

CS6308 Java Programming

V P Jayachitra

Assistant Professor
Department of Computer
Technology
MIT Campus
Anna University

Day 1

V P Jayachitra

Syllabus

MODULE I	FUNDAMENTALS OF JAVA LANGUAGE	L	T	P	EL
		3	0	4	3
Introduction to Java, Java basics – Variables, Operators, Expressions, Control flow Statements, Methods, Arrays					
SUGGESTED ACTIVITIES : <ul style="list-style-type: none"> • Practical-Implementation of simple Java programs Using Java Basic Constructs and Arrays using any standard IDE like NETBEANS / ECLIPSE • EL – Understanding JVM 					
SUGGESTED EVALUATION METHODS: <ul style="list-style-type: none"> • Assignment problems • Quizzes 					
TEXT BOOKS: <ol style="list-style-type: none"> 1. Y. Daniel Liang, "Introduction to Java Programming and Data Structures, Comprehensive Version", 11th Edition, Pearson Education, 2018. 2. Herbert Schildt, "Java: The Complete Reference", 11th Edition, McGraw-Hill Education, 2018. 					
REFERENCES: <ol style="list-style-type: none"> 1. Paul Dietel and Harvey Deitel, "Java - How to Program Early Objects", 11th Edition, Pearson Education, 2017. 2. Sachin Malhotra, Sourabh Choudhary, "Programming in Java", Revised 2nd Edition, Oxford University Press, 2018. 3. Cay S. Horstmann, "Core Java - Vol. 1, Fundamentals", 11th Edition, Pearson Education, 2018. 					
Web references: <ol style="list-style-type: none"> 1. NPTEL 2. MIT OCW 					
EVALUATION PATTERN:					
Category of Course	Continuous Assessment	Mid –Semester Assessment	End Semester		
Theory Integrated with Practical	15(T) + 25 (P)	20	40		

The Origins of Java

- Java is conceived by
 - James Gosling
 - Patrick Naughton
 - Chris Warth
 - Ed Frank, and
 - Mike Sheridanat Sun Microsystems in 1991.
- Known as Oak initially and renamed as Java in 1995.

Motivation

- First, to create a platform-independent language
 - To create software for various consumer electronic devices, such as toasters, microwave ovens, and remote controls.
 - Why?
 - computer languages were designed to be compiled into machine code that was targeted for a specific type of CPU.
 - compilers are expensive and time consuming to create for all kind of CPU.
 - What?
 - portable, cross-platform language that could produce code that would run on a variety of CPUs under differing environments.

Motivation contd...

- Second focus is the World Wide Web
 - Why?
 - Web, demanded portable programs
 - Java was an obscure language for consumer electronics.
 - What?
 - Java was propelled to the forefront of computer language design due to WWW.
 - architecture-neutral programming language
 - focus switch from consumer electronics to Internet programming.

Motivation contd...

- Third, Security
 - Java's support for networking.
- Why?
 - library of ready-to-use functionality enabled programmers to easily write programs that accessed or made use of the Internet.
 - Downloading and executing program at multiplatformed environment may harm the system (virus, Trojan etc.,)
- What?
 - Confine an application to the Java execution environment and prevent it from accessing other parts of the computer.

Java's Lineage: C and C++

- From C, Java inherits its syntax.
- Java's object model is adapted from C++.
 - Java is not an enhanced version of C++
 - Neither upwardly nor downwardly compatible with C++
 - C++ was designed to solve a different set of problems.
 - Java was designed to solve a certain set of problems.

Key features of Java

- Simple
- Secure
- Portable
- Object-oriented
- Robust
- Multithreaded
- Architecture-neutral
- Interpreted
- High performance
- Distributed
- Dynamic

Key features of Java contd...

- **Simple**

- Easy to learn and use effectively.
 - no pointers/stack concerns
- Because Java inherits the C/C++ syntax and many of the object-oriented features of C++

- **Object-Oriented**

- “everything is an object” paradigm
- Only primitive types, such as integers, are kept as high-performance nonobjects.

Key features of Java contd...

- **Robust**

- Java is a strongly typed language
 - checks the code at compile time, also checks the code at run time
- Garbage collector
 - Automatic memory management-
- Exception handling

- **Architecture-Neutral**

- JVM
 - code longevity and portability.
 - “write once; run anywhere, any time, forever.”

Key features of Java contd...

- **Multithreaded**

- write programs that do many things simultaneously.
- multiprocess synchronization -enables to construct smoothly running interactive systems

- **Interpreted and High Performance**

- Java bytecode -carefully designed to translate directly into native machine code
- For very high performance -just-in-time compiler

Key features of Java contd...

- **Distributed**

- Java is designed for the distributed environment of the Internet as it handles TCP/IP protocols.
- Remote Method Invocation (RMI). This feature enables a program to invoke methods across a network.

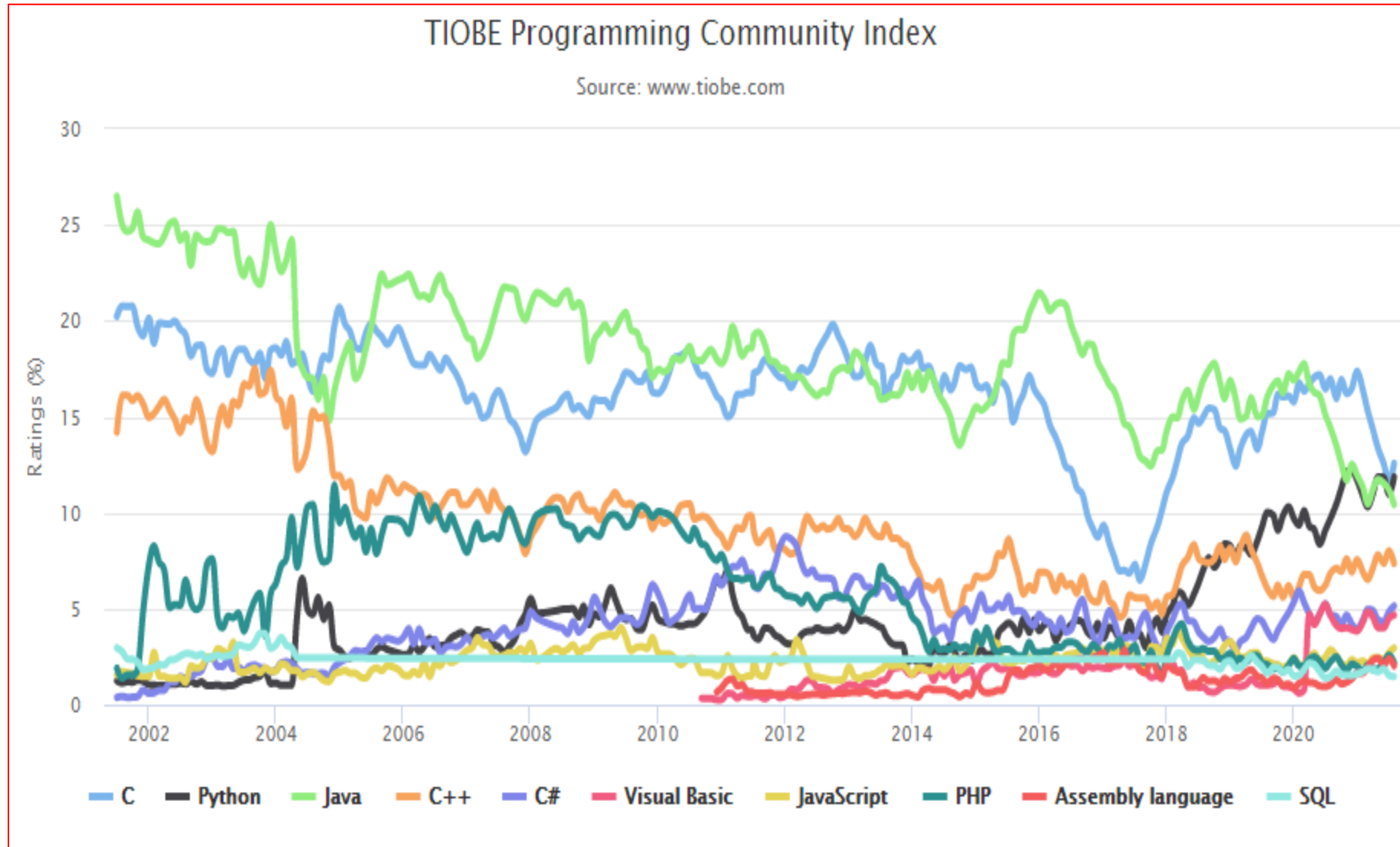
- **Interpreted and High Performance**

- Java bytecode -carefully designed to translate directly into native machine code
- For very high performance -just-in-time compiler

Key features of Java contd...

