23IT405: Java Programming

Topic: History of Java

Department of Information Technology

Birth of Java

- Year: 1991
- Initiators: James Gosling and team at Sun Microsystems.
- Project Name: "Oak“
- Objective: Create a platform-independent language for embedded systems.

Java's Early Days

- Year: 1995
- Renamed: From "Oak" to "Java”
- Reason: Trademark issues and inspiration from Java coffee.
- Launch: First public release of Java (JDK 1.0).

Key Features at Launch

- Platform Independence (Write Once, Run Anywhere - WORA).
- Object-Oriented Programming.
- Automatic Garbage Collection.
- Robustness and Security.

Evolution of Java

- 1998: JDK 1.2 - Introduction of "Swing" and "Collections Framework”.
- 2004: JDK 5.0 - Added Generics and Enhanced for-loop.
- 2014: Java 8 - Introduced Lambdas and Streams.

Challenges and Criticism

- Verbose syntax compared to modern languages.
- Performance overhead of JVM.
- Competition from newer languages like Python and Kotlin.

The Future of Java

- Continued evolution with regular updates.
- Integration with emerging technologies like AI and IoT.

23IT405: Java Programming

Topic: Need for OOP Paradigm

Department of Information Technology

Programming and Programming Language

A process of creating a set of instructions for a computer to perform tasks.

A programming language is a formal language used to communicate instructions to a computer in software development.

Introduction to Java

How Java was Introduced?

What is Object-Oriented Programming (OOP)?

Definition: OOP is a programming paradigm based on objects representing real-world entities.

Purpose: Simplifies system design with modularity and reusability.

Real-life Analogy: A **car** object has:

Attributes: color, model, engine size.

Behaviors: accelerate, brake, turn.

Limitations of Procedural Programming (C as an Example)

Challenges:

- Poor scalability for large systems.
- Code duplication and low reusability.

Example:

- **Library System** in C:
  - Functions like addBook() and removeBook() are standalone and not linked to specific objects, leading to redundant code.

Why Transition from C to Java?

C:

- Procedural programming language.
- Focuses on functions and processes.
- Suitable for system-level programming like OS development.

Java:

- Object-oriented programming language.
- Models real-world entities using objects.
- Platform-independent and ideal for large-scale applications.

Core Differences Between C and Java

Feature	C	Java
Paradigm	Procedural	Object-Oriented
Platform Dependence	Platform-dependent	Platform-independent (WORA)
Memory Management	Manual	Automatic (Garbage Collection)
Pointers	Supports pointers	No direct pointer access
Inheritance	Not supported	Fully supported
Application	System programming (OS, drivers)	Web, mobile, enterprise apps

Example:

- **C**: Focuses on writing functions like void calculateArea(int length, int breadth).
- **Java**: Uses objects like Rectangle with attributes (length, breadth) and methods (calculateArea()).

Java’s Key Features

Platform Independence:	Runs on any device with JVM.
Memory Management:	Garbage collection prevents memory leaks.
Security:	Built-in features like bytecode verification and class loaders.
Applications:	Web apps, Android apps, and enterprise systems.
Real-life Usage:	Powers Netflix, LinkedIn, and Spotify.

Advantages of Java and OOP

Modularity: Example: E-commerce platform with modular classes for Product, Cart, Payment.

Reusability: Example: Employee class reused across departments.

Maintainability: Example: Debugging in a single class propagates fixes to derived classes.

Scalability: Used by Amazon and LinkedIn to handle millions of users.

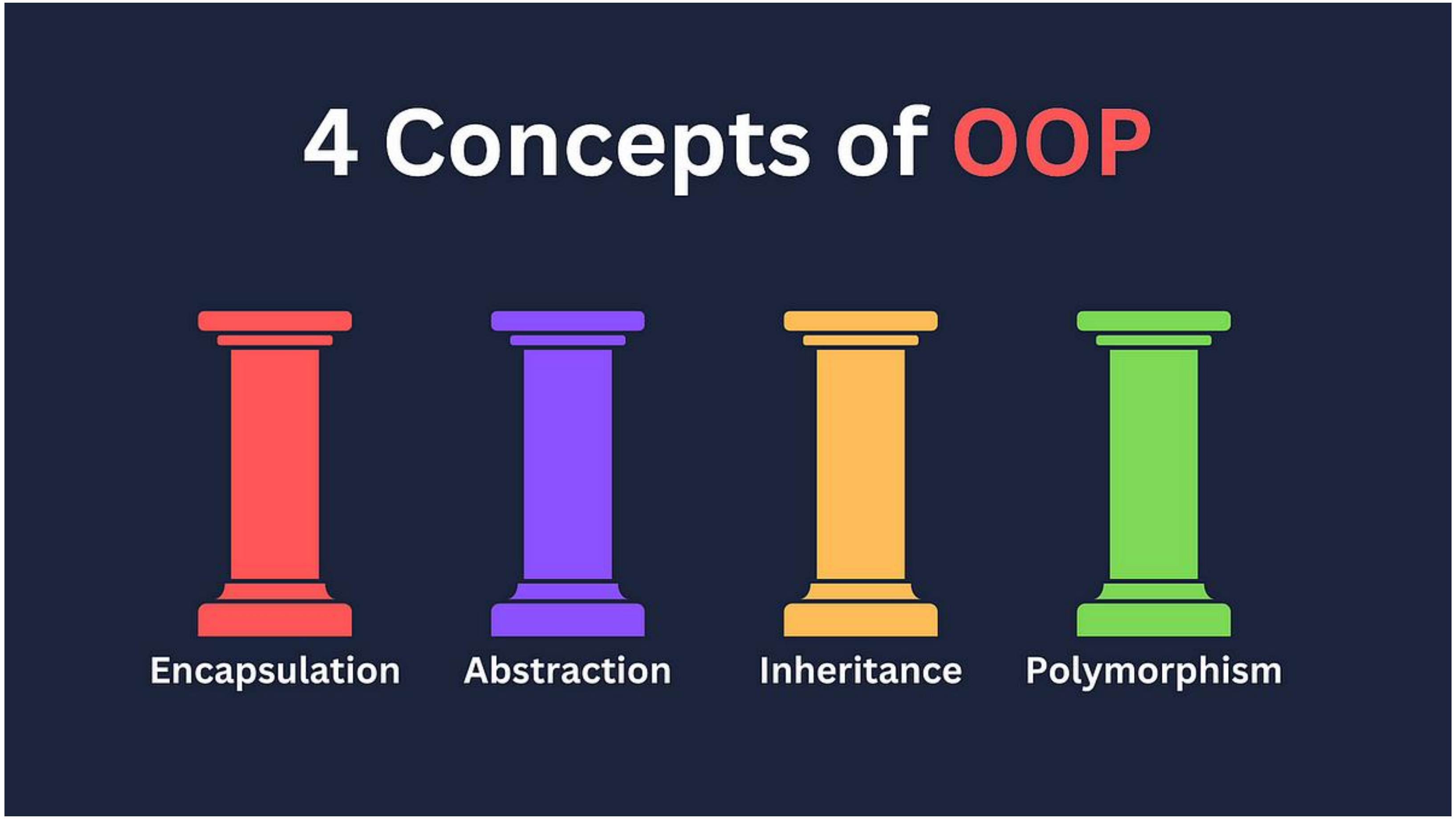
23IT405: Java Programming

Topic: Need for OOP Paradigm

Department of Information Technology

Four Pillars of OOP

- **Abstraction**: Hiding complexity and exposing only essential features
- **Inheritance**: Creating new classes from existing ones
- **Polymorphism**: Using a single interface to represent different types


4 Concepts of OOP
Encapsulation
Abstraction
Inheritance
Polymorphism

Objects and Classes

Class:

Blueprint for creating objects; defines attributes and methods.

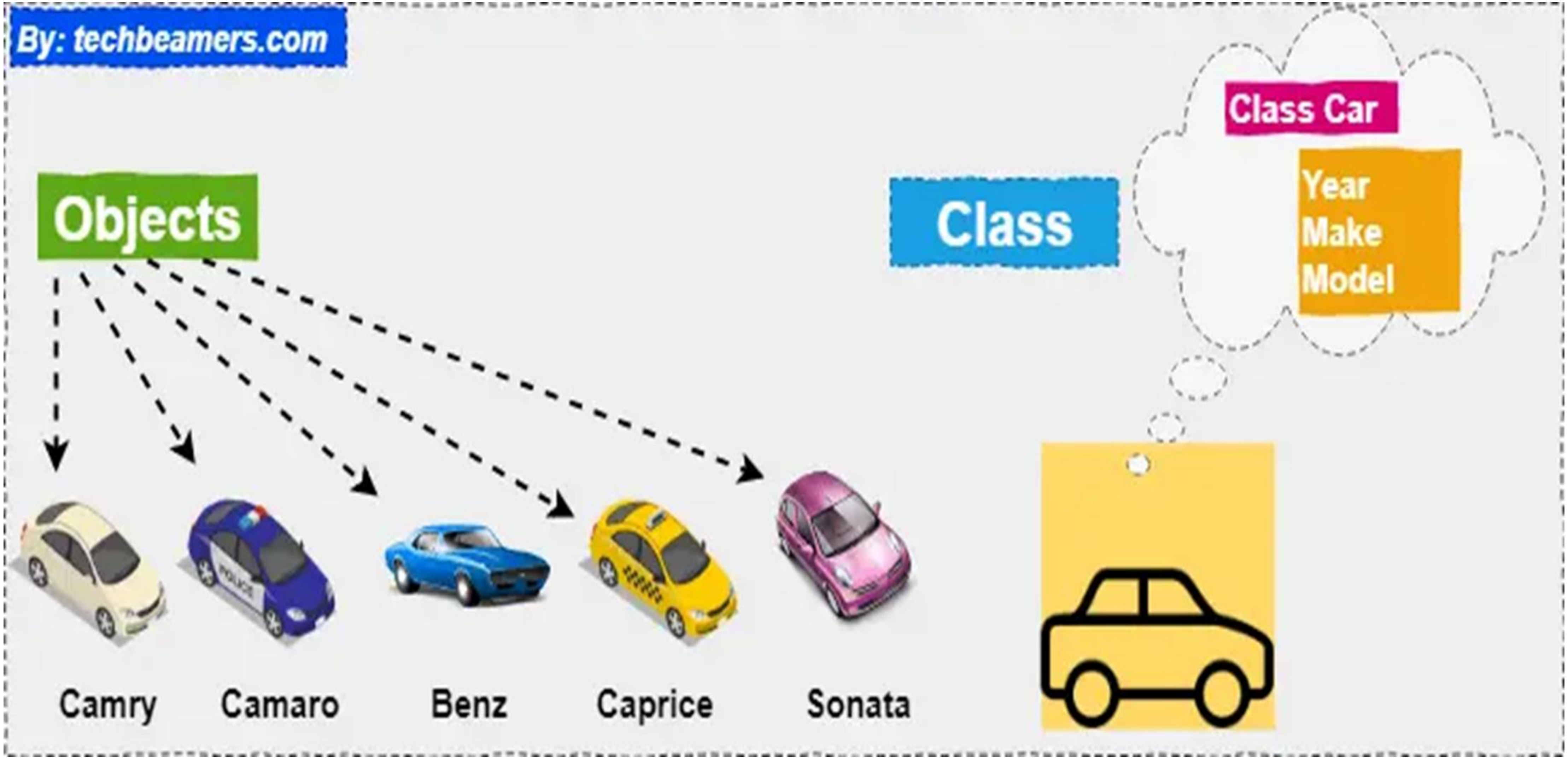
Object:

Instance of a class; represents real-world entities.

Real-world Scenario:

Class: "Car" with attributes like brand, model, and methods like start, stop.

Object: Specific car instance (e.g., "Toyota Corolla 2022").

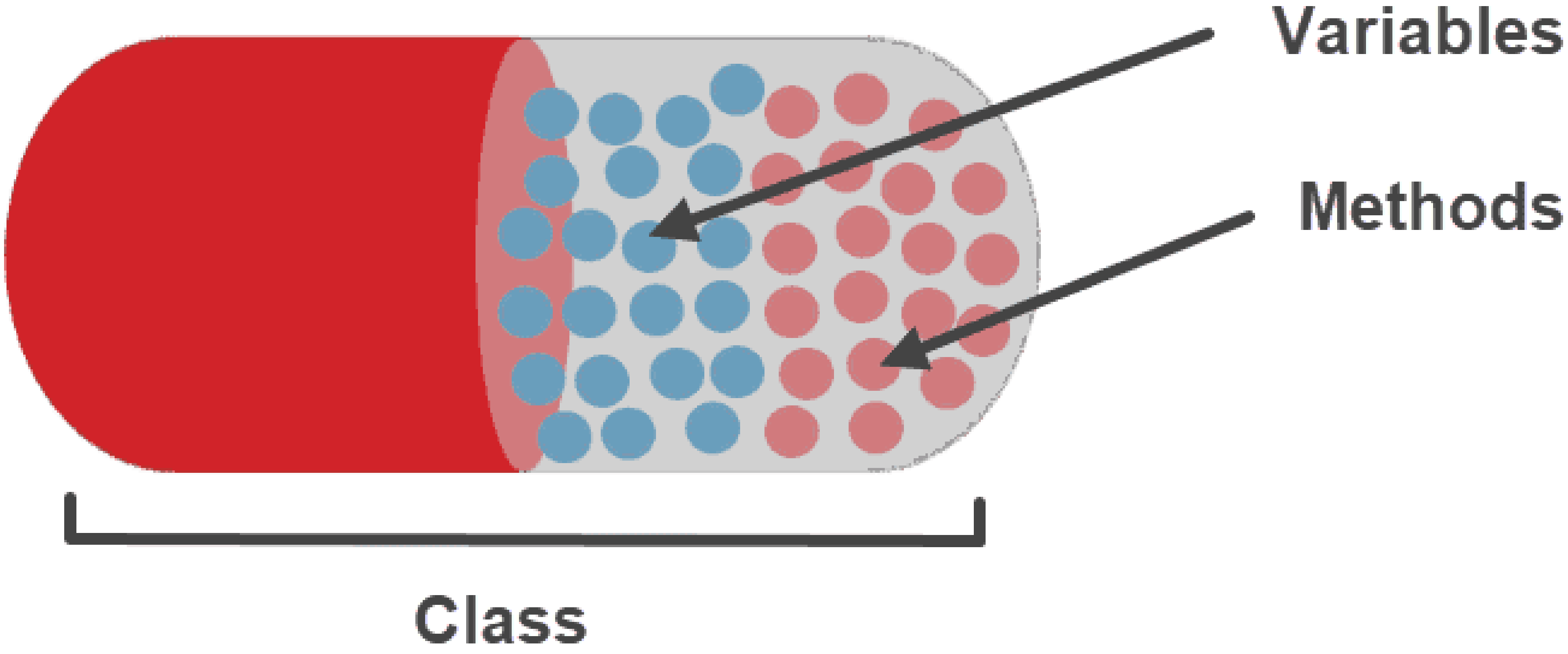

By: techbeamers.com
Objects
Class
Class Car
Year
Make
Model
Camry
Camaro
Benz
Caprice
Sonata

Encapsulation

- Combines data and methods into a single unit (class)
- Access control using access modifiers (private, public, protected)
- Example:

Real-world scenario: A bank account hides sensitive details like account number and balance but provides access through secure methods like depositing or withdrawing money.

