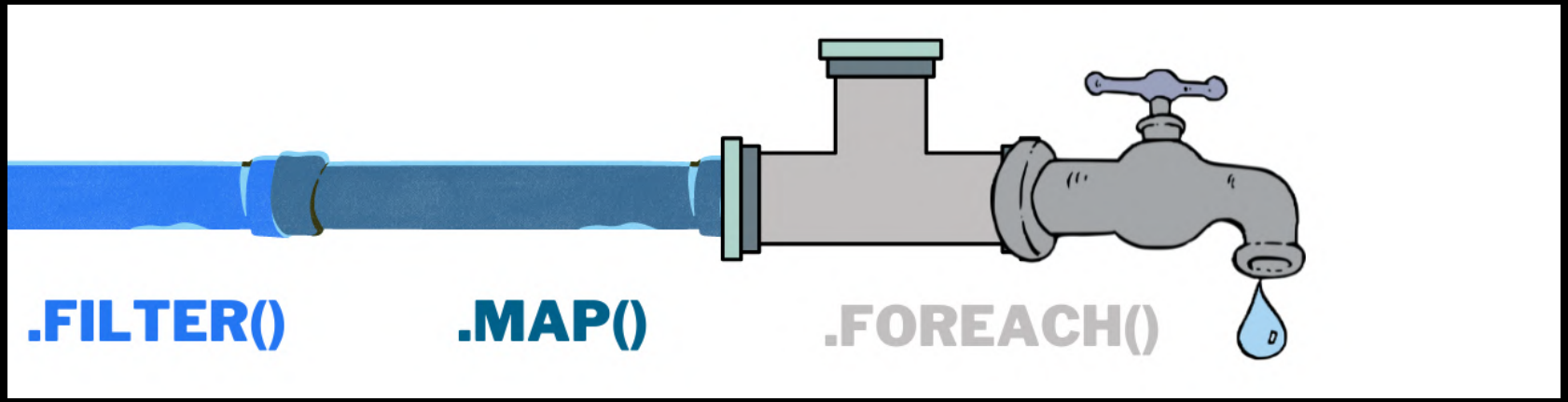
13.8 Optional Class

```java
// Creating Optional objects
Optional<String> optionalEmpty = Optional.empty();
Optional<String> optionalOf = Optional.of("Java");
Optional<String> optionalNullable = Optional.ofNullable(null);

// Checking presence of value
if (optionalOf.isPresent()) {
    System.out.println("Value is present: " + optionalOf.get());
}

// Using orElse to provide a fallback
String orElseExample = optionalEmpty.orElse("Default Value");
System.out.println("Using orElse: " + orElseExample);

// Using ifPresent to perform an action if value is present
optionalOf.ifPresent(System::out::println);
```

CHALLENGE

110. Create your own functional interface with a single abstract method that accepts an integer and returns a boolean. Implement it using a lambda that checks if the number is prime.
111. Write two versions of a program that calculates the factorial of a number: one using structural (procedural) programming, and the other using functional programming.
112. Write a function that accepts a string and returns an Optional<String>. If the string is empty or null, return an empty Optional, otherwise, return an Optional containing the uppercase version of the string.

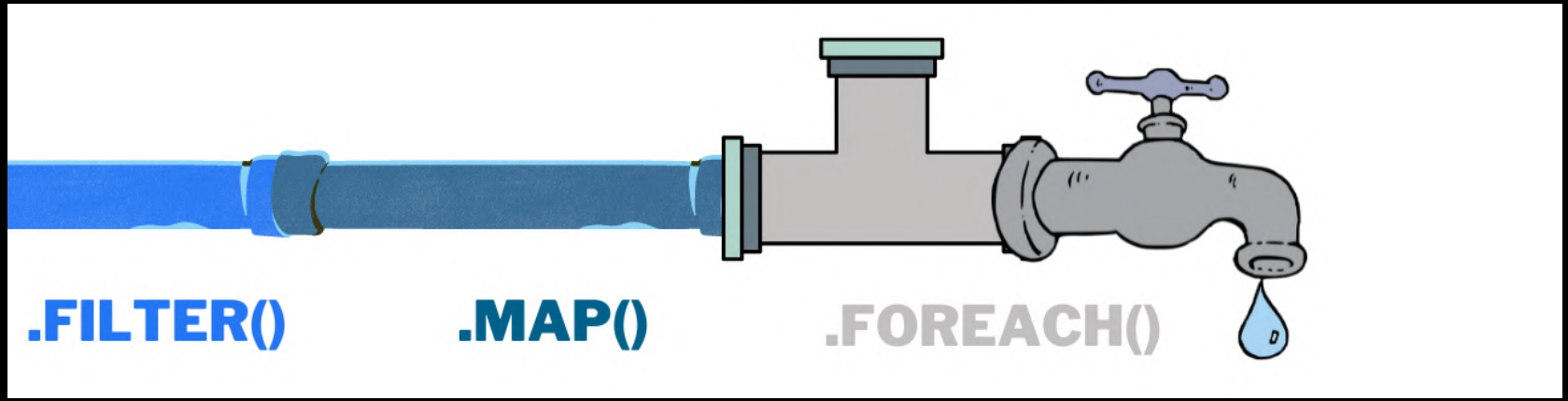
13.9 Intermediate vs Terminal Operations



Intermediate Operations

1. Laziness: Executed only when a terminal operation is invoked, setting up a pipeline without processing data.
2. Stream Transformation: Transform one stream into another, e.g., filter, map. They're chainable, allowing multiple transformations.
3. State Handling: Can be stateless (like map) or stateful(like sorted), affecting processing.