- Dynamic
 - run-time type information -used to verify and resolve accesses to objects at run time.

Java – 2020 popular language



Java Editions

- Java 2 Platform, Standard Edition (J2SE)
 - Desktop based application and networking applications
- Java 2 Platform, Enterprise Edition (J2EE)
 - Large-scale, distributed networking applications and Web-based applications
- Java 2 Platform, Micro Edition (J2ME)
 - Small memory-constrained devices, such as cell phones, pagers and PDAs

Java development environment

- Text editor
- IDE (Integrated development environment)

Java IDE

- Integrated Development Environment(IDE)
 - a GUI based application that facilitates application development such as coding, compilation or interpretation and debug programs more easily.
 - An IDE contains
 - Code editor-syntax highlights
 - Build tools -Compiler /Debugger/Interpreter
 - Auto documentation-comments
 - Libraries
 - Top 3 lightweight IDE -2021
 - IntelliJ IDEA
 - Eclipse
 - NetBeans

Overview of Java

- Two Paradigms
 - Procedural oriented paradigm
 - process-oriented model
 - code acting on data
 - problems with this approach appear as programs grow larger and more complex.
 - object-oriented programming paradigm
 - data controlling access to code.
 - organizes a program around its data (that is, objects) and a set of well-defined interfaces to that data.
 - problems with this approach appear as programs grow larger and more complex.

Overview of Java

- Abstraction
 - A powerful way to manage abstraction is through the use of hierarchical classifications.
 - To manage complexity

Overview of Java

- The Three OOP Principle
 - Encapsulation
 - Inheritance, and
 - Polymorphism

Overview of Java

- Encapsulation:
 - mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse.
 - protective wrapper that prevents the code and data from being arbitrarily accessed by other code defined outside the wrapper.
 - knows how to access it and thus can use it regardless of the implementation details—and without fear of unexpected side effects.
- Class
 - To encapsulate complexity
 - A class defines the structure and behavior (data and code) that will be shared by a set of objects.
 - Members-code and data
 - Data-member variables or instance variables.
 - Code-member method or method
 - behavior and interface of a class are defined by the methods that operate on its instance data.
- objects are referred to as instances of a class.
- Thus, a class is a logical construct; an object has physical reality.
- Each method or variable in a class may be marked private or public.
- the *public* interface of a class represents everything that external users of the class need to know, or may know.
- The *private* methods and data can only be accessed by code that is a member of the class

Overview of Java

- **Inheritance**

- *Inheritance* is the process by which one object acquires the properties of another object.
- Without the use of hierarchies, each object would need to define all of its characteristics explicitly.
- an object need only define those qualities that make it unique within its class. It can inherit its general attributes from its paren

Overview of Java

- **Polymorphism**

- is a feature that allows one interface to be used for a general class of actions.
- The specific action is determined by the exact nature of the situation

A First Simple Program

/* First Java program

call this file Example.java

```
*/  
class Example{  
    public static void main(String args[]){  
        System.out.println("This is java template");  
    }  
}
```

The diagram highlights four syntax elements in the code with yellow boxes and blue arrows:

- Keyword**: Points to the opening comment marker `/*`.
- Identifier**: Points to the class name `Example`.
- Access modifier**: Points to the keyword `public` inside the `main` method.
- Keyword**: Points to the keyword `static` inside the `main` method.

A First Simple Program

- The keyword **class** to declare that a new class is being defined.
- **Example** is an *identifier* that is the name of the class.
- The entire class definition, including all of its members, will be between the opening curly brace ({) and the closing curly brace (}).
- The public keyword is an access modifier, which allows the programmer to control the visibility of class members.
- The keyword static allows main() to be called without having to instantiate a particular instance of the class.
- This is necessary since main() is called by the Java Virtual Machine before any objects are made.
- The keyword void simply tells the compiler that main() does not return a value.
- Java compiler will compile classes that do not contain a main() method. But java has no way to run these classes
- String args[] declares a parameter named args, which is an array of instances of the class String.
- **args** receives any command-line arguments present when the program is executed.
- A complex program will have dozens of classes, only one of which will need to have a **main()** method to get things started.
- **System** is a predefined class that provides access to the system, and **out** is the output stream that is connected to the console.
- println() is the method

Java program

- In Java, a source file is officially called a compilation unit.
- The Java compiler requires that a source file use the .java filename extension.
- In Java, all code must reside inside a class.
- Java is case-sensitive
- Filename should match the class name

Compiling the Program

- Execute the compiler, **javac**, specifying the name of the source file on the command line
- C:\>javac Example.java
- The **javac** compiler creates a file called **Example.class** that contains the bytecode version of the program.
- Java bytecode is the intermediate representation of your program that contains instructions the Java Virtual Machine will execute.
- The output of **javac** is not code that can be directly executed.

Run the program

- Use Java application launcher called **java**.
- Pass the class name **Example** as a command-line argument
- C:\>java Example
- Output:
 - This is java template

Comment

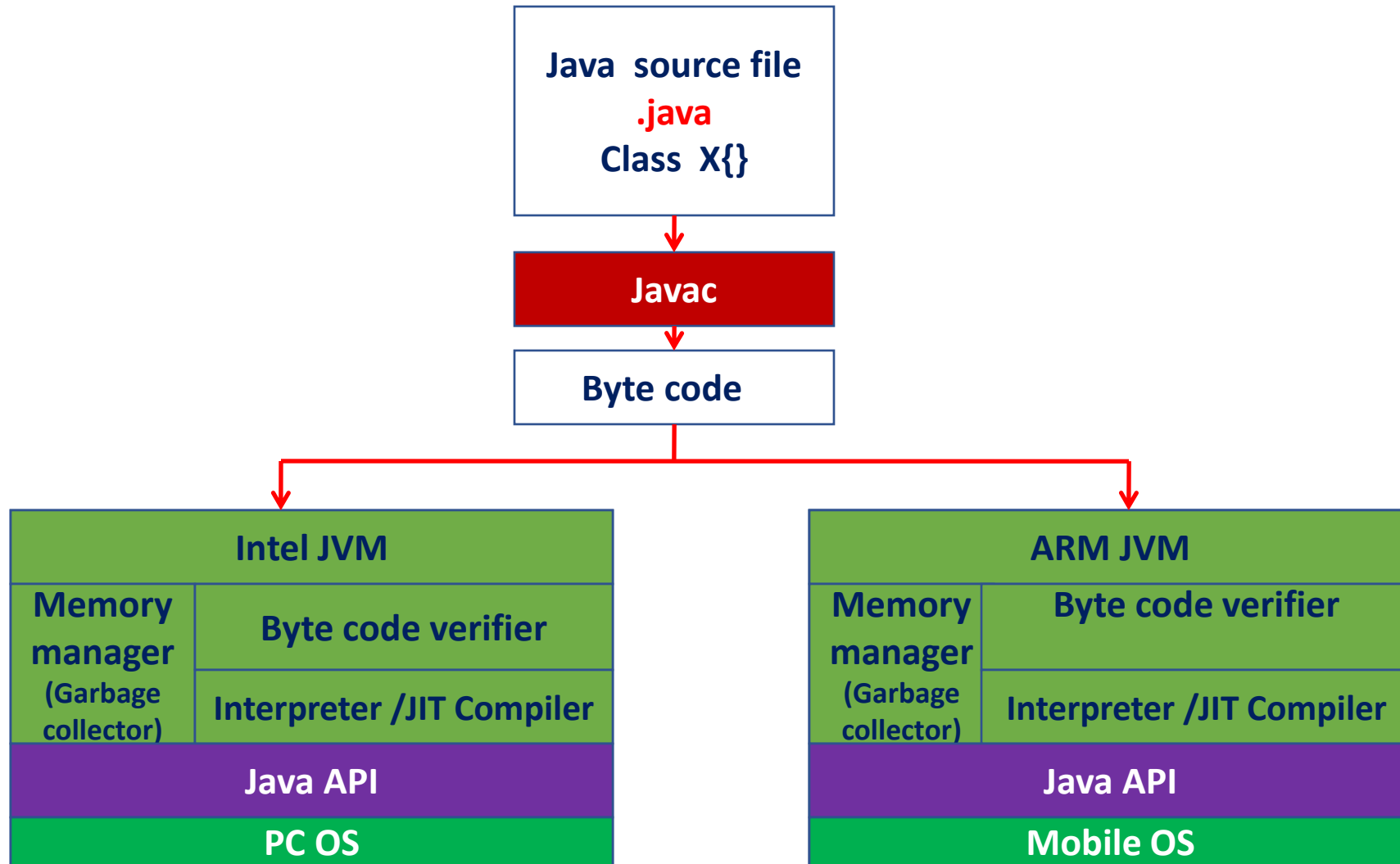
- The contents of a comment are ignored by the compiler.
- Java supports three styles of comments.
 - multiline comment.
 - begin with `/*` and end with `*/`
 - Single line comment
 - `//`
 - Documentation comment
 - begin with `/**` and end with `*/`.

```
/**  
 * <h1> sum of two numbers !</h1>  
 * program finds the sum  
 *and gives the output on  
 *the screen.  
 *  
 */
```