ENCAPSULATION



Abstraction

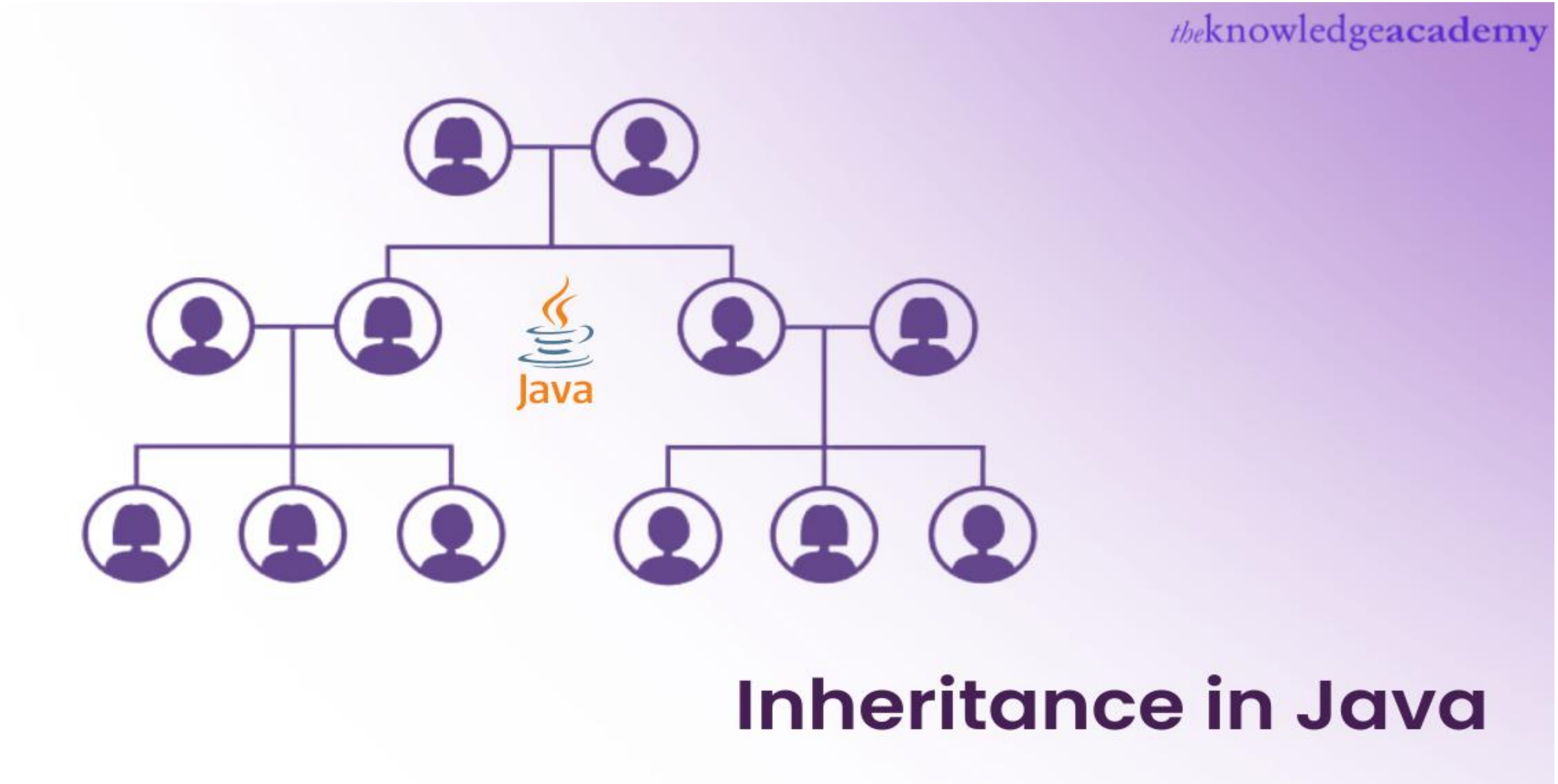
- Focus on "what" an object does, not "how"
- Implementation details are hidden from the user
- Example:

Real-world scenario: When using an ATM, users only interact with a simplified interface (insert card, enter PIN, withdraw cash) without knowing the underlying technical processes.

Inheritance

- Mechanism to derive a new class from an existing one
- Reuses code, promotes consistency
- Example:

Real-world scenario: A vehicle classification where a general "Vehicle" category has common attributes (e.g., engine type, wheels), and specific types like "Car" or "Truck" inherit these attributes while adding unique features.

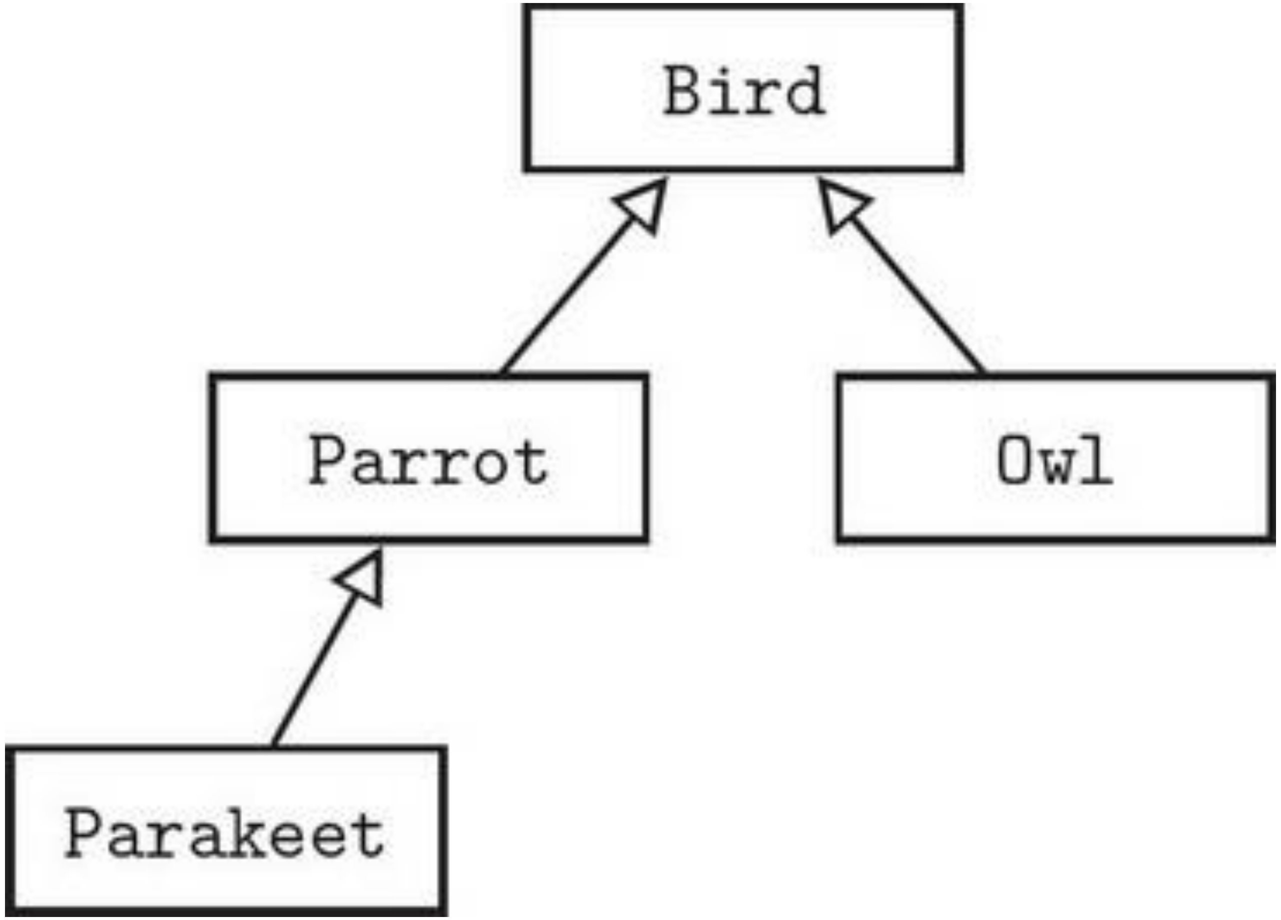


theknowledgeacademy
Java
Inheritance in Java

Polymorphism

- Objects of different classes can be treated as objects of a common superclass
- Two types: Compile-time and Run-time polymorphism
- Example:

Real-world scenario: A universal remote can operate various devices (TV, AC, DVD player) through a consistent set of buttons, but the specific actions depend on the device.


Bird
Parrot
Owl
Parakeet

Real-world Examples

- Banking Systems: Accounts, transactions, users
- Gaming: Characters, behaviors, environments
- E-commerce: Products, users, orders

23IT405: Java Programming

Topic: Java Buzzwords

Department of Information Technology

What are Java Buzzwords?

- Buzzwords are terms used to describe the design philosophy and features of Java.
- These define why Java became a preferred programming language.

1.Simple "Easy to Learn and Use"

- Intuitive syntax similar to C++.
- Eliminates complex concepts like pointers and multiple inheritance.

2.Object-Oriented "Everything is an Object"

- Follows principles of encapsulation, inheritance, and polymorphism.
- Promotes code reuse and modularity.

3.Portable "Write Once, Run Anywhere"

- Bytecode compiled by the Java compiler can run on any platform with a JVM.
- Independence from hardware or operating systems.

4.Platform-Independent "Runs Across Devices"

- Code compiled on one system can execute on another.
- Ensures consistency across environments.

5.Secured "Safe from Vulnerabilities"

- No explicit pointers.
- Robust security features like bytecode verification, class loader, and security manager.

6.Robust "Handles Errors Gracefully"

- Automatic garbage collection prevents memory leaks.
- Exception handling mechanisms ensure stability.
- Strong type checking during compilation.

7.Multithreaded "Enables Concurrent Execution"

- Supports threads for performing multiple tasks simultaneously.
- Simplifies interactive applications like games and multimedia.

8.High Performance "Optimized for Speed"

- JIT (Just-In-Time) compiler enhances execution speed.
- Efficient memory management through garbage collection.

9.Architecture Neutral "Independent of Underlying Systems"

- Designed to be architecture-agnostic.
- Ensures a consistent runtime environment.

10.Distributed "Built for Networked Applications"

- Facilitates distributed computing using technologies like RMI and EJB.
- Supports internet-based applications seamlessly.

11.Dynamic "Adapts to Evolving Needs"

- Loads classes at runtime as needed.
- Simplifies upgrades and integrates new libraries.

23IT405: Java Programming

Topic: Variables and Data Types

Department of Information Technology

What are Variables?

- Variables are containers for storing data values.
- Each variable has:

Name: Identifier used in code.

Type: Defines the kind of data it can hold.

Value: Actual data assigned to the variable.

Ex: int age;

Declaring Variables in Java

How to Declare Variables?

Syntax : datatype variableName ;

Example: double salary

Rules:

Variable names must start with a letter, $, or _.Cannot be a keyword or contain spaces.

Variable Types

- **Local Variables:** Declared inside methods, constructors, or blocks.
- **Instance Variables:** Declared inside a class but outside methods.
- **Static Variables:** Declared with the static keyword.

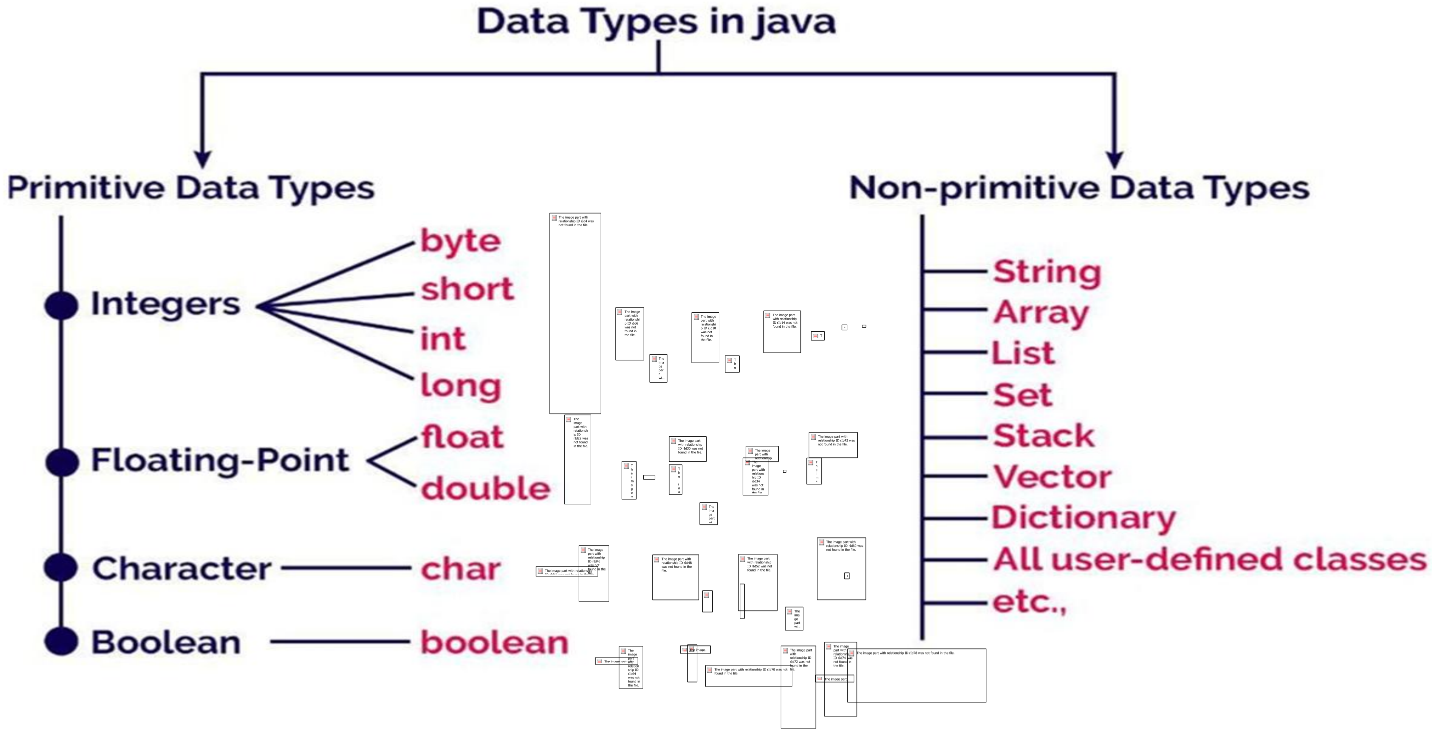
Data Types in Java

- Java is a statically typed language.
- Two categories:

Primitive Data Types: byte, short, int, long, float, double, char, boolean.