13.9 Intermediate vs Terminal Operations



Terminal Operations

1. Computation Trigger: Initiates the stream processing and closes the stream. After this, the stream can't be reused.
2. Final Outcome: Produces a result (like a sum or list) or a side-effect (like printing each element). Not chainable.
3. Examples: Operations like collect, forEach, reduce, sum, max, min, and count are terminal.

13.10 Max, Min, Collect to List

```java
List<Integer> numbers = List.of(4, 2, 5, 1, 3);
Optional<Integer> maxNumber = numbers.stream()
    .max(Integer::compareTo);
// Output: 5
maxNumber.ifPresent(System.out::println);

List<Integer> numbers = List.of(4, 2, 5, 1, 3);
Optional<Integer> minNumber = numbers.stream()
    .min(Integer::compareTo);
// Output: 1
minNumber.ifPresent(System.out::println);

List<String> words = Arrays.asList("Stream",
    "Operations", "Java");
List<String> collectedWords = words.stream()
    .collect(Collectors.toList());
// Output: [Stream, Operations, Java]
System.out.println(collectedWords);
```

1. max() finds the largest element in the stream according to a given comparator or natural ordering.
2. min() identifies the smallest element in the stream based on a provided comparator or natural ordering.
3. collect(Collectors.toList()) gathers all the elements of the stream into a new List.

13.11 Sort, Distinct, Map

```java
List<Integer> numbers = List.of(4, 2, 5, 1, 3);
List<Integer> sortedNumbers = numbers.stream()
    .sorted()
    .collect(Collectors.toList());
// Output: [1, 2, 3, 4, 5]
System.out.println(sortedNumbers);


List<String> items = List.of("apple",
    "banana", "apple", "orange", "banana");
List<String> distinctItems = items.stream()
    .distinct()
    .collect(Collectors.toList());
// Output: [apple, banana, orange]
System.out.println(distinctItems);

List<String> words = List.of("Stream",
    "Operations", "Java");
List<String> uppercaseWords = words.stream()
    .map(String::toUpperCase)
    .collect(Collectors.toList());
// Output: [STREAM, OPERATIONS, JAVA]
System.out.println(uppercaseWords);
```

1. sorted() orders the elements of a stream based on their natural order or a provided comparator.
2. distinct() filters out duplicate elements, ensuring that every element in the resulting stream is unique.
3. map() applies a function to each element of a stream, transforming them into a new stream of results based on the function logic.

CHALLENGE

113. Given an array of integers, create a stream, use the distinct operation to remove duplicates, and collect the result into a new list.
114. Create a list of employees with name and salary fields. Write a comparator that sorts the employees by salary. Then, use this comparator to sort your list using the sort stream operation.
115. Create a list of strings representing numbers ("1", "2", ...). Convert each string to an integer, then again calculating squares of each number using the map operation and sum up the resulting integers.

Revision

1. What is Functional Programming
2. Lambda Expression
3. What is a Stream
4. Filtering & Reducing
5. Functional Interfaces
6. Method References
7. Functional vs Structural Programming
8. Optional Class
9. Intermediate vs Terminal Operations
10. Max, Min, Collect to List
11. Sort, Distinct, Map

Practice Exercise

Functional Programming

Answer in True/False

1. Functions can't be assigned to variables, passed as arguments, or returned from other functions.
2. A lambda expression in Java can be used to implement any interface, regardless of the number of abstract methods.
3. A Java stream represents a sequence of elements and supports various methods which can be pipelined to produce the desired result.
4. The filter method in streams is a terminal operation that returns a boolean value.
5. A functional interface in Java is an interface with exactly one abstract method.
6. Method references in Java can only refer to static methods.
7. The Optional class in Java is used to avoid NullPointerException.
8. Intermediate operations on streams are executed immediately and are always followed by terminal operations.
9. The max and min operations on streams return an Optional describing the maximum or minimum element.
10. The sorted operation in streams is a terminal operation and it sorts the elements of the stream in their natural order.

Practice Exercise

Functional Programming

Answer in True/False

1. Functions can't be assigned to variables, passed as arguments, or returned from other functions. — False
2. A lambda expression in Java can be used to implement any interface, regardless of the number of abstract methods. — False
3. A Java stream represents a sequence of elements and supports various methods which can be pipelined to produce the desired result. — True
4. The filter method in streams is a terminal operation that returns a boolean value. — False
5. A functional interface in Java is an interface with exactly one abstract method. — True
6. Method references in Java can only refer to static methods. — False
7. The Optional class in Java is used to avoid NullPointerException. — True
8. Intermediate operations on streams are executed immediately and are always followed by terminal operations. — False
9. The max and min operations on streams return an Optional describing the maximum or minimum element. — True
10. The sorted operation in streams is a terminal operation and it sorts the elements of the stream in their natural order. — False