JAVA'S COMPONENT

- JVM: Platform-independent environment for running Java programs by loading, validating, and executing code.
- JRE: Software bundle that includes the JVM, Java class libraries, and class loader to create an environment for running Java files.
- JDK: Software development environment including a private JVM, tools, and libraries for creating Java programs

JVM Architecture



JVM architecture

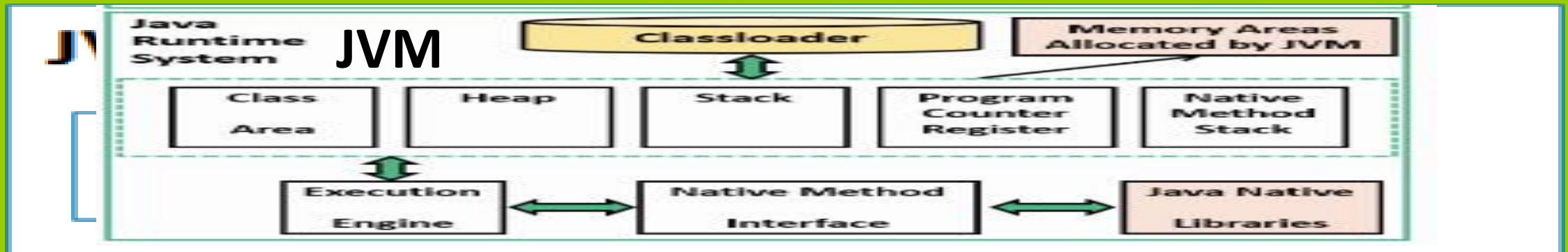
JVM

JDK

javac, jar, debugging tools,
javadoc

JRE

java, javaw, libraries,
rt.jar



JVM

- Bootstrap ClassLoader: java.lang package classes, java.net package classes, java.util package classes, java.io package classes, java.sql package classes
- Extension ClassLoader: loads the jar files located inside \$JAVA_HOME/jre/lib/ext directory.
- System/Application ClassLoader: This is the child classloader of Extension classloader. It loads the classfiles from classpath.
- Class(Method) Area Class(Method) Area stores per-class structures such as the runtime constant pool, field and method data, the code for methods.
- Heap It is the runtime data area in which objects are allocated.
- Stack Java Stackstores frames. It holds local variables and partial results, and plays a part in method invocation and

JVM

- Program Counter Register PC (program counter) register: contains the address of the Java virtual machine instruction currently being executed.
- Native Method Stack It contains all the native methods used in the application.
- Execution Engine It contains:
 - 1. A virtual processor
 - 2. Interpreter: Read bytecode stream then execute the instructions.
 - 3. Just-In-Time(JIT) compiler:
- Java Native Interface Java Native Interface (JNI) is a framework which provides an interface to communicate with another application written in another language like C, C++, Assembly etc.
 - Java uses JNI framework to send output to the Console or interact with OS libraries.

JDK VS JRE VS JVM

JDK	JRE	JVM
Java Development Kit (JDK) a software development kit used to develop Java applications.	JavaRuntime Environment (JRE) a software package that provides Java Virtual Machine (JVM), class libraries, and other components to run applications in Java.	Java Virtual Machine (JVM) an abstract machine that provides an environment for the execution of Java ByteCodes.
JDK contains tools for developing, monitoring, and debugging Java codes.	JRE contains class libraries and other supporting files required by JVM for executing Java programs.	JVM does not include any software development tools.
platform-dependent i.e. different platforms require different JDK.	platform-dependent.	platform-dependent.
Used to create an environment to write Java programs that JRE and JVM can execute.	Used to create an environment for the execution of Java programs.	JVM specifies all the implementations.
JDK = JRE + Development tools	JRE = JVM + Class libraries V P Jayachitra	JVM = provides a runtime environment.

write once and run anywhere

- It is the bytecode.
- Java compiler converts the Java programs into the .class file known as the Byte Code,
- Byte code is the intermediate language between source code and machine code.
- This bytecode is not platform-specific and can be executed on any machine.

JVM?

- JVM (Java Virtual Machine): A virtual machine that interprets Java bytecode.
- Responsibilities:
 - Loading Bytecode: Loads .class files containing Java bytecode.
 - Verifying Bytecode: Ensures that the bytecode adheres to Java's security and integrity rules.
 - Executing Bytecode: Executes the bytecode by converting it into machine code.
- Implementation: Known as the Java Runtime Environment (JRE).
- Platform Dependency: The JVM software is tailored for different operating systems, making it platform-dependent.
- Platform Independence: Despite its platform dependency, the JVM enables Java to be platform-independent by allowing the same bytecode to run on any system with a compatible JVM.

JIT Compiler

- Write Java Code: Develop your program using Java.
- Compile with javac: The Java compiler (javac) converts the source code into bytecode, which is saved in .class files.
- Load Bytecode: The Java Virtual Machine (JVM) loads the .class files at runtime.
- Interpret Bytecode: The JVM's interpreter converts the bytecode into machine-understandable instructions.
- JIT Compilation: The Just-In-Time (JIT) compiler within the JVM analyzes frequently executed methods and compiles them into optimized native machine code.
- Execute Optimized Code: The JVM executes the optimized native code directly, bypassing the need for interpretation.
- Improved Performance: This process results in faster execution and better performance of your Java program.

public static void main(String args[]) i

- Public: the public is the access modifier responsible for making the main function globally available. It is made public so that JVM can invoke it from outside the class, as it is not present in the current class.
- Static: It is a keyword to use the element without initiating the class to avoid the unnecessary allocation of the memory.
- Void: void is a keyword used to specify that a method doesn't return anything. The main method doesn't return anything.
- Main: main helps JVM to identify that the declared function is the main function to begin the execution
- String args[]: It stores Java command-line arguments and is an array of type java. lang.String class.

Java command line argument

```
class CommandLineArgs {  
    public static void main(String[] args) {  
        if (args.length > 0) {  
            System.out.println("The command line arguments are:");  
            for (String value : args)  
                System.out.println(value);  
        } else  
            System.out.println("No command line arguments found.");  
    }  
}
```

```
javac CommandLineArgs.java  
java CommandLineArgs Welcome to Java  
The command line arguments are:  
Welcome  
to  
Java
```


Can main method in java be overloaded?

Yes, the main method can be overloaded.

We can create as many overloaded main methods as we want.

However, JVM has a predefined calling method that JVM will only call the main method with the definition of

```
public static void main(string[] args)
```

Can main method in java be overloaded?

```
class Main {  
    public static void main(String[] args) {  
        System.out.println("This is the main method");  
    }  
  
    public static void main(int[] args) {  
        System.out.println("Overloaded integer array main method");  
    }  
}
```

This is the main method

What is System.out.println?

```
public final class System {  
    // Static field representing the standard output stream  
    public static final PrintStream out = new PrintStream(new  
    FileOutputStream(FileDescriptor.out));  
  
    // Other fields and methods of System class  
}
```

```
package java.io;  
public class PrintStream extends OutputStream{  
    public void println() { /* implementation */ }  
    public void println() { /* implementation */ }  
}
```

- **System** - It is a class present in java.lang package.
- **out** is the **static variable** of type PrintStream class present in the System class
- Key Members of System:
 - **out**: A static field of type PrintStream which is used for outputting text to the console.
 - **in**: A static field of type InputStream used for reading input from the console.
- **println()** is the method present in the PrintStream class.