Reference Data Types: Objects, Arrays, etc.

JAVA DATA TYPES



Primitive Data Types

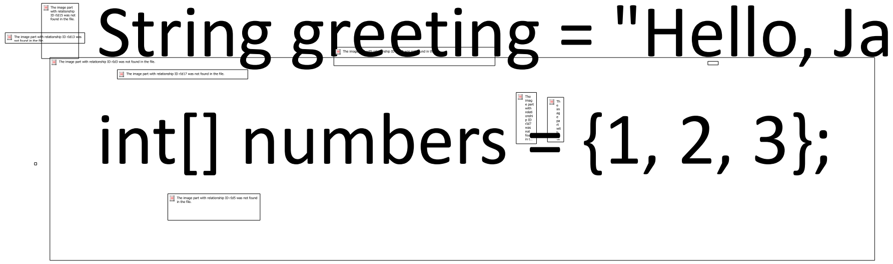
- Numeric Types:
  - Integer: byte, short, int, long
  - Floating-point: float, double
- Character Type:

  char
- Boolean Type:
  - boolean (true/false)

Reference Data Types

- Used to store references to objects.
- Examples: Arrays, Strings, User-defined classes

```
String greeting = "Hello, Java!";
int[] numbers = {1, 2, 3};
```



23IT405: Java Programming

Topic: Operators and Expressions

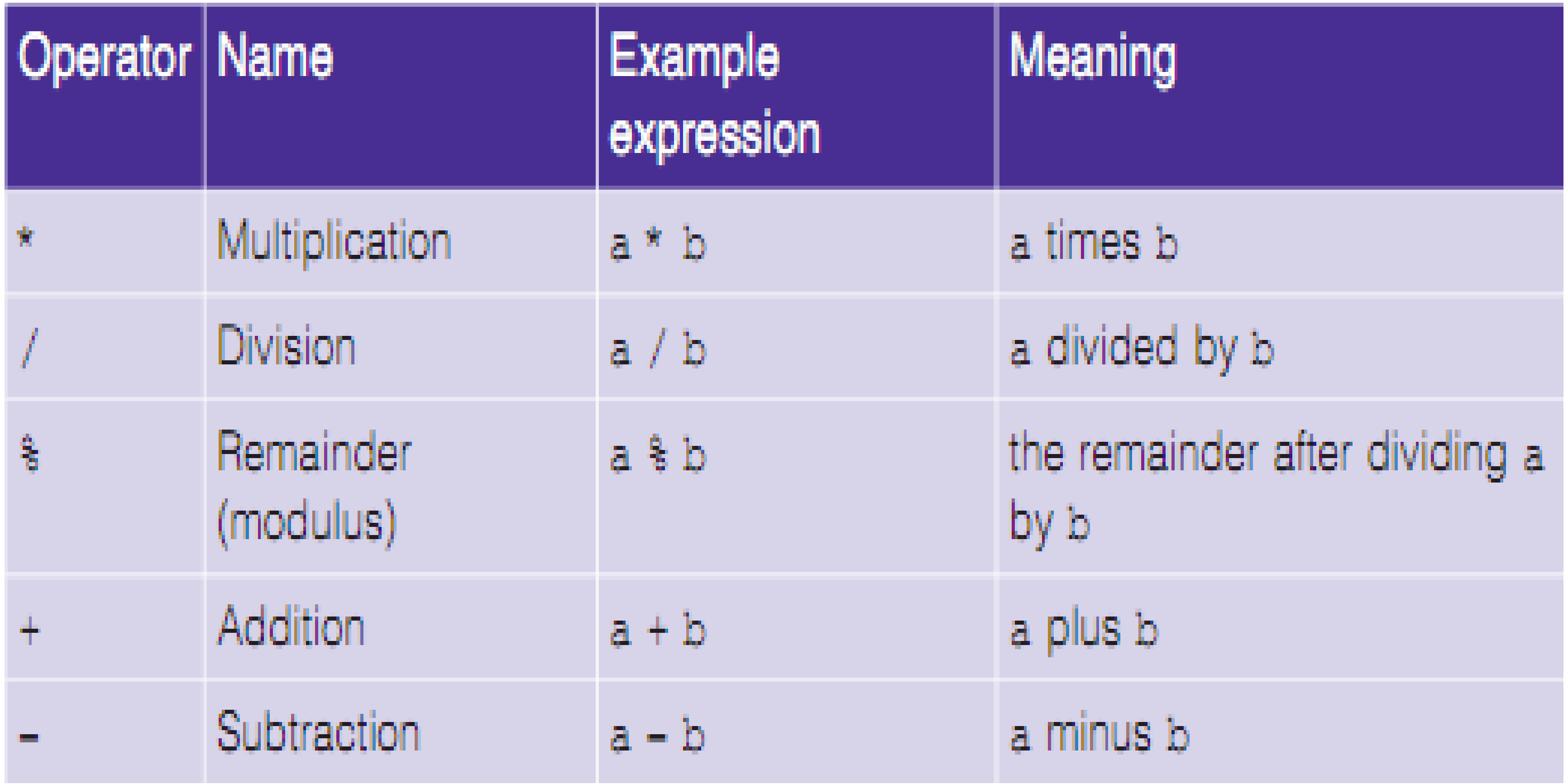
Department of Information Technology

JAVA OPERATORS

An operator is a symbol used to perform arithmetic and logical operations. Java provides a rich set of operators. In java, operators are classified into the following types.

- Arithmetic Operators
- Relational (or) Comparison Operators
- Logical Operators
- Assignment Operators
- Bitwise Operators
- Conditional Operators

Arithmetic Operators

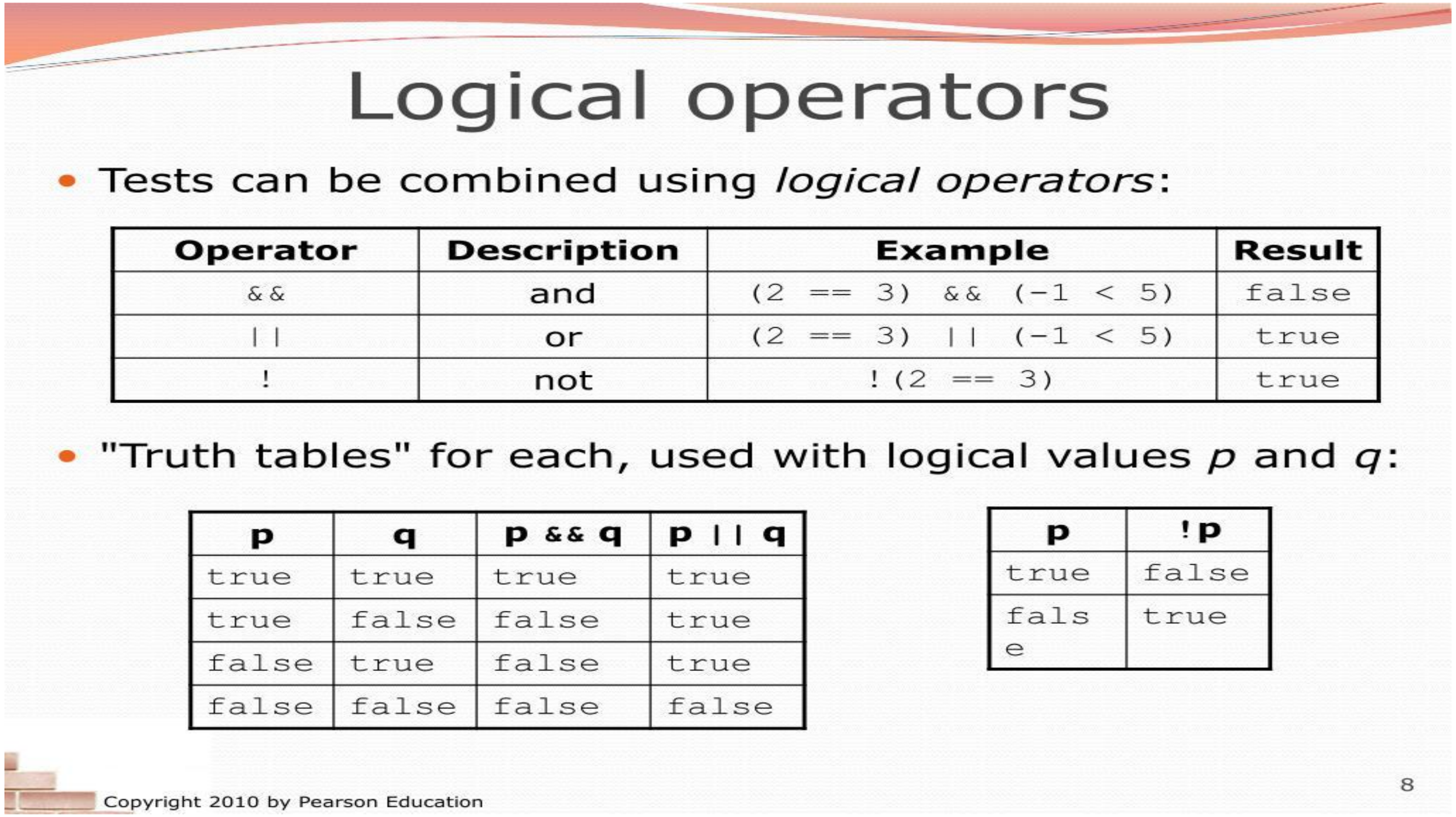


Operator	Name	Example expression	Meaning
*	Multiplication	a * b	a times b
/	Division	a / b	a divided by b
%	Remainder (modulus)	a % b	the remainder after dividing a by b
+	Addition	a + b	a plus b
-	Subtraction	a - b	a minus b

Relational Operators (<, >, <=, >=, ==, !=)

Operator	**Meaning**	**Example**
<	Returns TRUE if the first value is smaller than second value otherwise returns FALSE	10 < 5 is FALSE
>	Returns TRUE if the first value is larger than second value otherwise returns FALSE	10 > 5 is TRUE
<=	Returns TRUE if the first value is smaller than or equal to second value otherwise returns FALSE	10 <= 5 is FALSE
>=	Returns TRUE if the first value is larger than or equal to second value otherwise returns FALSE	10 >= 5 is TRUE
==	Returns TRUE if both values are equal otherwise returns FALSE	10 == 5 is FALSE
!=	Returns TRUE if both values are not equal otherwise returns FALSE	10 != 5 is TRUE

Logical Operators



Logical operators

- Tests can be combined using *logical operators*:

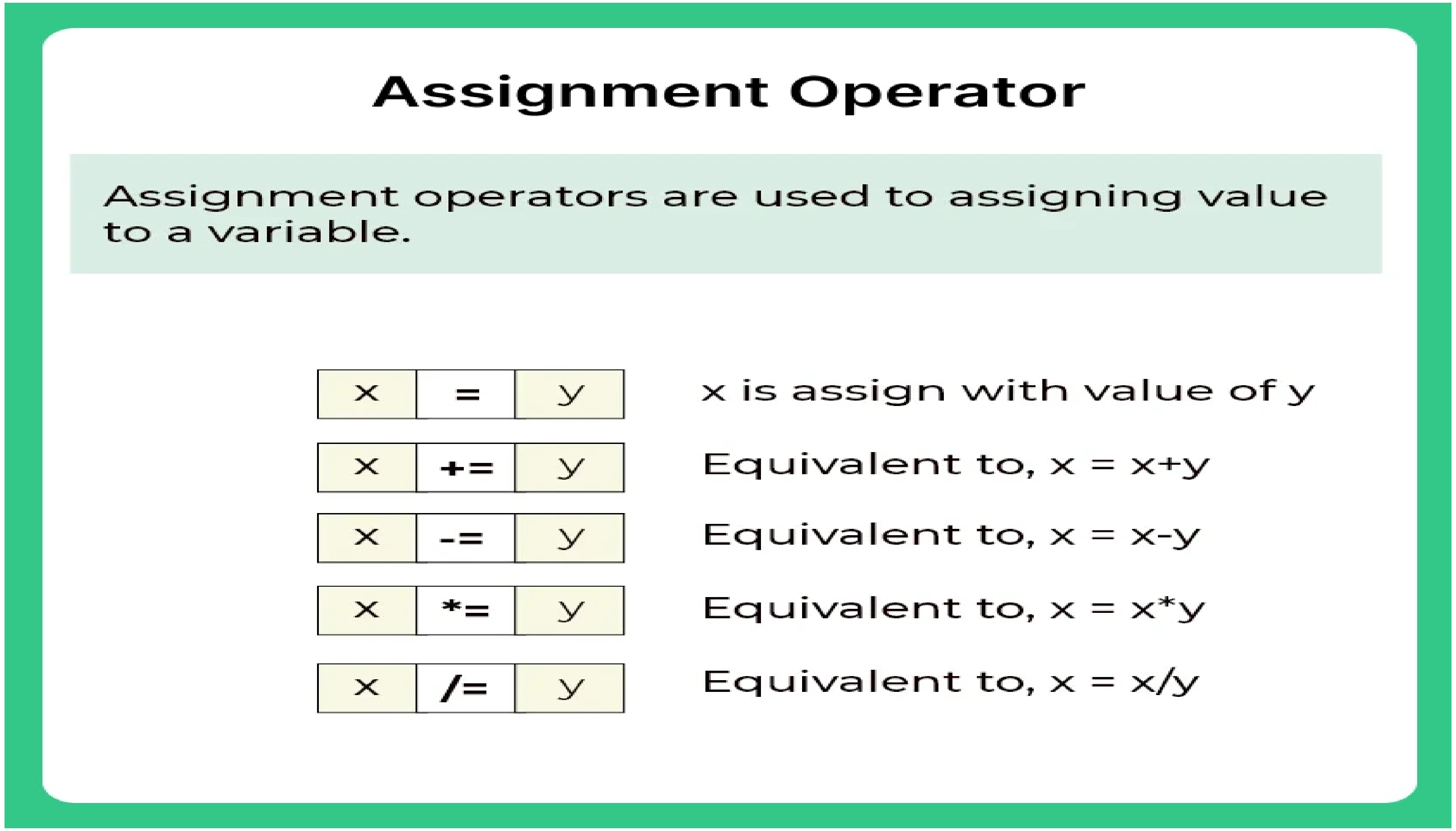
Operator	Description	Example	Result
`&&`	and	`(2 == 3) && (-1 < 5)`	`false`
`||`	or	`(2 == 3) || (-1 < 5)`	`true`
`!`	not	`!(2 == 3)`	`true`

- "Truth tables" for each, used with logical values *p* and *q*:

p	**q**	**p && q**	**p || q**
`true`	`true`	`true`	`true`
`true`	`false`	`false`	`true`
`false`	`true`	`false`	`true`
`false`	`false`	`false`	`false`

p	**!p**
`true`	`false`
`false`	`true`

Assignment Operators

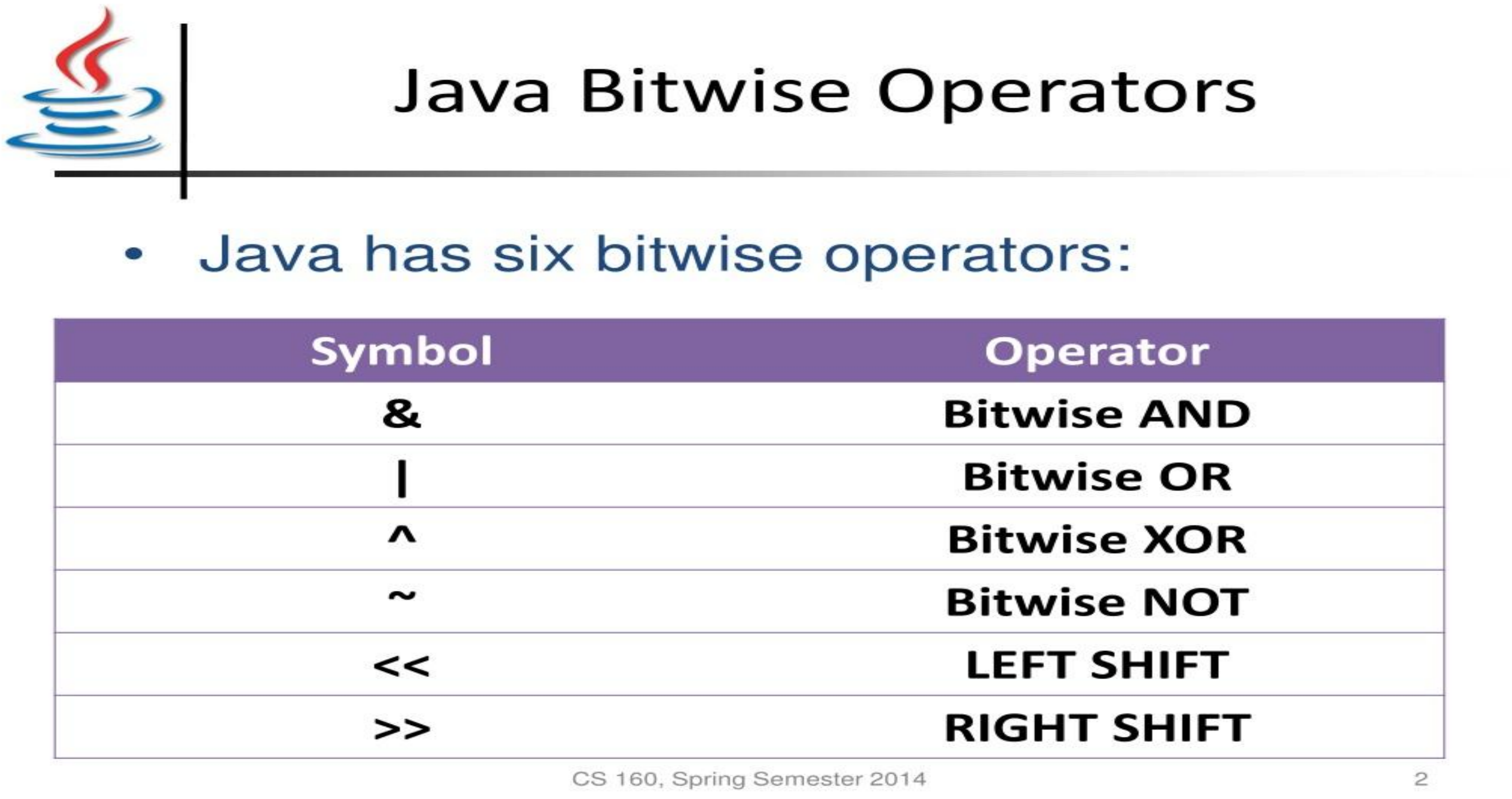


Assignment Operator

Assignment operators are used to assigning value to a variable.

x	=	y	x is assign with value of y
x	+=	y	Equivalent to, x = x+y
x	-=	y	Equivalent to, x = x-y
x	*=	y	Equivalent to, x = x*y
x	/=	y	Equivalent to, x = x/y

Bitwise Operators



Java Bitwise Operators

- Java has six bitwise operators:

Symbol	Operator
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
~	Bitwise NOT
<<	LEFT SHIFT
>>	RIGHT SHIFT

CS 160, Spring Semester 2014 2

Conditional Operator

- The conditional operator is also called a ternary operator because it requires three operands.
- This operator is used for decision making. In this operator, first, we verify a condition, then we perform one operation out of the two operations based on the condition result.
- If the condition is TRUE the first option is performed, if the condition is FALSE the second option is performed.
- Syntax
- Condition ? TRUE Part : FALSE Part;

Expressions

- An expression is a combination of variables, constants, operators, and method calls that evaluates to a single value.
- Used to perform computations and logic in a program.
- Example: int result = 10 + 20;

Types of Expressions

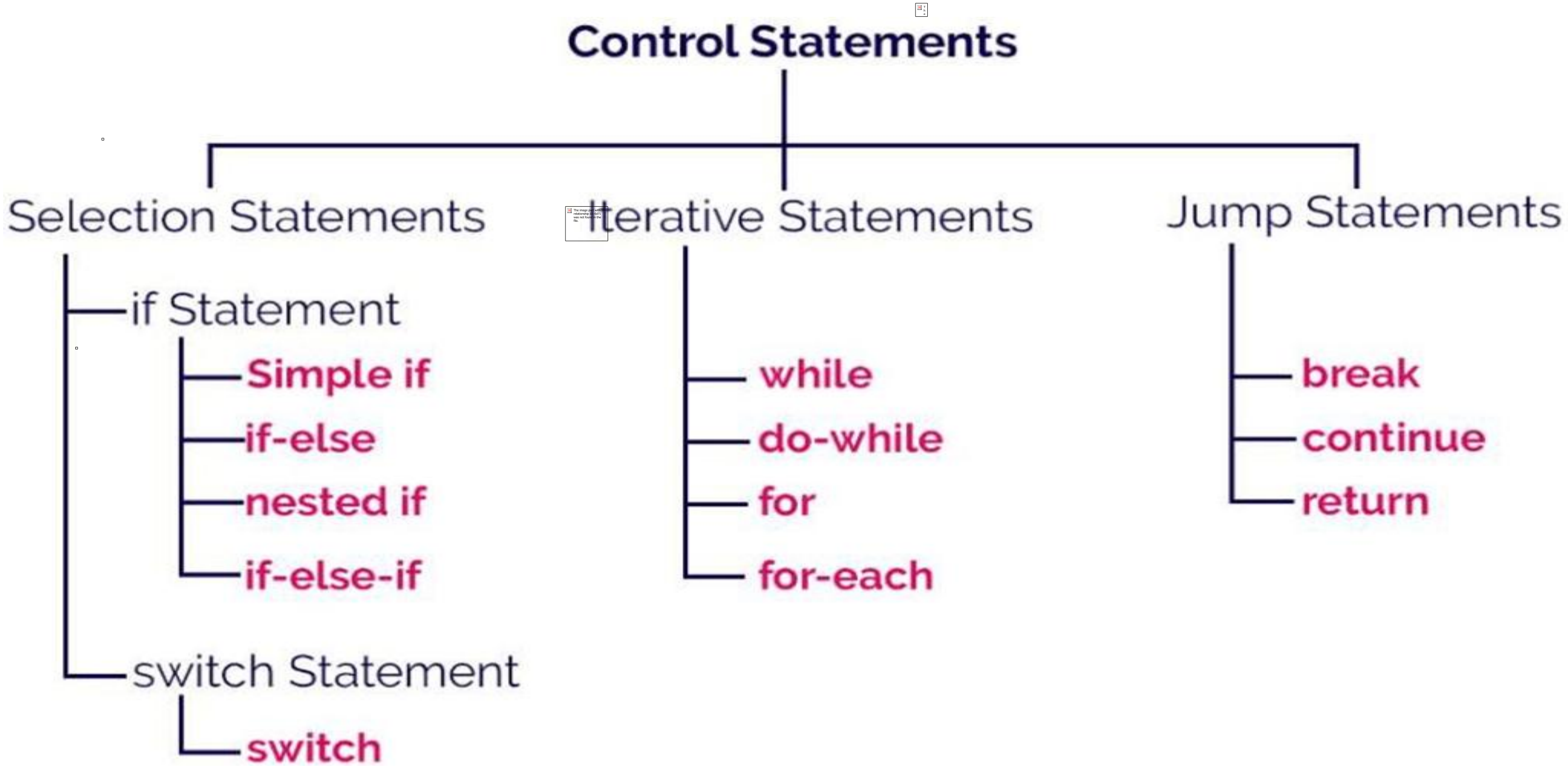
- Arithmetic Expressions: Perform mathematical calculations.
- Relational Expressions: Compare values.
- Logical Expressions: Combine boolean conditions.
- Bitwise Expressions: Perform bit-level operations.
- Assignment Expressions: Assign values to variables.

23IT405: Java Programming

Topic: Control Statements

Department of Information Technology

JAVA CONTROL STATEMENTS

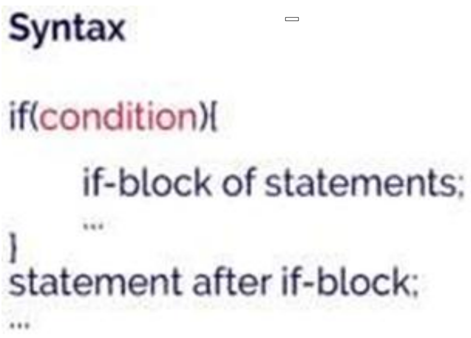


Control Statements

- Selection Statements
  - if Statement
    - Simple if
    - if-else
    - nested if
    - if-else-if
  - switch Statement
    - switch
- Iterative Statements
  - while
  - do-while
  - for
  - for-each
- Jump Statements
  - break
  - continue
  - return

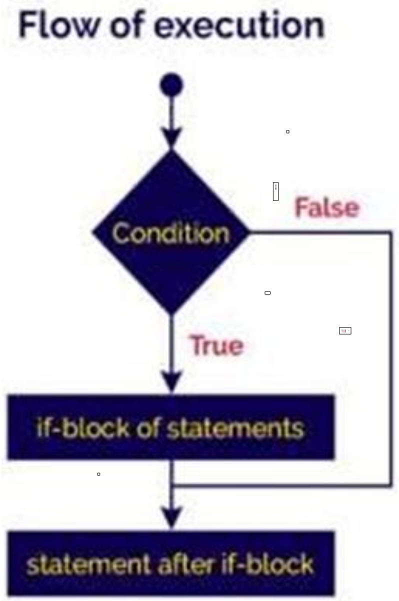
Decision-Making Statements

- if statement



Syntax

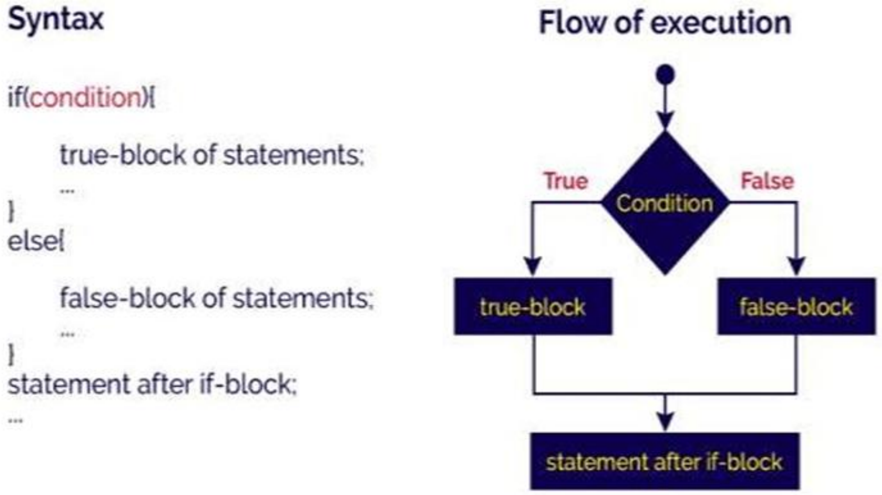
```
if(condition){
       if-block of statements;
       ...
}
statement after if-block;
...
```



Flow of execution

Decision-Making Statements

- if-else statement



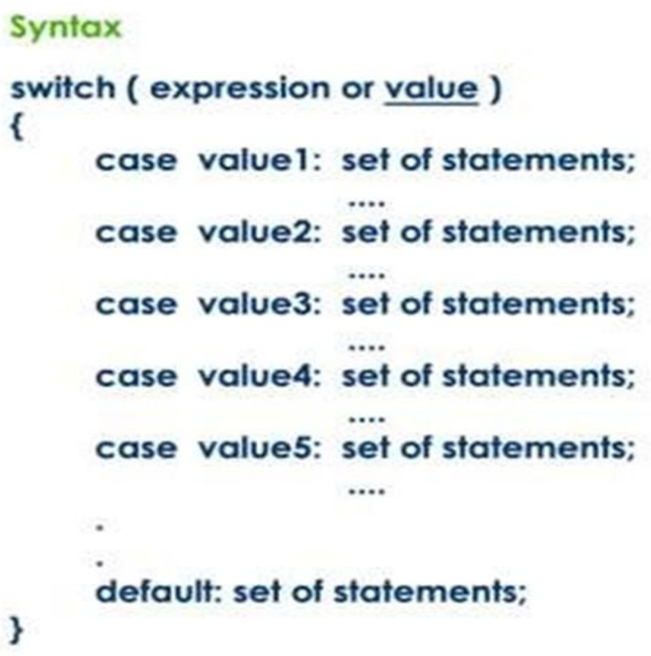
Syntax

```
if(condition){
        true-block of statements;
        ...
}
else{
        false-block of statements;
        ...
}
statement after if-block;
...
```