Elements of a Java Application

Elements

Java Classes

Basic building blocks
Define structure and behavior

Constructors

Special methods for initialization
Same name as the class, no return type

Objects

Instances of classes
Hold state and perform actions

Main Method

Entry point of a Java application
`public static void main(String[] args)`

Methods

Functions within a class
Define actions and behaviors

Comments

Document code
Help with code readability

Fields (Attributes)

Variables in a class
Store data related to objects

```

public class Car {
    // Class variables
    String color;
    int modelYear;
    // Class constructor
    public Car(String color, int modelYear) {
        this.color = color;
        this.modelYear = modelYear;
    }
    // Class methods
    public void start() {
        System.out.println("The car is starting.");
    }
    public void stop() {
        System.out.println("The car is stopping.");
    }
}

```

The car is starting.
The car is stopping.

```

public class Main {
    public static void main(String[] args) {
        // Create instances of Car
        Car myCar = new Car("Red", 2024);
        Car yourCar = new Car("Blue", 2022);
        // Display details and use methods of Car
        System.out.println("My car color: " + myCar.color + ", Model year: " +
myCar.modelYear);
        myCar.start();
        myCar.stop();
        System.out.println("Your car color: " + yourCar.color + ", Model year: " +
yourCar.modelYear);
        yourCar.start();
        yourCar.stop();    }}

```

//creating instance of a class using new keyword
ClassName objectName = new ClassName();

Class vs Object

A class in Java is a blueprint or template that determines the structure and behavior of objects

An object is an instance of a class that represents a real-world entity, whereas

Aspect	Class	Object
Definition	Blueprint or template defining properties and behavior	Instance of a class with actual values
Usage	Used to create objects	Created using a class constructor
Properties	Defines properties (variables)	Holds values for those properties
Methods	Defines behavior (methods)	Can call methods to perform tasks
Memory Allocation	Not allocated memory in JVM	Allocated memory when created with new keyword

```

public class Phone {
    // Class variables (attributes)
    private String brand;
    private String model;
    private int storageCapacity; // in GB
    private double screenSize; // in inches
    private boolean isSmartphone;
    // Constructor
    public Phone(String brand, String model, int storageCapacity,
        double screenSize, boolean isSmartphone) {
        this.brand = brand;
        this.model = model;
        this.storageCapacity = storageCapacity;
        this.screenSize = screenSize;
        this.isSmartphone = isSmartphone;
    }
    // Method to display phone details
    public void displayDetails() {
        System.out.println("Phone Brand: " + brand);
        System.out.println("Phone Model: " + model);
        System.out.println("Storage Capacity: " + storageCapacity + " GB");
        System.out.println("Screen Size: " + screenSize + " inches");
        System.out.println("Smartphone: " + (isSmartphone ? "Yes" : "No"));
    }
    public void setSmartphone(boolean isSmartphone) {
        this.isSmartphone = isSmartphone;
    } }

```

```

// Method to make a call (example of functionality)
    public void makeCall(String phoneNumber) {
        System.out.println("Making a call to: " + phoneNumber);
    }
    // Method to send a message (example of functionality)
    public void sendMessage(String phoneNumber, String message) {
        System.out.println("Sending message to: " + phoneNumber);
        System.out.println("Message: " + message);
    }
}

public class Amazon {
    public static void main(String[] args) {
        // Create an instance of Phone
        Phone myPhone = new Phone("Samsung", "Galaxy S21", 128, 6.8, true);
        // Display phone details
        myPhone.displayDetails();
        // Make a call
        myPhone.makeCall("+123456789");
        // Send a message
        myPhone.sendMessage("+123456789", "Hello, this is a test message.");
    } }

```

```
// Define the class
public class ClassName {
    // Fields (Instance variables)
    private DataType variableName;
    // Constructor
    public ClassName() {
        // Initialize fields
        variableName = initialValue; // Example: message = "";
    }
    // Method to perform some action
    public void methodName(ParameterType parameter) {
        // Method body
        variableName = parameter; // Example: message = newMessage;
        System.out.println("Message posted: " + variableName);
    }
    // Method to perform another action
    public ReturnType anotherMethod() {
        // Method body
        return value; // Example: return message;
    }
}
```

```
// Main method to test the class
public static void main(String[] args) {
    // Create an instance of the class
    ClassName instanceName = new ClassName();
    // Call methods on the instance
    instanceName.methodName(parameterValue);
    // Example: app.postMessage("Hello, World!");
    instanceName.anotherMethod();
    // Example: app.readMessage();
}
}
```



```
public class WatsApp {  
  
    private String message;  
private int messageCount;  
    // Constructor  
    public SimpleMessagingApp() {  
        message = "";  
        messageCount = 0;  
    }  
    // Method to post a message  
    public void postMessage(String newMessage) {  
        message = newMessage;  
        messageCount++;  
        System.out.println("Message posted: " + newMessage);  
    }  
    // Method to read the current message  
    public void readMessage() {  
        if (messageCount > 0) {  
            System.out.println("Reading message: " + message);  
        } else {  
            System.out.println("No messages to read.");  
        }  
    }  
}
```

```
public static void main(String[] args) {  
    // Create an instance of SimpleMessagingApp  
    SimpleMessagingApp app = new SimpleMessagingApp();  
    app.postMessage("Hello, World!");  
    app.readMessage();  
    app.postMessage("How are you?");  
    app.readMessage();  
}  
}
```

Default values

Primitive Data Types:

byte: 0

short: 0

int: 0

long: 0L

float: 0.0f

double: 0.0d

char: '\u0000' (null character)

boolean: false

Reference Data Types:

Object: null

String: null

Arrays: null (since arrays are objects in Java)

Elements of a Java Application

Packages Organize related classes and interfaces Avoid name conflicts	Polymorphism Treat objects as instances of a common class Allows generic handling
Interfaces Define a set of methods without implementations Used for abstraction	Encapsulation Hide internal state Access via methods (getters and setters)
Inheritance Mechanism to reuse code Subclass inherits fields and methods from superclass	

Encapsulation

- Encapsulation groups data and methods into a class while hiding implementation-specifics and exposing a public interface.
- Encapsulation in Java includes limiting direct access by defining instance variables as private.
- There are defined public getters and setters for these variables.
- Advantages of Encapsulation in Java
 - Make a class read-only or write-only using only getter or setter methods, respectively.
 - Control data by providing logic in setter methods.
 - Hide data so other classes can't access private members directly.

```
public class User {  
    // Public username for direct access  
    public String username;  
  
    // Private password with encapsulation  
    private String password;  
  
    // Constructor  
    public User(String username, String password) {  
        this.username = username;  
        this.password = password;  
    }  
  
    // Setter for password with validation  
    public void setPassword(String password) {  
        if (password != null && password.length() >= 6) {  
            this.password = password;  
        } else {  
            System.out.println("Password must be at least 6  
characters long.");  
        }  
    }  
}
```

```
// Main method to demonstrate usage  
public static void main(String[] args) {  
    // Create a User object  
    User user = new User("Alice", "securePass123");  
  
    // Access and modify public field directly  
    user.username = "Bob";  
  
    // Access and modify private field using getter and setter  
    user.setPassword("newPass456");  
  
    // Display user information using object fields and  
    methods  
    System.out.println("Username: " + user.username);  
    System.out.println("Password: " + user.getPassword());  
}  
}
```

ENCAPSULATION

Abstraction in Java

- Abstraction in Java is achieved through interfaces and abstract classes.
- Data abstraction involves identifying only the essential characteristics of an object while ignoring irrelevant details.

```
abstract class Shape {  
    // Abstract method  
    public abstract double calculateArea();  
}
```

```
// Concrete class representing a Circle  
class Circle extends Shape {  
    private double radius; public Circle(double radius) {  
        this.radius = radius;  
    } @Override  
    public double calculateArea() {  
        return Math.PI * radius * radius; }  
}
```

```
// Concrete class representing a Rectangle  
class Rectangle extends Shape {  
    private double length;  
    private double width;  
    public Rectangle(double length, double width) {  
        this.length = length;  
        this.width = width;  
    } @Override  
    public double calculateArea() {  
        return length * width;  
    }  
}
```

```
Circle - Area: 78.53981633974483  
Rectangle - Area: 24.0
```

```
public class AbstractionExample {  
    public static void main(String[] args) {  
        // Creating objects of Circle and Rectangle  
        Circle circle = new Circle(5);  
        Rectangle rectangle = new Rectangle(4, 6); // Using the abstracted methods to calculate area  
        and perimeter  
        System.out.println("Circle - Area: " + circle.calculateArea() + ", Perimeter: " +  
            circle.calculatePerimeter());  
        System.out.println("Rectangle - Area: " + rectangle.calculateArea() + ", Perimeter: " +  
            rectangle.calculatePerimeter());  
    }  
}
```