Flow of execution

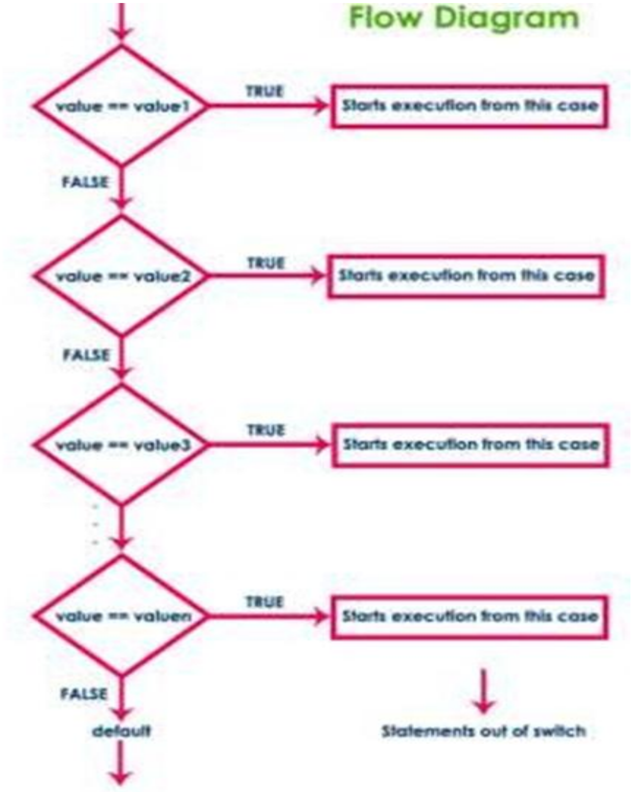
Decision-Making Statements

- Switch statement



Syntax

```
switch ( expression or value )
{
        case  value1:  set of statements;
                           ....
        case  value2:  set of statements;
                           ....
        case  value3:  set of statements;
                           ....
        case  value4:  set of statements;
                           ....
        case  value5:  set of statements;
                           ....
        .
        .
        default: set of statements;
}
```

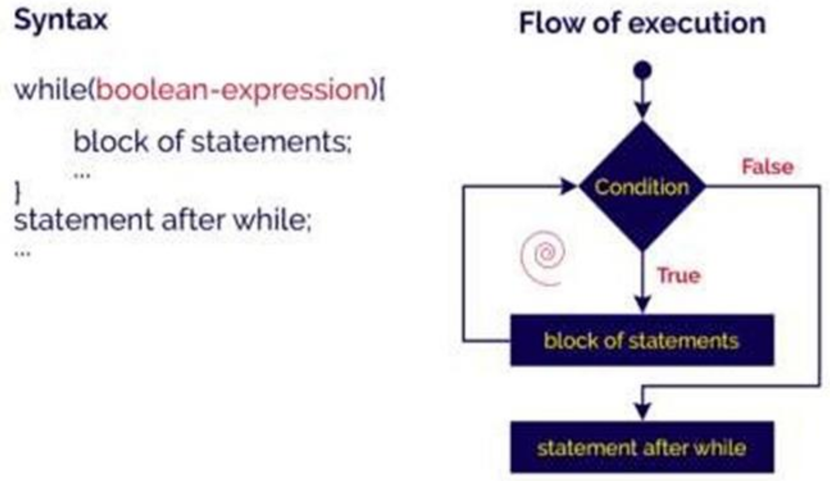


Flow Diagram

value == value1
TRUE
Starts execution from this case
FALSE
value == value2
TRUE
Starts execution from this case
FALSE
value == value3
TRUE
Starts execution from this case
value == valuen
TRUE
Starts execution from this case
FALSE
default
Statements out of switch

Looping Statements

- while statement



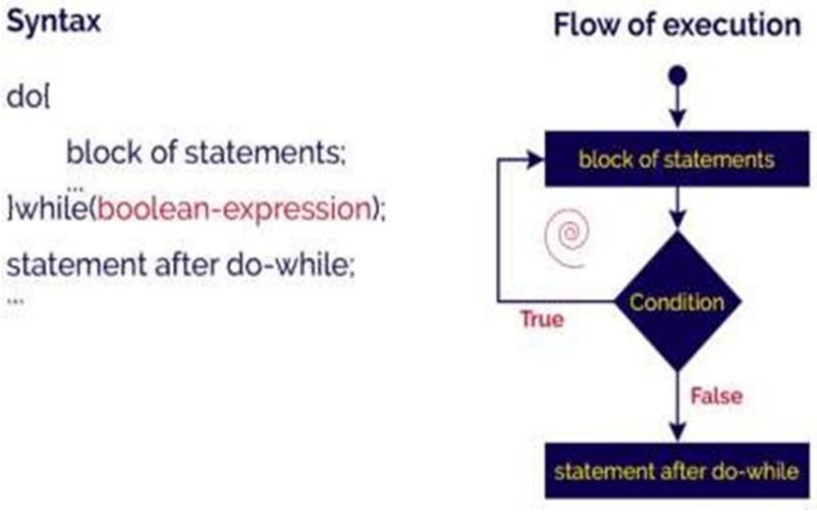
Syntax

```
while(boolean-expression){
        block of statements;
        ...
}
statement after while;
...
```

Flow of execution

Looping Statements

- do-while statement



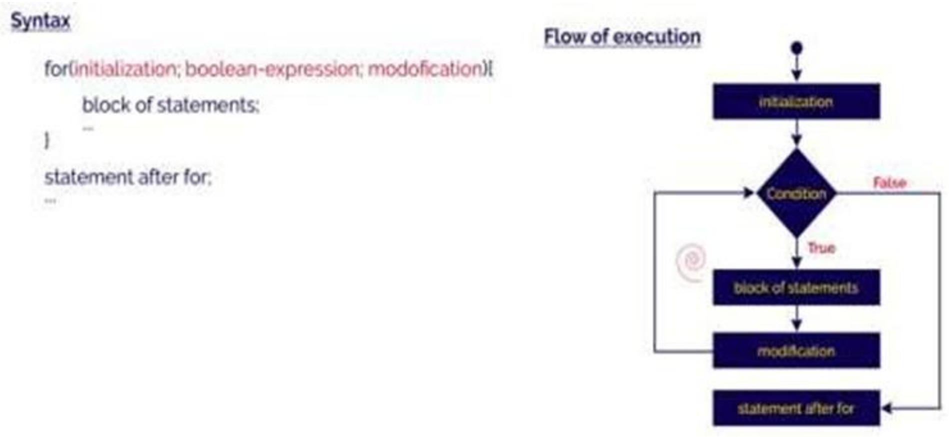
Syntax

```
do{
        block of statements;
        ...
}while(boolean-expression);
statement after do-while;
...
```

Flow of execution

Looping Statements

- for statement



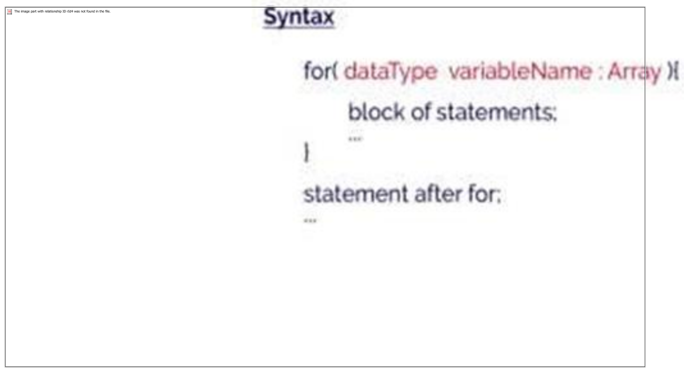
Syntax

```
for(initialization; boolean-expression; modofication){
      block of statements;
      ...
}
statement after for;
...
```

Flow of execution

Looping Statements

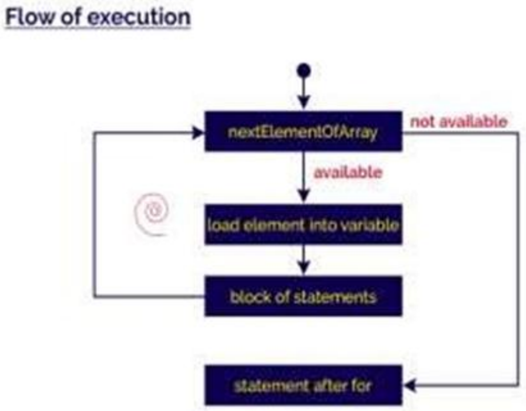
- for-each statement



The image part with relationship ID rId4 was not found in the file.

Syntax

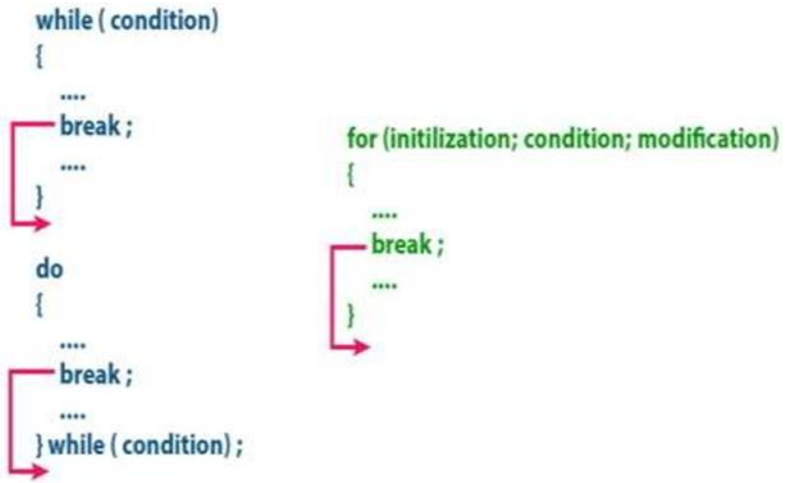
```
for( dataType  variableName : Array ){
        block of statements;
        ...
}
statement after for;
...
```



Flow of execution

Branching Statements

- break statement



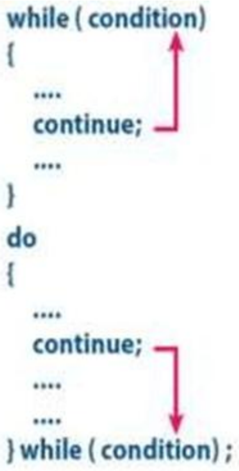
```
while ( condition)
{
  ....
  break ;
  ....
}

do
{
  ....
  break ;
  ....
} while ( condition) ;
```

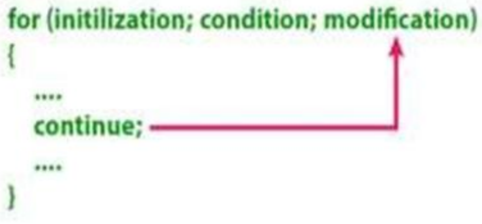
```
for (initilization; condition; modification)
{
  ....
  break ;
  ....
}
```

Branching Statements

- continue statement



```
while ( condition)
{
   ....
   continue;
   ....
}
do
{
   ....
   continue;
   ....
   ....
} while ( condition) ;
```



```
for (initilization; condition; modification)
{
   ....
   continue;
   ....
}
```

231T405: JAVA PROGRAMMING

Topic: Elements of Java: Class and Objects

Department of Information Technology

What is a Class?

- - A blueprint or template for creating objects
- - Defines properties (fields) and behaviors (methods)
- - Example:

```
class Car {
    String color;
    int speed;
    void accelerate() {
        System.out.println("Car is accelerating");
    }
}
```

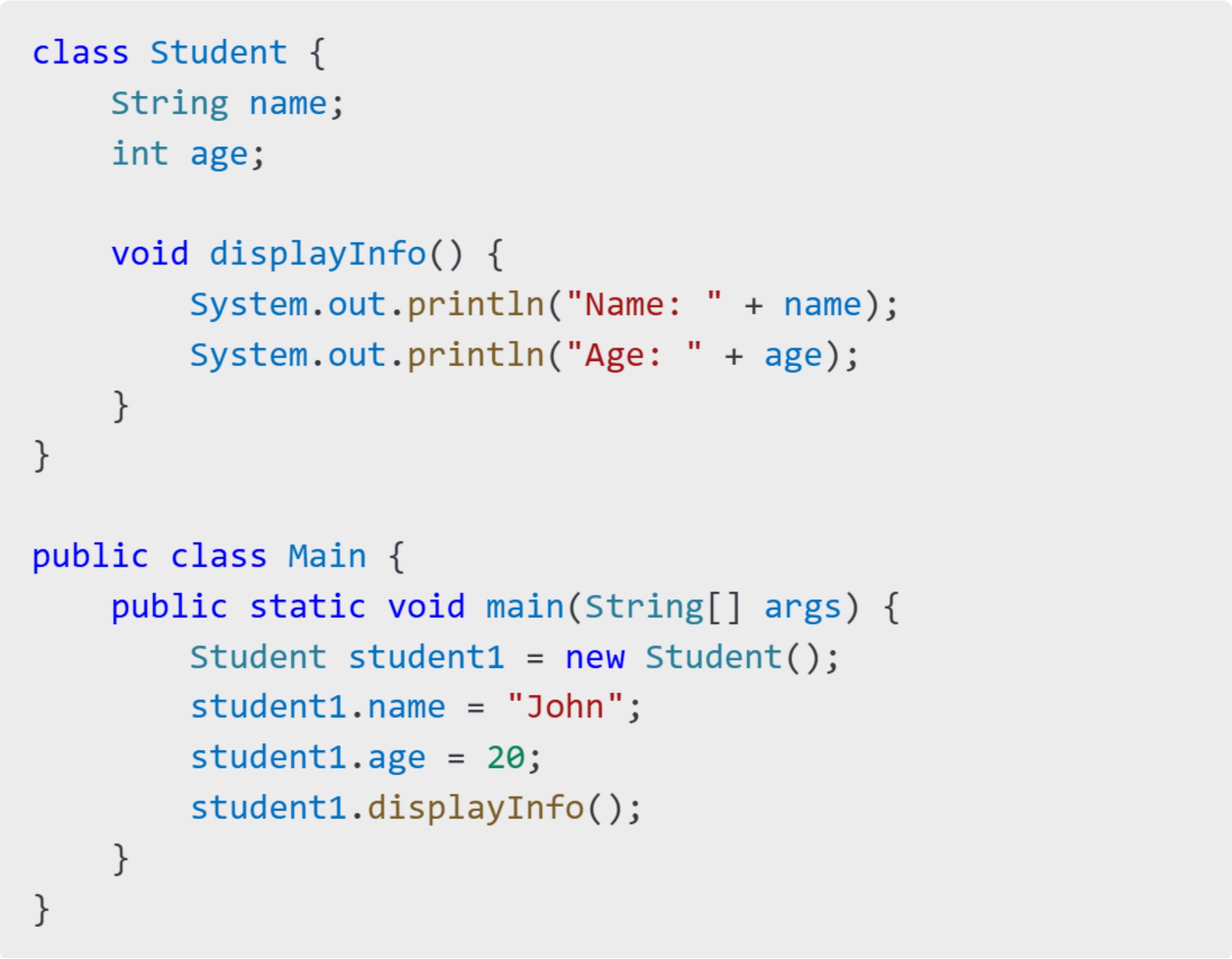
What is an Object?