Inheritance

- In Java, inheritance encourages the reuse of existing code and the development of hierarchies of related classes.
- The superclass or base class is the one from which the subclass inherits.
- The properties and methods of a superclass can be used by a subclass.
- To increase the functionality of the superclass, subclasses can also add additional fields and methods.


```
// Base class (superclass)
class Animal {
    String name;
    public Animal(String name) {
        this.name = name;
    }
    public void eat() {
        System.out.println(name + " is eating.");
    }
    public void sleep() {
        System.out.println(name + " is sleeping.");
    }
}
```

```
// Derived class (subclass)
class Dog extends Animal {
    public Dog(String name) {
        super(name);
    } public void bark() {
        System.out.println(name + " is barking.");
    }
}
```

Buddy is eating.
Buddy is sleeping.
Buddy is barking

```
public class InheritanceExample {
    public static void main(String[] args) {
        // Create a Dog object
        Dog myDog = new Dog("Buddy"); // Call methods from the base class (Animal)
        myDog.eat();
        myDog.sleep(); // Call a method from the derived class (Dog)
        myDog.bark();
    }
}
```

Polymorphism

- One interface or function signature may have many implementations
- It supports OOP's flexibility and code reuse.
- Method overloading and method overriding are frequently used to achieve polymorphism.
- Removing the underlying implementation details facilitates code maintenance and improves code readability.
- To achieve dynamic binding, also known as late binding, where the exact method to be executed is determined at runtime, polymorphism is an essential concept.

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

```
class Circle extends Shape {  
    @Override  
    public void draw() {  
        System.out.println("Drawing a circle");  
    }  
}
```

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

```
public class PolymorphismExample {  
    public static void main(String[] args) {  
        Shape shape1 = new Circle();  
        Shape shape2 = new Rectangle();  
        shape1.draw(); // Calls the draw() method of Circle  
        shape2.draw(); // Calls the draw() method of Rectangle  
    }  
}
```

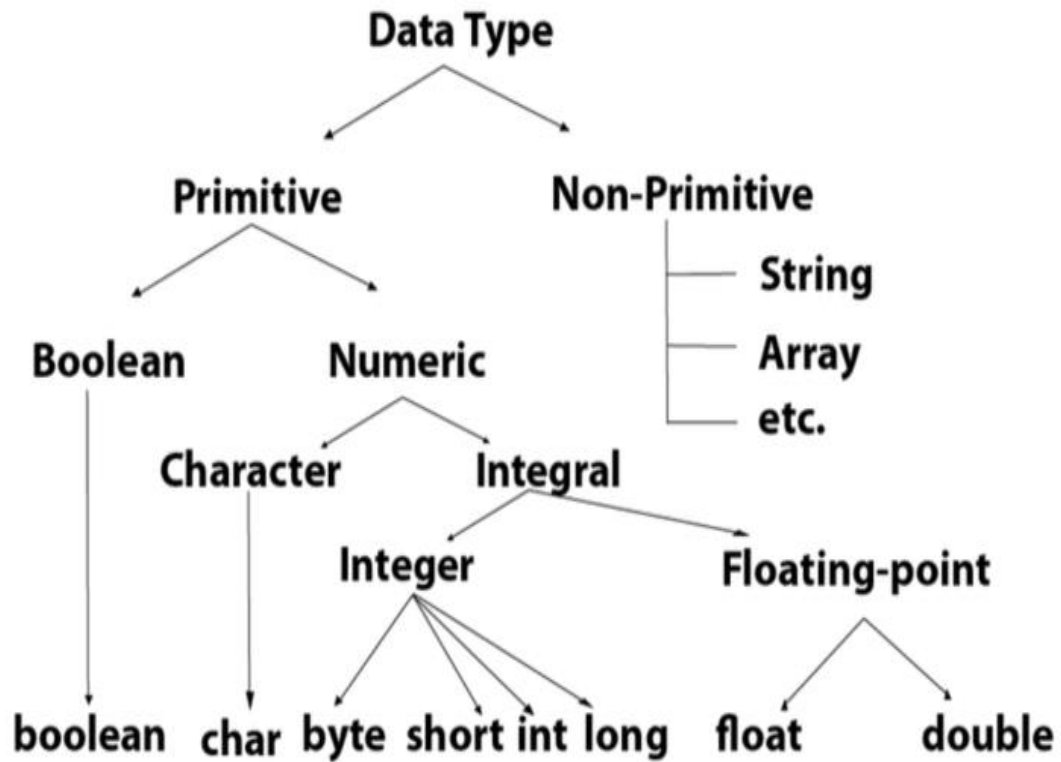
String literals vs String Object

```
public class StringExample {  
    public static void main(String[] args) {  
        String s = "sasf";    // String literal  
        String s1 = new String("sasf"); // String object  
  
        // Compare references  
        System.out.println(s == s1); // false, different references  
        System.out.println(s.equals(s1)); // true, same value  
    }  
}
```

String literals vs String Object

```
public class StringAddressExample {  
    public static void main(String[] args) {  
        String s = "sasf";    // Step 1: String literal  
        String s1 = new String("sasf"); // Step 2: String object  
  
        // Print references  
        System.out.println("Address of s: " + System.identityHashCode(s));  
        System.out.println("Address of s1: " + System.identityHashCode(s1));  
  
        // Compare references and values  
        System.out.println("s == s1: " + (s == s1)); // false, different references  
        System.out.println("s.equals(s1): " + s.equals(s1)); // true, same value  
    }  
}
```

Data Type



<i>Data Type</i>	<i>Default Value</i>	<i>Default size</i>
<i>boolean</i>	<i>false</i>	<i>1 bit</i>
<i>char</i>	<i>'\u0000'</i>	<i>2 byte</i>
<i>byte</i>	<i>0</i>	<i>1 byte</i>
<i>short</i>	<i>0</i>	<i>2 byte</i>
<i>int</i>	<i>0</i>	<i>4 byte</i>
<i>long</i>	<i>0L</i>	<i>8 byte</i>
<i>float</i>	<i>0.0f</i>	<i>4 byte</i>
<i>double</i>	<i>0.0d</i>	<i>8 byte</i>

Java vs python

- Java: Suitable for web development, Big Data, and IoT applications in addition to mobile apps.
- Python: used in machine learning, image processing, multimedia applications, and other fields.

CS6308- Java Programming

V P Jayachitra

Assistant Professor

Department of Computer Technology

MIT Campus

Anna University

Java's fundamental elements

- Data types
- Variables
- Arrays

Data types

- **No automatic coercions or conversions of conflicting types.**
- **Primitive data type**
- **Non primitive type**

Primitive data type

- Java defines eight primitive types of data: **byte, short, int, long, char, float, double, and boolean**

type	size	range
byte	8 bits	-128 to 127
short	16 bits	-32,768 to 32,767
int	32 bits	-2,147,483,648 to 2,147,483,647
long	64 bits	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
double	64 bits	-4.9e-324 to 1.8e+308
float	32 bits	-1.4e-045 to 3.4e+038
char	16 bits	0 to 65,536
boolean	1 bit or byte *	true or false

Integer literals

- When a literal value is assigned to a byte or short variable, no error is generated if the literal value is within the range of the target type.
 - An integer literal can always be assigned to a long variable.
 - **long literal ?**
 - you will need to explicitly tell the compiler that the literal value is of type long.
 - Do this by appending an upper- or lowercase L to the literal.
 - For example, 0x7ffffffffffffL or 9223372036854775807L is the largest long.
 - An integer can also be assigned to a char as long as it is within range.
 - Decimal numbers cannot have a leading zero. Therefore signify a hexadecimal constant with a leading zero-x, (0x or 0X).
 - `int x = 0b1010; //0B or 0b for binary`
 - embed one or more underscores in an integer literal. cannot come at the beginning or the end of a literal
 - `int x = 123_456_789; // x will be 123456789. The underscores will be ignored.`
 - `int x = 123___456___789; // x will be 123456789. The underscores will be ignored.`
 - `int x = 0b1101_0101_0001_1010; // x will be 54554`

Float literal

- a float literal, must append an *F* or *f* to the constant.
- A double literal, must append a *D* or *d* to the constant.
- *When the literal is compiled, the underscores are discarded.*
- `double num = 9_423_497.1_0_9; // 9423497.109.`
- `double num = 9_423_497_862.0;`

Boolean Literals

- **Two logical values that a boolean value can have, true and false.**
- **The values of true and false do not convert into any numerical representation.**
- **The true literal in Java does not equal 1, nor does the false literal equal 0.**

Character Literals

- A literal character is represented inside a pair of **single quotes**.
- All of the visible ASCII characters can be directly entered inside the quotes, such as 'a', 'z', and '@'.
- For characters that are impossible to enter directly, use **escape sequences** that allow you to enter the character
 - ' \' ' for the single-quote character itself and ' \n ' for the newline character.
- For **octal** notation, use the backslash followed by the three-digit number.
 - For example, ' \141 ' is the letter 'a'.
- enter a backslash-u (\u), then exactly four **hexadecimal** digits.
 - For hexadecimal example, ' \u0061 ' is the ISO-Latin-1 'a'
 - ' \u432 ' is a Japanese Katakana character.