- - An instance of a class
- - Represents real-world entities with states and behaviors
- - Example:
  - *Car myCar = new Car();*
  - *myCar.color = "Red";*
  - *myCar.speed = 120;*
  - *myCar.accelerate();*

Class vs Object

Aspect	Class	Object
Definition	Blueprint for creating objects	Instance of a class
Memory	No memory allocated	Memory allocated
Example	class Car	Car myCar = new Car();

Hands-on Example



```java
class Student {
    String name;
    int age;

    void displayInfo() {
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
    }
}

public class Main {
    public static void main(String[] args) {
        Student student1 = new Student();
        student1.name = "John";
        student1.age = 20;
        student1.displayInfo();
    }
}
```

Real-World Analogy

- Class: Blueprint of a house

- Object: Actual house built using the blueprint

- Example:

- Class: Car blueprint (design)

- Object: A specific car (red, 120 km/h speed)

Why Use Classes and Objects?

- Promotes reusability and modularity

- Encapsulation of data and behavior

- Simplifies maintenance and debugging

- Enables real-world modeling

231T405: JAVA PROGRAMMING

Topic: Elements of Java -Methods, Constructors

Department of Information Technology

Why Methods and Constructors?

- Methods enable reusable, modular code.

- Constructors initialize objects when they are created.

- Essential for building structured, maintainable programs.

What are Methods?

- - A block of code that performs a specific task.
- - Can accept input (parameters) and return output.
- - Syntax:

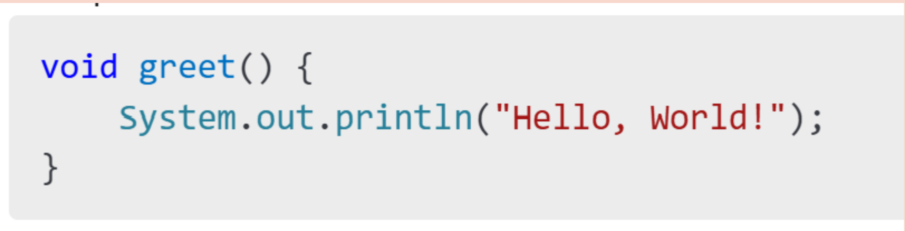
```
returnType methodName(parameters) {
   // code to be executed
}
```

Different Types of Methods

Built-in Methods: Predefined methods in Java (e.g., Math.max()).

User-defined Methods: Created by programmers for specific tasks.

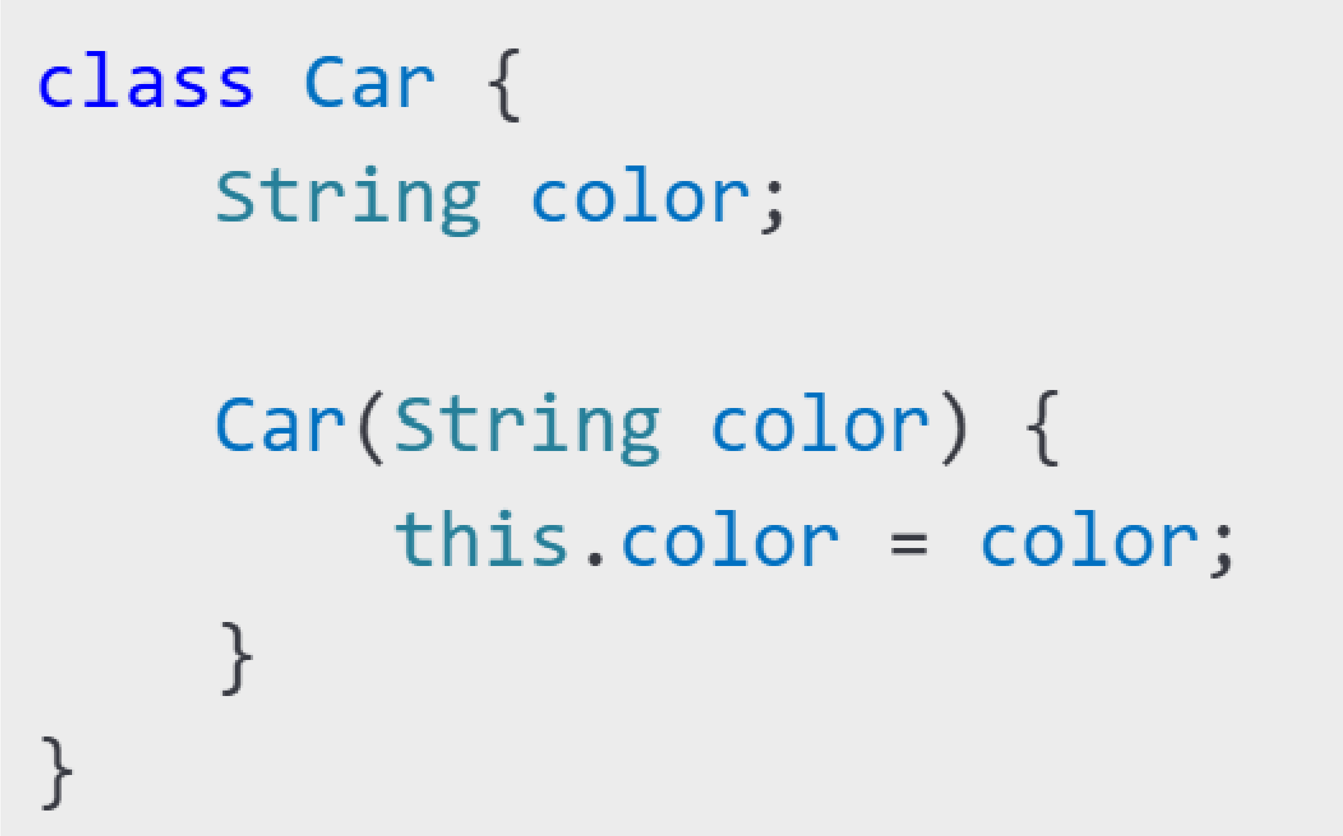
Example:



```
void greet() {
    System.out.println("Hello, World!");
}
```

What are Constructors?

- Special methods used to initialize objects.
- Same name as the class and no return type.
- Automatically called when an object is created.
- Example:

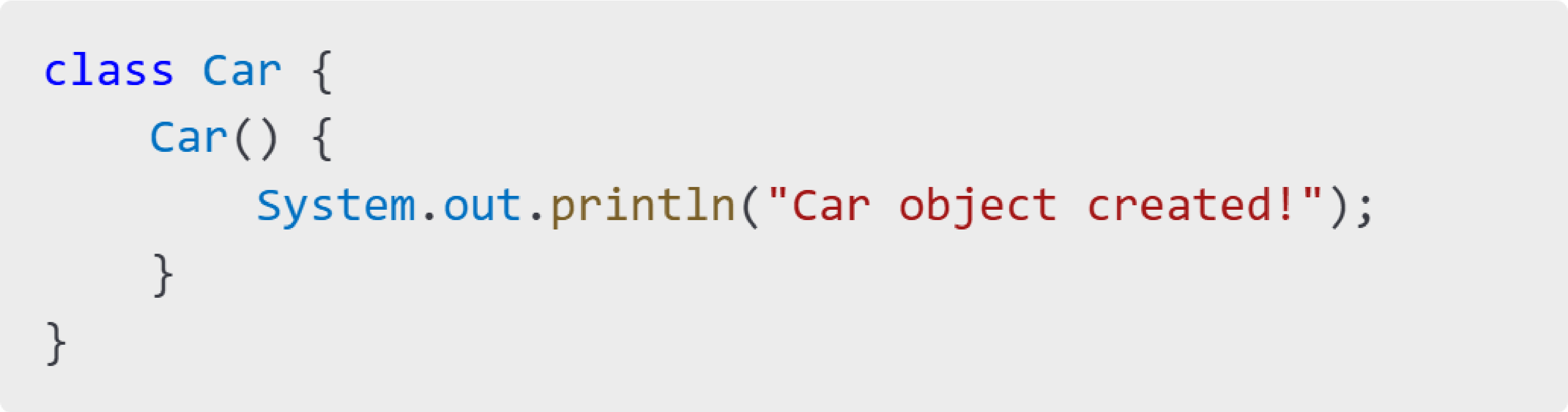


```
class Car {
    String color;

    Car(String color) {
        this.color = color;
    }
}
```

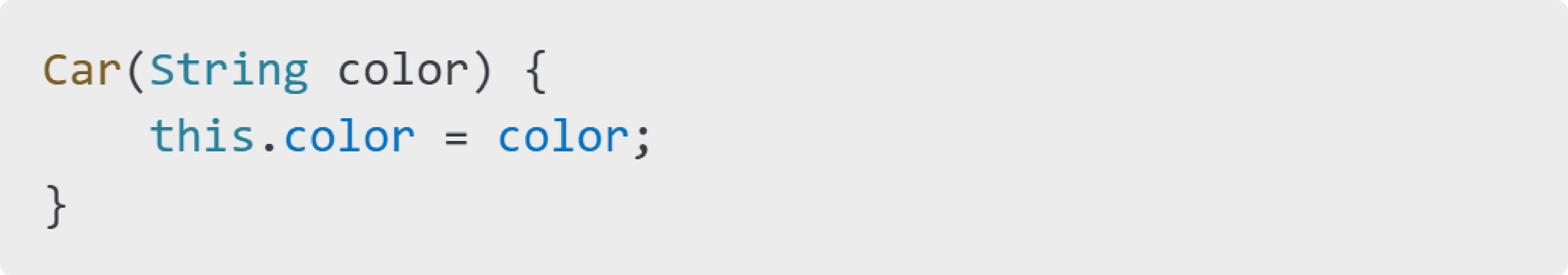
Different Types of Constructors

- **Default Constructor:** Provided by Java if no constructor is defined.



```java
class Car {
    Car() {
        System.out.println("Car object created!");
    }
}
```

- **Parameterized Constructor:** Accepts arguments to initialize fields.

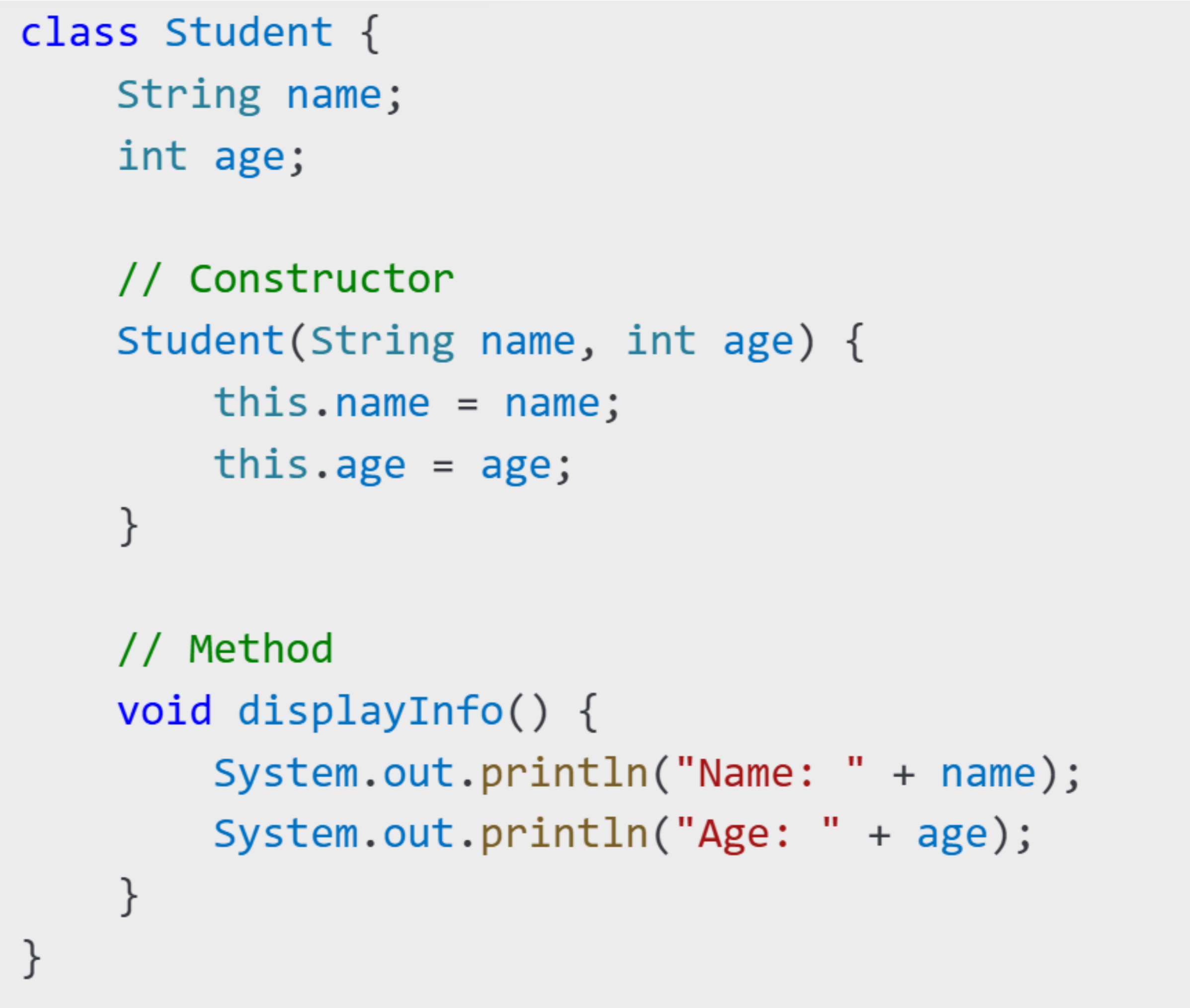


```java
Car(String color) {
    this.color = color;
}
```

Difference Between Methods and Constructors

Aspect	Methods	Constructors
Purpose	Performs a task	Initializes objects
Name	Any valid name	Same as class name
Return Type	Must have a return type	No return type
Explicit Call	Called explicitly	Called automatically

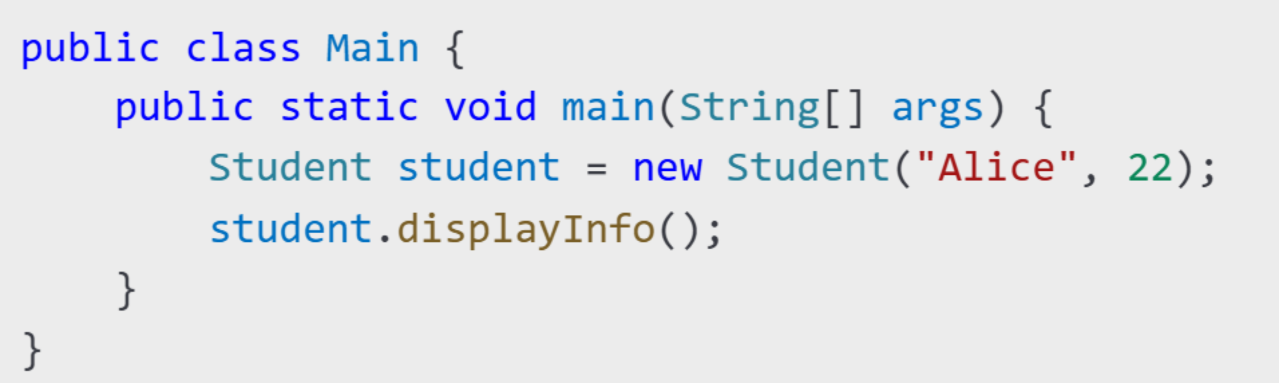
Methods and Constructors in Action



```java
class Student {
    String name;
    int age;

    // Constructor
    Student(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Method
    void displayInfo() {
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
    }
}
```



```java
public class Main {
    public static void main(String[] args) {
        Student student = new Student("Alice", 22);
        student.displayInfo();
    }
}
```

Writing Effective Methods and Constructors

- Keep methods small and focused on a single task.
- Use meaningful names for methods and parameters.
- Use constructors to enforce mandatory fields.
- Overload methods and constructors for flexibility.

231T405: JAVA PROGRAMMING

Topic: Elements of Java -Access Modifiers, Generics

Department of Information Technology

Why Learn Access Modifiers and Generics?

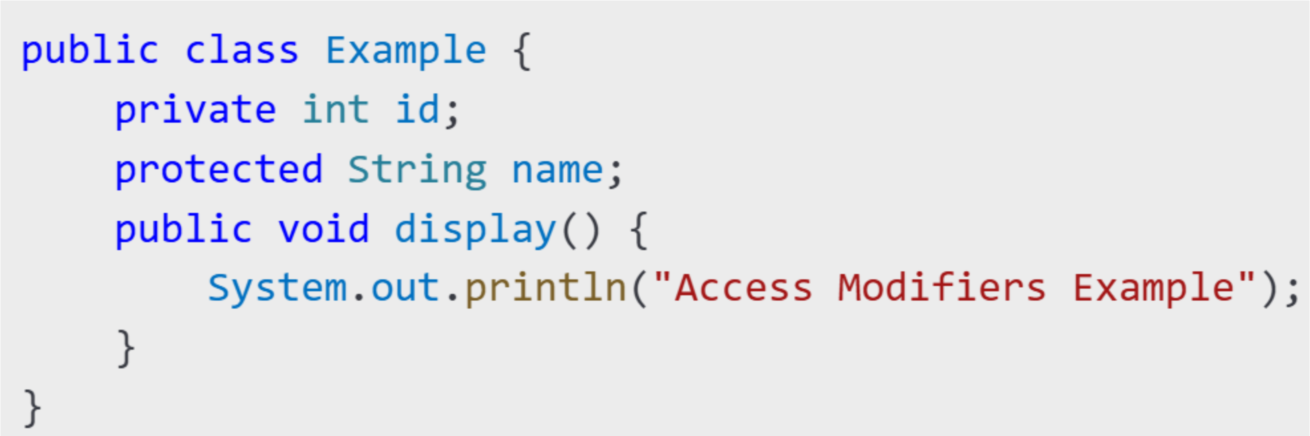
Access Modifiers control visibility and access to classes, methods, and fields.

Generics enable type-safe and reusable code.

Essential for robust and maintainable Java applications.

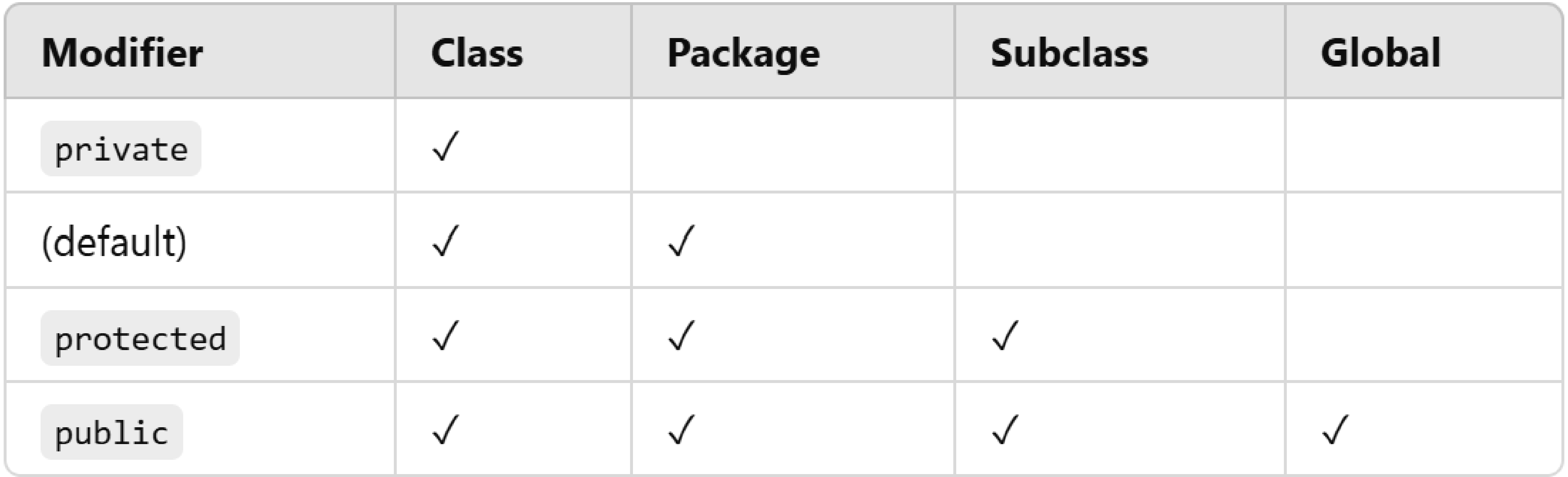
What Are Access Modifiers?

- Keywords used to define the scope of accessibility.
- Four levels of access:
  - **Private:** Accessible within the same class only.
  - **Default (Package-private):** Accessible within the same package.
  - **Protected:** Accessible within the same package and by subclasses.
  - **Public:** Accessible from everywhere.



```java
public class Example {
    private int id;
    protected String name;
    public void display() {
        System.out.println("Access Modifiers Example");
    }
}
```

Access Levels at a Glance



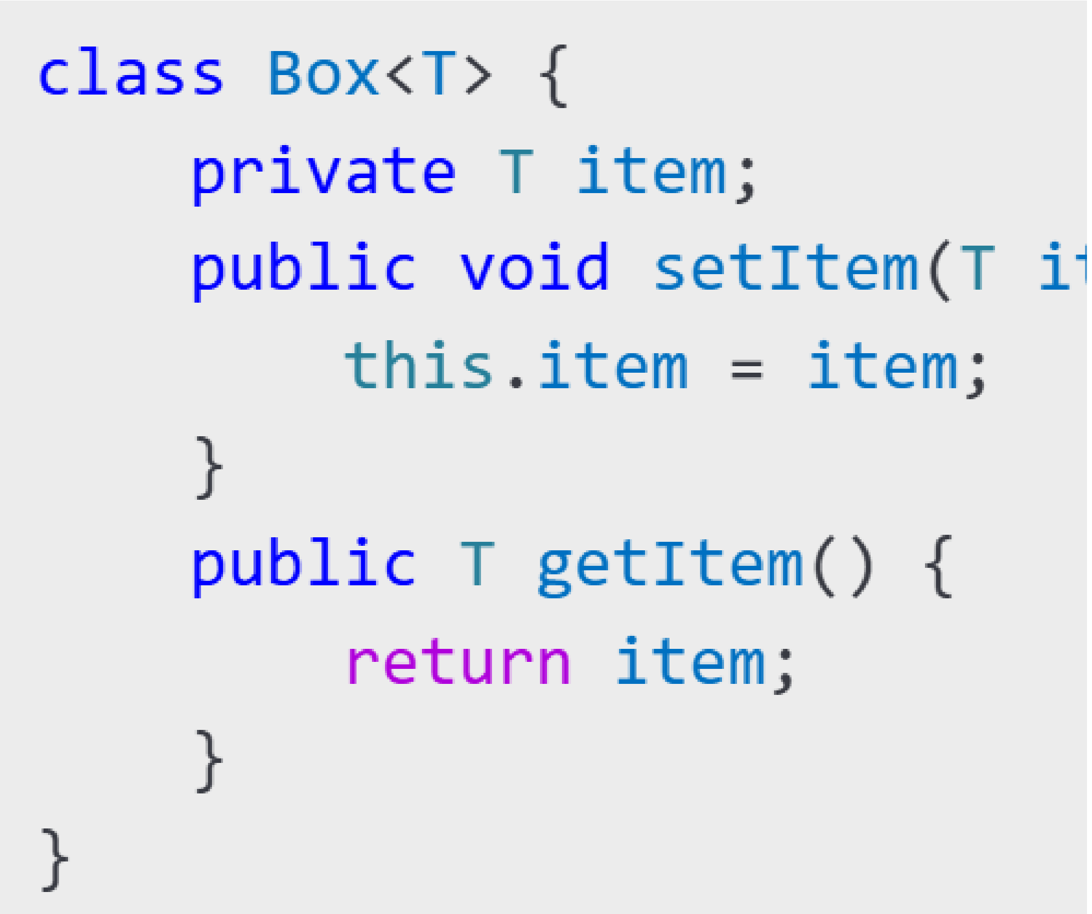
Modifier	Class	Package	Subclass	Global
`private`	✓			
(default)	✓	✓		
`protected`	✓	✓	✓	
`public`	✓	✓	✓	✓

Generics

Generics means **parameterized types**. The idea is to allow a type (like Integer, String, etc., or user-defined types) to be a parameter to methods,
classes, and interfaces. Using Generics, it is possible to create classes that work with different data types. An entity such as a class,
interface, or method that operates on a parameterized type is a **generic entity**.

Understanding Generics in Java

- Introduced in Java 5 for type safety.
- Enables the definition of classes, methods, and interfaces with type parameters.
- Syntax example:

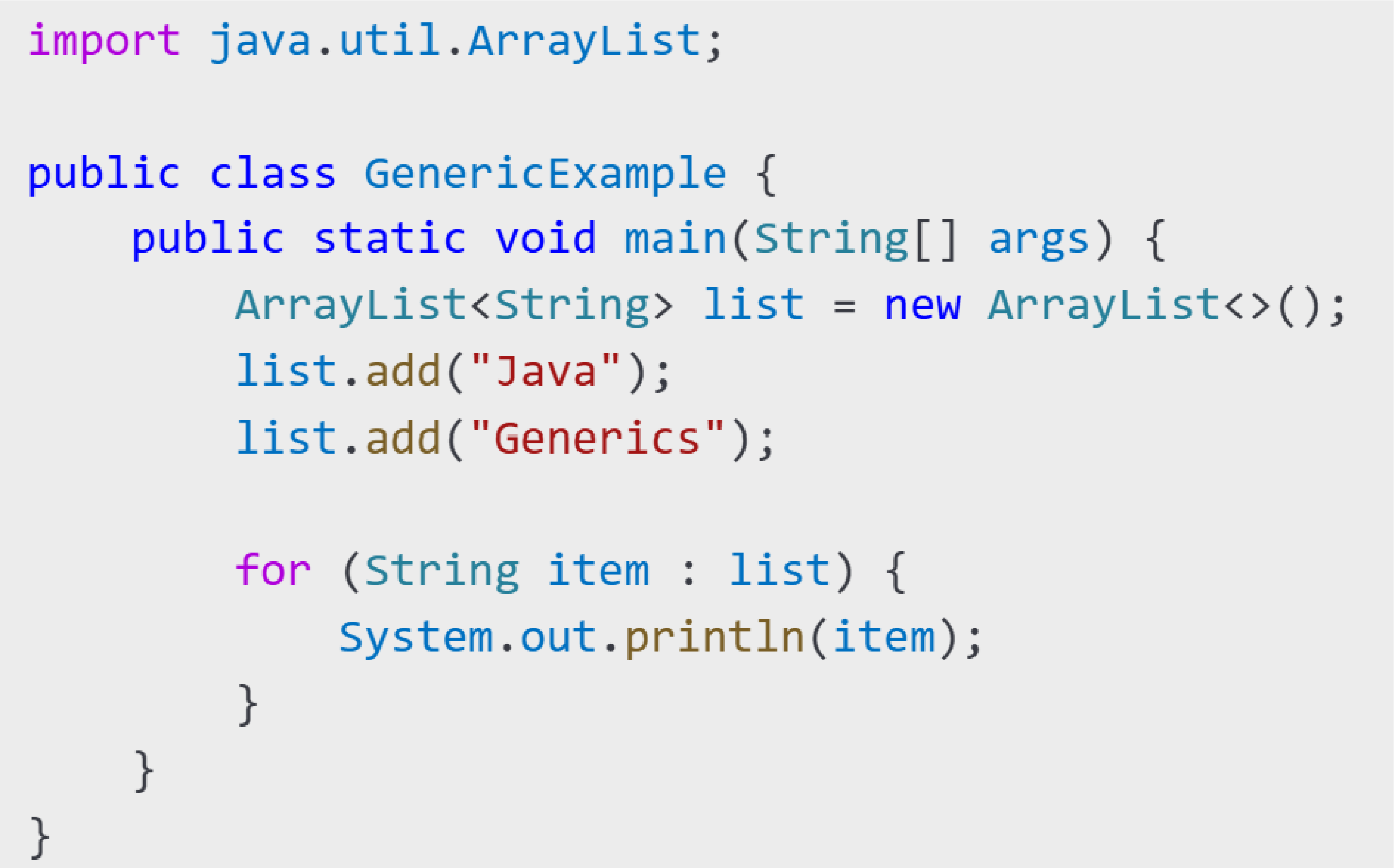


```
class Box<T> {
    private T item;
    public void setItem(T i
        this.item = item;
    }
    public T getItem() {
        return item;
    }
}
```

Why Use Generics?

- **Type Safety:** Prevents runtime errors.
- **Code Reusability:** Single definition for multiple data types.
- **Readability:** Explicit types make code easier to understand.
- **Performance:** Reduces the need for type casting.