Escape characters	description
\ddd	octal
\uxxxx	Unicode
'	single quote
"	double quote
\	backslash
\r	Carriage return
\n	New line
\f	Form feed
\t	Tab
\b	backspace

String Literals

- String literals in Java are specified by enclosing a sequence of characters between a pair of double quotes.
- Examples of string literals are
 - "Hello World"
 - "two\nlines"
 - " \"This is in quotes\""

Variables

- The variable is the basic unit of storage in a Java program.
- A variable is defined by the combination of an identifier, a type, and an optional initializer.
- In addition, all variables have a scope, which defines their visibility, and a lifetime.

Declaring a Variable

- In Java, all variables must be declared before they can be used. The basic form of a variable declaration is shown here:

```
type identifier [ = value ][, identifier [= value ] ...];
```

Variables contd...

Variable declarations of various types.

```
public class VariableDeclarations {
    public static void main(String[] args) {
        // Declare and initialize variables in a single line
        int a, b, c;           // Multiple declarations for int
        int d = 1, e, f = 10;  // Initialize d and f, declare e
        byte g = 12;          // Initialize byte variable g
        double pi = 3.14159;   // Initialize double variable pi
        char x = 'x';          // Initialize char variable x
        boolean done = false;  // Initialize boolean variable done
        float h = 10.0f;       // Initialize float variable h (include 'f' suffix)
        long l = 9876543210L;  // Initialize long variable l (include 'L' suffix)
        // Print the variables to verify
        System.out.println("int a: " + a);
        System.out.println("int b: " + b);
        System.out.println("int c: " + c);
        System.out.println("int d: " + d);
        System.out.println("int e: " + e);
        System.out.println("int f: " + f);
        System.out.println("byte g: " + g);
        System.out.println("double pi: " + pi);
```

```
        System.out.println("char x: " + x);
        System.out.println("boolean done: " + done);
        System.out.println("float h: " + h);
        System.out.println("long l: " + l);
    }
}
```

Variables contd...

Dynamic Initialization

```
import java.util.Scanner;

public class CircleMath {
    public static void main(String[] args) {
        // Create a Scanner object for user input
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter the radius of the circle: ");
        double radius = scanner.nextDouble(); // Read user input
        double area = Math.PI * radius * radius;
        double circumference = 2 * Math.PI * radius;
        System.out.printf("For a circle with radius %.2f:\n", radius);
        System.out.printf("Area: %.2f\n", area);
        System.out.printf("Circumference: %.2f\n", circumference);
    }
}
```

```
Enter the radius of the circle: 5
For a circle with radius 5.00:
Area: 78.54
Circumference: 31.42
```

Scope:Block Scope

```
class ScopeExample {  
    public static void main(String[] args[]) {  
        int alpha; // known to all code within main  
        alpha = 15;  
        if (alpha == 15) { // start new scope  
            int beta = 25; // known only to this block  
            // alpha and beta both known here  
            System.out.println("alpha and beta: " + alpha + " " + beta);  
            alpha = beta * 2;  
        }  
        // beta = 100;  
        // Error! beta not known here  
        // alpha is still known here  
        System.out.println("alpha is " + alpha);  
    }  
}
```

alpha is 15

Scope

```
class VariableLifetime {  
    public static void main(String[] args[]) {  
        int i;  
        for (i = 0; i < 3; i++) {  
            int j = -5; // j is initialized each time block is entered  
            System.out.println("j is: " + j);  
            // this always prints -5  
            j = 200;  
            System.out.println("j is now: " + j);  
        }  
    }  
}
```

```
j is now: 200  
j is now: 200  
j is now: 200
```

Scope

- Although blocks can be nested, you cannot declare a variable to have the same name as one in an outer scope.

```
// This program will not compile
class ScopeError {
    public static void main(String args[]) {
        int foo = 1;
        {           // creates a new scope
            int foo = 2; // Compile-time error - foo already defined!
        }
    }
}
```

java: variable foo is already defined in method main(java.lang.String[])

Type Conversion and Casting

- If the two types are compatible, then Java will perform the conversion automatically.
 - For example, it is always possible to assign an int value to a long variable.
 - For instance, there is no automatic conversion defined from double to byte.
 - use a *cast*, which performs an explicit conversion between incompatible types.
- Java's Automatic Conversions: no explicit cast statement is required.
 - The two types are compatible.
 - The destination type is larger than the source type.
 - no automatic conversions from the numeric types to char or boolean.
 - performs an automatic type conversion when storing a literal integer constant into variables of type byte, short, long, or char.

Casting Incompatible Types

- what if you want to assign an int value to a byte variable?
- This conversion will not be performed automatically, because a byte is smaller than an int.
- This kind of conversion is sometimes called a **narrowing conversion**
 - explicitly making the value narrower so that it will fit into the target type.
- To create a conversion between two incompatible types, you must use a cast.
- A cast is simply an explicit type conversion. It has this general form:

- (target-type) value

```
int a;  
byte b;  
// ...  
b = (byte) a;
```


Casting Incompatible Types

```
class TypeConversion
public static void main(String args[]) {
    byte a;
    int m = 257;
    double n = 323.142;

    System.out.println("\nConversion of int to byte.");
    a = (byte) m;
    System.out.println("m and a " + m + " " + a);

    System.out.println("\nConversion of double to int.");
    m = (int) n;
    System.out.println("n and m " + n + " " + m);

    System.out.println("\nConversion of double to byte.");
    a = (byte) n;
    System.out.println("n and a " + n + " " + a);
}
```

Conversion of int to byte.
m and a 257 1

Conversion of double to int.
n and m 323.142 323

Conversion of double to byte.
n and a 323.142 67

• Automatic Type Promotion in Expressions

238.14 + 515 - 126.3616
result = 626.7784146484375

```
class TypePromotion {  
    public static void main(String args[]) {  
        byte p = 42;  
        char q = 'a';  
        short r = 1024;  
        int s = 50000;  
        float t = 5.67f;  
        double u = .1234;  
        double result = (t * p) + (s / q) - (u * r);  
        System.out.println((t * p) + " + " + (s / q) + " - " + (u * r));  
        System.out.println("result = " + result);  
    }  
}
```

• Type Promotion Rules

- First, all byte, short, and char values are promoted to int, as just described.
- Then, if one operand is a long, the whole expression is promoted to long.
- If one operand is a float, the entire expression is promoted to float.
- If any of the operands are double, the result is double.

Arrays

- **An array is a group of like-typed variables that are referred to by a common name.**
- **Arrays of any type can be created and may have one or more dimensions.**
- **A specific element in an array is accessed by its index.**

One-Dimensional Arrays

```
type var-name[ ];  
array-var = new type [size];
```

```
int month_days[];  
month_days = new int[12];
```

```
int month_days[] = new int[12];
```

```
public class ComputeAverage {  
    public static void main(String[] args) {  
        double[] values = {10.1, 11.2, 12.3, 13.4, 14.5};  
        double sum = 0;  
        int count;  
        for (count = 0; count < 5; count++)  
            sum = sum + values[count];  
        System.out.println("Average is " + sum / 5);  
    }  
}
```

Average is 12.3

```
public class MonthDaysArray {  
    public static void main(String[] args) {  
        int[] daysInMonth = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };  
        System.out.println("April has " + daysInMonth[3] + " days.");  
    }  
}
```

April has 30 days.

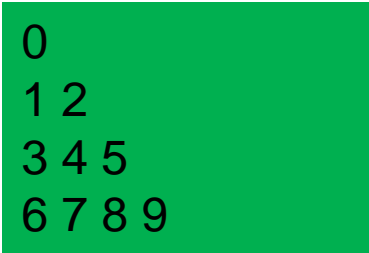
Multidimensional Arrays

```
int twoD[][] = new int[4][5];
```

```
public class TwoDimensionalArrayDemo {  
    public static void main(String[] args) {  
        int[][] matrix = new int[4][5];  
        int row, col, value = 0;  
  
        for (row = 0; row < 4; row++)  
            for (col = 0; col < 5; col++) {  
                matrix[row][col] = value;  
                value++;  
            }  
  
        for (row = 0; row < 4; row++) {  
            for (col = 0; col < 5; col++)  
                System.out.print(matrix[row][col] + " ");  
            System.out.println();  
        }  
    }  
}
```

```
0 1 2 3 4  
5 6 7 8 9  
10 11 12 13 14  
15 16 17 18 19
```

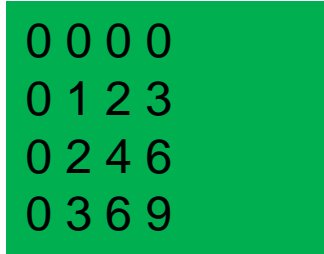
```
class IrregularArray {  
    public static void main(String[] args[]) {  
        int irregular[][] = new int[4][];  
        irregular[0] = new int[1];  
        irregular[1] = new int[2];  
        irregular[2] = new int[3];  
        irregular[3] = new int[4];  
  
        int row, col, element = 0;  
  
        for(row=0; row<4; row++)  
            for(col=0; col<row+1; col++) {  
                irregular[row][col] = element;  
                element++;  
            }  
        for(row=0; row<4; row++) {  
            for(col=0; col<row+1; col++)  
                System.out.print(irregular[row][col] + " ");  
            System.out.println();  
        }  
    }  
}
```



```
0  
1 2  
3 4 5  
6 7 8 9
```

```
// Initialize a two-dimensional array.
class MatrixInitialization {
    public static void main(String[] args[]) {
// Initialize a 4x4 two-dimensional array
        double grid[][] = {
            { 0*0, 1*0, 2*0, 3*0 },
            { 0*1, 1*1, 2*1, 3*1 },
            { 0*2, 1*2, 2*2, 3*2 },
            { 0*3, 1*3, 2*3, 3*3 }
        };
        int r, c;

        for(r=0; r<4; r++) {
            for(c=0; c<4; c++)
                System.out.print(grid[r][c] + " ");
            System.out.println();
        }
    }
}
```



0	0	0	0
0	1	2	3
0	2	4	6
0	3	6	9

Alternative Array Declaration Syntax

```
type[ ] var-name;
```

```
int al[] = new int[3];  
int[] a2 = new int[3];
```

```
char twod1[][] = new char[3][4];  
char[][] twod2 = new char[3][4];
```

```
int[] nums, nums2, nums3; // create three arrays  
int nums[], nums2[], nums3[]; // create three arrays
```


Introducing Type Inference with Local Variables

- Recently, an exciting new feature called *local variable type inference* was added to the Java language.
- In the past, all **variables** required an explicitly declared type, whether they were initialized or not.
- Beginning with JDK 10, it is now possible to let the compiler infer the type of a local variable based on the type of its initializer, thus avoiding the need to explicitly specify the type.
- Local variable type inference offers a number of advantages.
 - eliminating the need to redundantly specify a variable's type when it can be inferred from its initializer.
 - It can simplify declarations in cases in which the type name is quite lengthy, such as can be the case with some class names.

Introducing Type Inference with Local Variables

```
double avg = 10.0;  
var avg = 10.0;
```

```
var myArray = new int[10]; // This is valid.  
var[] myArray = new int[10]; // Wrong  
var myArray[] = new int[10]; // Wrong  
var counter; // Wrong! Initializer required.  
var myArray = new int[10]; // This is valid.  
var myArray = new int[10]; // This is valid.
```

```
public class TypeInferenceExample {  
    public static void main(String[] args) {  
  
        // Using type inference with 'var'  
        var d = 5;      // Inferred as int  
        var e = 5.5f;   // Inferred as float  
        var f = "Java"; // Inferred as String  
        System.out.println("Inferred int: " + d);  
        System.out.println("Inferred float: " + e);  
        System.out.println("Inferred String: " + f);  
    }  
}
```

```
Inferred int: 5  
Inferred float: 5.5  
Inferred String: Java
```

Strings

- **String, is not a primitive type**

```
String str = "this is a test";  
System.out.println(str);
```

Round a number using format

```
public class Decimal {  
    public static void main(String[] args) {  
        double num = 1.234567;  
        System.out.format("%.2f", num);  
    }  
}
```

Formatted output in Java

```
class JavaFormat
{
    public static void main(String args[])
    {
        int x = 10;
        System.out.printf("Print integer: x = %d\n", x);

        // print it upto 2 decimal places
        System.out.printf("Formatted with precision: PI = %.2f\n", Math.PI);
        float n = 5.2f;
        // automatically appends zero to the rightmost part of decimal
        System.out.printf("Formatted to specific width: n = %.4f\n", n);
        n = 2324435.3f;

        // width of 20 characters
        System.out.printf("Formatted to right margin: n = %20.4f\n", n);
    }
}
```

CS6308- Java Programming

V P Jayachitra

Assistant Professor

Department of Computer Technology

MIT Campus

Anna University

Variables

- A variable is a storage location used to hold data values that can be modified and accessed during the execution of a program.
- Allowed Characters:
 - Letters: Uppercase (A-Z) and lowercase (a-z)
 - Digits: (0-9), but not as the first character
 - Underscores: (_)
 - Dollar Sign: (\$), though its use is generally discouraged for regular variable names

```
public class Variables {  
    public static void main(String[] args) {  
        // Valid variable names  
        // Valid: starts with a letter, only contains letters & digits  
        int userAge = 70;  
        // Valid: starts with an underscore, contains letters  
        String _userName = "James Gosling";  
        // Valid: contains a dollar sign, starts with a letter  
        double annualSalary$ = 10000000.50;  
  
        // Invalid variable names (cause errors)  
        /*  
        int 1stVariable = 100;      // Invalid: starts with a digit  
        String user-age = "Alice"; // Invalid: contains a hyphen  
        double @salary = 199.99;   // Invalid: contains @ symbol  
        boolean user Name = true;  // Invalid: contains a space  
        */  
    }  
}
```

- **Starting Character:** Variable names must start with a letter (A-Z or a-z) or an underscore (_)
- **Subsequent Characters:** After the first character, variable names can include letters, digits, underscores, and dollar signs.
- **Case Sensitivity:** Java is case-sensitive(myVariable, MyVariable, and MYVARIABLE are different variables)
- **Reserved Words:** Variable names cannot be Java reserved keywords.

Variable types

In Java programming the main types of variables are local, instance, class and parameters.