Example of Generics

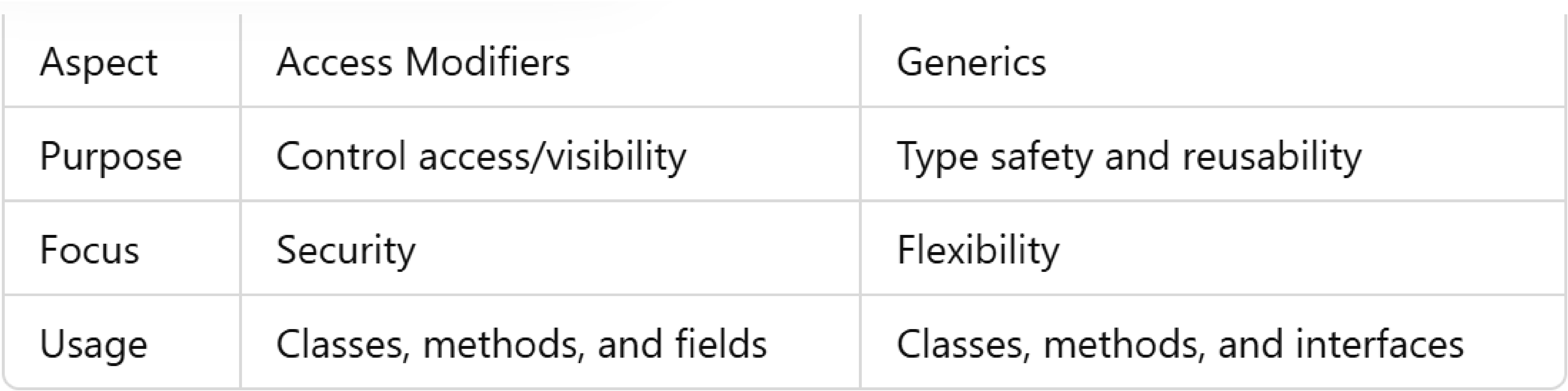


```java
import java.util.ArrayList;

public class GenericExample {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>();
        list.add("Java");
        list.add("Generics");

        for (String item : list) {
            System.out.println(item);
        }
    }
}
```

Access Modifiers vs Generics



Aspect	Access Modifiers	Generics
Purpose	Control access/visibility	Type safety and reusability
Focus	Security	Flexibility
Usage	Classes, methods, and fields	Classes, methods, and interfaces

Using Access Modifiers and Generics Effectively

Use the least permissive access modifier possible.

Prefer protected over public for inheritance.

Use Generics for type-safe collections.

Avoid raw types when using Generics.

231T405: JAVA PROGRAMMING

Topic: Elements of Java -Inner classes, String class, Annotations

Department of Information Technology

Why Learn These Concepts?

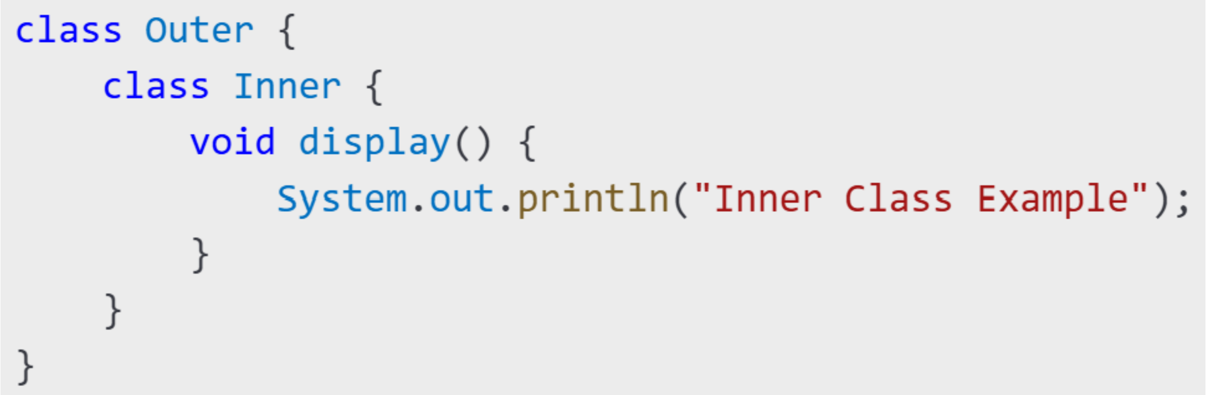
Inner classes encapsulate logic and simplify structure.

The String class is fundamental for text manipulation in Java.

Annotations provide metadata to enhance functionality.

What Are Inner Classes?

- A class defined within another class.
- Types of inner classes:
  - **Non-static (Member) Inner Class:** Associated with an instance of the outer class.
  - **Static Nested Class:** Acts like a static member of the outer class.
  - **Local Inner Class:** Defined within a method or block.
  - **Anonymous Inner Class:** Used for implementing interfaces or abstract classes inline.



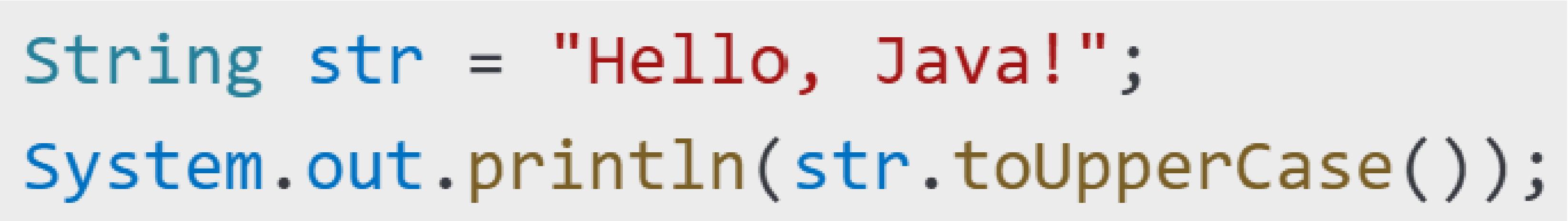
```java
class Outer {
    class Inner {
        void display() {
            System.out.println("Inner Class Example");
        }
    }
}
```

Types of Inner Classes

Type	Characteristics	Example Use Case
Member Inner Class	Non-static, tied to an instance	Accessing instance members
Static Nested Class	Static, independent of outer instance	Utility or helper functions
Local Inner Class	Defined within a method	Method-specific logic
Anonymous Inner Class	No name, one-time use	Inline implementation

What Is the String Class?

- A sequence of characters, immutable by design.
- Part of the java.lang package.
- Commonly used methods:
  - length(), charAt(int index), substring(int start, int end).
  - toLowerCase(), toUpperCase(), replace(), equals(), equalsIgnoreCase()



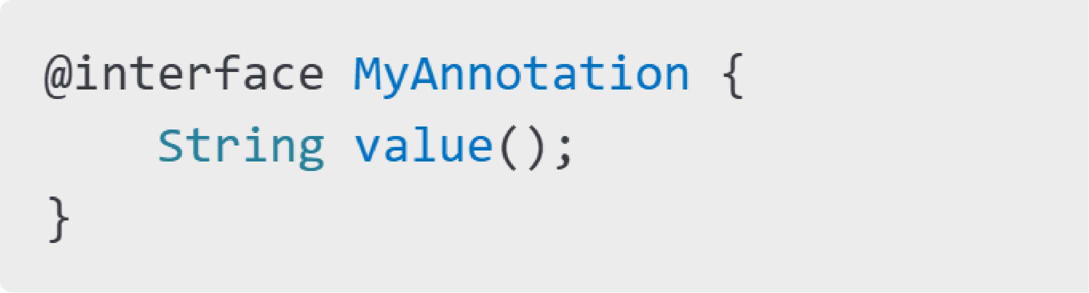
```java
String str = "Hello, Java!";
System.out.println(str.toUpperCase());
```

Common String Methods

Method	Description	Example
length()	Returns the length of the string	str.length()
substring()	Extracts a portion of the string	str.substring(0, 5)
equals()	Compares two strings for equality	str1.equals(str2)
toUpperCase()	Converts the string to uppercase	str.toUpperCase()

What Are Annotations?

- **Content:**
  - Provide metadata for Java code.
  - Built-in annotations:
    - `@Override`
    - `@Deprecated`
    - `@SuppressWarnings`
  - Custom annotations:



```
@interface MyAnnotation {
    String value();
}
```

Built-in Annotations in Java

Annotation	Purpose	Example
@Override	Indicates method overriding	@Override void display()
@Deprecated	Marks a method as deprecated	@Deprecated void oldMethod()
@SuppressWarnings	Suppresses specific warnings	@SuppressWarnings("unchecked")

Effective Use of Inner Classes, Strings, and Annotations

Use inner classes to logically group classes.

Prefer **StringBuilder** or **StringBuffer** for mutable strings.

Use annotations for clarity and to reduce boilerplate code.

Understanding Packages in Java

What is a Package in Java?

- - A package is a container for classes, interfaces, and sub-packages.
- - It helps in organizing code logically.
- - Prevents class name conflicts and provides access protection.

- Example:

```
• package mypackage;
• public class MyClass {
•     public void show() {
•         System.out.println("Hello from MyClass");
•     }
• }
```

Types of Packages in Java

- 1. Built-in Packages – Provided by Java (e.g., java.util, java.io).
- 2. User-defined Packages – Created by developers.

Advantages of Using Packages

- ⍰ Code Organization – Groups related classes together.
- ⍰ Encapsulation – Controls access using access modifiers.
- ⍰ Avoids Name Conflicts – Different packages can have classes with the same name.
- ⍰ Reusability – Classes in a package can be reused across projects.

Built-in Packages in Java

- Common built-in packages:

- - java.lang: Basic classes (String, Math, Object, etc.)
- - java.util: Utility classes (ArrayList, HashMap, Scanner, etc.)
- - java.io: Input-output operations (File handling)
- - java.sql: Database connectivity (JDBC)
- - javax.swing: GUI components (JFrame, JButton, etc.)

- Example: import java.util.Scanner;

Creating a User-Defined Package

- 1. Define a package:
- package mypackage;
- public class MyClass {
-   public void show() {
-     System.out.println("Hello from MyClass");
-   }
- }

- 2. Compile using:
- javac -d . MyClass.java

- 3. Use in another class:
- import mypackage.MyClass;
- public class Test {
-   public static void main(String[] args) {
-     MyClass obj = new MyClass();
-     obj.show();
-   }
- }

Importing Packages in Java

- Ways to import packages:

- - Importing a specific class: import java.util.Scanner;
- - Importing the entire package: import java.util.*;
- - Using fully qualified class name: java.util.Scanner sc = new java.util.Scanner(System.in);

Package Hierarchy and Sub-Packages

- A package can contain sub-packages:

- Example structure:
- com.company
-   ├── finance
-   │   ├── Invoice.java
-   │   └── Tax.java
-   ├── hr
-   │   ├── Employee.java
-   │   ├── Payroll.java

- Accessing a sub-package class: import com.company.finance.Invoice;

Access Control in Packages

- | Access Modifier | Same Class | Same Package | Subclass (Different Package) | Other Packages |
- |----------------|-----------|-------------|-----------------------------|---------------|
- | public | □ Yes | □ Yes | □ Yes | □ Yes |
- | protected | □ Yes | □ Yes | □ Yes | □ No |
- | default | □ Yes | □ Yes | □ No | □ No |
- | private | □ Yes | □ No | □ No | □ No |

Best Practices for Using Packages

- ⍰ Use meaningful names (e.g., com.company.module).
- ⍰ Follow Java naming conventions (all lowercase).
- ⍰ Group related classes logically.
- ⍰ Use access modifiers to control visibility.
- ⍰ Avoid too many nested packages.

Conclusion

- - Packages help in organizing Java programs efficiently.
- - Java provides many built-in packages for common tasks.
- - User-defined packages improve reusability and modularity.
- - Proper use of access modifiers ensures security and encapsulation.

String Tokenizer in Java

What is StringTokenizer?

- - `StringTokenizer` is a class in `java.util` package.
- - Used to split (tokenize) a string into smaller parts called tokens.
- - It is an alternative to `split()` method.

- Example:

~~~
•   ```java
•   import java.util.StringTokenizer;

•   public class Test {
•       public static void main(String[] args) {
•           StringTokenizer st = new StringTokenizer("Java is fun");
•           while (st.hasMoreTokens()) {
•               System.out.println(st.nextToken());
•           }
•       }
•   }
•   ```
•   Java
•   is
•   fun
~~~

Why Use StringTokenizer?

- ⍰ Efficient for simple string splitting.
- ⍰ Does not create extra arrays (unlike `split()`).
- ⍰ Useful for parsing structured data (e.g., CSV, logs).

Constructors of StringTokenizer

- | Constructor | Description |
- |------------|-------------|
- | `StringTokenizer(String str)` | Splits `str` using default delimiter (whitespace). |
- | `StringTokenizer(String str, String delim)` | Splits `str` using a custom delimiter `delim`. |
- | `StringTokenizer(String str, String delim, boolean returnDelims)` | If `returnDelims` is `true`, delimiters are also returned as tokens. |

Tokenizing with Custom Delimiters

- Example (Using `,` as a delimiter):

```java
import java.util.StringTokenizer;

public class Test {
    public static void main(String[] args) {
        StringTokenizer st = new StringTokenizer("Apple,Banana,Cherry", ",");
        while (st.hasMoreTokens()) {
            System.out.println(st.nextToken());
        }
    }
}
```

- Apple
- Banana
- Cherry

Returning Delimiters as Tokens

- Use `true` as the third argument to include delimiters as tokens.

- Example:

-
  ```java
- StringTokenizer st = new StringTokenizer("A,B;C", ",;", true);
- while (st.hasMoreTokens()) {
-     System.out.println(st.nextToken());
- }
- ```
- A
- ,
- B
- ;
- C
  ```

Important Methods in StringTokenizer

- | Method | Description |
- |--------|-------------|
- | `hasMoreTokens()` | Returns `true` if more tokens are available. |
- | `nextToken()` | Returns the next token. |
- | `nextToken(String delim)` | Returns the next token, using a new delimiter. |
- | `countTokens()` | Returns the number of tokens left. |

Counting Tokens

- Example:

-
  ```java
- StringTokenizer st = new StringTokenizer("Hello World Java");
- System.out.println("Total tokens: " + st.countTokens());
- ```

- **Output:**
- ```
- Total tokens: 3
- ```
  ```

StringTokenizer vs split()

- | Feature | StringTokenizer | `split()` |
- |---------|---------------|----------|
- | Performance | Faster for simple tokenization | Slower due to regex processing |
- | Returns | Individual tokens one by one | Array of tokens |
- | Delimiter Handling | Can choose to return delimiters | Always removes delimiters |
- | Flexibility | Less flexible | More flexible with regex |

Best Practices

- ⍰ Use `StringTokenizer` for simple splitting tasks.
- ⍰ Prefer `split()` when using regex-based splitting.
- ⍰ Use `returnDelims = true` if delimiters should be preserved.
- ⍰ Use `countTokens()` to pre-check token count before iteration.

Conclusion

- - `StringTokenizer` is useful for simple and efficient string splitting.
- - Supports custom delimiters and delimiter return mode.
- - Alternative to `split()`, but lacks regex support.