	Local Variables	Instance Variables or Non-static fields	Class Variables or Static Variables	Parameters
Scope:	Declared inside methods, constructors, or blocks.	Declared within a class but outside methods.	Declared with the static keyword within a class.	Declared in method or constructor definitions.
Lifetime:	Exist only during the execution of the method or block where they are declared.	Exist as long as the object of the class exists.	Exist for the duration of the program and are shared among all instances of the class.	Exist only during the execution of the method or constructor
Initialization:	Must be initialized before use.	Automatically initialized to default values if not explicitly initialized.	Automatically initialized to default values if not explicitly initialized.	Initialized with the values passed to the method or constructor when it is called.


```

public class Variable {
    // Class variable (static variable)
    static int classVariable; // Default value: 0
    // Instance variable
    int instanceVariable; // Default value: 0
    // Constructor with parameter
    public Variable(int instanceVariable) {
        // Initialize instance variable via constructor parameter
        this.instanceVariable = instanceVariable;
    }
    public void method() {
        // Local variable
        int localVariable = 20; // must be initialized before use
        System.out.println("Class Variable: " + classVariable);
        System.out.println("Instance Variable:" + instanceVariable);
        System.out.println("Local Variable: " + localVariable);
    }
}

```

```

public static void main(String[] args) {
    // Create two instances of Variable
    Variable obj1 = new Variable(10);
    Variable obj2 = new Variable(20);
    // Modify the class variable
    Variable.classVariable = 100;
    // instance variables
    System.out.println("obj1");
    System.out.println("#####");
    obj1.method();
    System.out.println("obj2");
    System.out.println("#####");
    obj2.method();
    // class variable
    System.out.println("#####");
    System.out.println("Class Variable:" + Variable.classVariable);
    // instance variables from each object
    System.out.println("obj1: " + obj1.instanceVariable);
    System.out.println("obj2: " + obj2.instanceVariable);
}
}

```

```

obj1
#####
Class Variable: 100
Instance Variable: 10
Local Variable: 20
obj2
#####
Class Variable: 100
Instance Variable: 20
Local Variable: 20
#####
Class Variable : 100
obj1: 10
obj2: 20

```

Operators

- Java provides a rich operator environment.
- Most of its operators can be divided into the following four groups:
 - Arithmetic
 - Bitwise
 - Relational
 - Logical

Arithmetic Operators

- The operands of the arithmetic operators must be of a numeric type.
- Cannot use them on **boolean** types, but you can use them on **char** types, since the **char** type in Java is, essentially, a subset of **int**.
- Integer Division: Result is an integer. Any fractional part is discarded.
- Floating-Point Division: Retain fractional results.

OPERATOR	Description
+, -, *, /, %	Addition, subtraction, multiplication, division, modulus
+, -	Unary plus, unary minus
++, --	Increment, Decrement
+=, -=, *=, /=, %=	Addition assignment, subtraction assignment, multiplication assignment, division assignment, modulus assignment

The Bitwise Operators

- Java defines several *bitwise operators* that can be applied to the integer types: **long**, **int**, **short**, **char**, and **byte**.
- These operators act upon the individual bits of their operands.

OPERATOR	Description
~	Bitwise unary NOT
&, , ^	Bitwise AND, OR, EXCLUSIVE OR
>>, <<, >>>	Shift Right, Shift Left, Shift Right with zero fill
&=, =, ^=	Bitwise AND assignment, Bitwise OR assignment, Bitwise Exclusive OR assignment
>>=, <<=, >>>=	Shift Right Assignment, Shift Left Assignment, Shift Right with zero fill Assignment

```

public class BitwiseOperators {
    public static void main(String[] args) {
        int a = 5; // binary: 0101
        int b = 3; // binary: 0011
        // Bitwise Operators
        System.out.println("Bitwise AND: " + (a & b)); // 1 (binary 0001)
        System.out.println("Bitwise OR: " + (a | b)); // 7 (binary 0111)
        System.out.println("Bitwise XOR: " + (a ^ b)); // 6 (binary 0110)
        System.out.println("Bitwise Complement: " + (~a)); // -6 (binary 11111010)
        // Bitwise Assignment Operators
        a &= b;
        System.out.println("Bitwise AND Assignment: " + a); // 1
        a = 5; // Resetting
        a |= b;
        System.out.println("Bitwise OR Assignment: " + a); // 7
        a = 5; // Resetting
        a ^= b;
        System.out.println("Bitwise XOR Assignment: " + a); // 6
        // Shift Operators
        System.out.println("Left Shift: " + (a << 1)); // 12 (binary 1100)
        System.out.println("Right Shift: " + (a >> 1)); // 3 (binary 0011)
        System.out.println("Unsigned Right Shift: " + (a >>> 1)); // 3 (binary 0011)
        // Negative number example for unsigned right shift
        int negNum = -5;
        System.out.println("Unsigned Right Shift of -5: " + (negNum >>> 1));
    }
}

```

```

Bitwise AND: 1
Bitwise OR: 7
Bitwise XOR: 6
Bitwise Complement: -6
Bitwise AND Assignment: 1
Bitwise OR Assignment: 7
Bitwise XOR Assignment: 6
Left Shift: 12
Right Shift: 3
Unsigned Right Shift: 3
Unsigned Right Shift of -5: 2147483642

```

```

class BitLogic {
    public static void main(String args[]) {
String binary[] = { "0000", "0001", "0010", "0011", "0100", "0101", "0110", "0111",
    "1000", "1001", "1010", "1011", "1100", "1101", "1110", "1111" };
    int a = 3;
    int b = 6;
    int or = a | b;
    int and = a & b;
    int xor = a ^ b;
    int xnor = (~a & b) | (a & ~b);
    int not = ~a & 0x0f;

    System.out.println("    a = " + binary[a]);
    System.out.println("    b = " + binary[b]);
    System.out.println("    (a | b) or= " + binary[or]);
    System.out.println("    (a & b) and= " + binary[and]);
    System.out.println("    (a ^ b) xor = " + binary[xor]);
    System.out.println("    (~a & b | a & ~b) xnor = " + binary[xnor]);
    System.out.println("    ~a = " + binary[not]);
    }
}

```

```

a = 0011
b = 0110
(a | b) or= 0111
(a & b) and= 0010
(a ^ b) xor = 0101
(~a & b | a & ~b) xnor = 0101
~a = 1100

```

Relational Operators

- The *relational operators* determine the relationship that one operand has to the other. Specifically, they determine equality and ordering. outcome of these operations is a **boolean** value.
- only integer, floating-point, and character operands may be compared to see which is greater or less than the other.

```
int done;  
//...  
if(!done)... // Valid in C/C++  
if(done)... // but not in Java.
```

```
if(done == 0)... // This is Java-style.  
if(done != 0)...
```

- The reason is that Java does not define true and false in the same way as C/C++. In C/C++, true is any nonzero value and false is zero.
- In Java, **true** and **false** are nonnumeric values that do not relate to zero or nonzero. Therefore, to test for zero or nonzero, you must explicitly employ one or more of the relational operators.

Boolean Logical Operators

- The Boolean logical operators shown here operate only on **boolean** operands. All of the binary logical operators combine two **boolean** values to form a resultant **boolean** value.

OPERATOR	Description
!, &, , ^	Logical Unary NOT, Logical AND, Logical OR, Logical XOR
&=, =, ^=	Logical AND assignment, Logical OR assignment, Logical XOR assignment
, &&	Short-circuit OR, short-circuit AND
==, !=	Equal To, Not Equal To
?:	Ternary(If Then Else)

Boolean Logical Operators

- The logical Boolean operators, **&**, **|**, and **^**, operate on **boolean** values. The logical **!** operator inverts the Boolean state:

!true==false

!false == true.

// Demonstrate the boolean logical operators.

```
class BooleanLogic {  
    public static void main(String args[]) {  
        boolean a = true;  
        boolean b = false;  
        boolean c = a | b;  
        boolean d = a & b;  
        boolean e = a ^ b;  
        boolean f = (!a & b) | (a & !b);  
        boolean g = !a;  
        System.out.println("a = " + a);  
        System.out.println("b = " + b);  
        System.out.println("a | b = " + c);  
        System.out.println("a & b = " + d);  
        System.out.println("a ^ b = " + e);  
        System.out.println("!a & b | a & !b = " + f);  
        System.out.println("!a = " + g);  
    }  
}
```

a = true
b = false
a | b = true
a & b = false
a ^ b = true
!a & b | a & !b = true
!a = false

Short-Circuit Logical Operators

Short-Circuit OR (||):

- Evaluates the right-hand operand only if the **left-hand operand** is **false**
- Prevent unnecessary computations or avoid errors (e.g., check if an object is null before accessing its methods).

Short-Circuit AND (&&):

- Evaluates the right-hand operand only if the **left-hand operand** is **true**.
- Avoid operations that could cause errors or be inefficient if the result is already determined by the left-hand operand.

Advantages of Short-Circuit Operators:

Efficiency: Reduces computational overhead by avoiding unnecessary evaluations.

Error Prevention: Avoids runtime errors or exceptions (e.g., NullPointerException) by ensuring certain conditions are met before executing potentially risky code.

```
public class ShortCircuitEffects {  
    private static int globalCounter = 0;  
    public static void main(String[] args) {  
        boolean condition1 = false;  
        boolean condition2 = true;  
        // Using short-circuit AND  
        if (condition1 && performSideEffect()) {  
            System.out.println("Condition met with AND");  
        } else {  
            System.out.println("Condition not met with AND");  
        }  
        // Using short-circuit OR  
        if (condition2 || performSideEffect()) {  
            System.out.println("Condition met with OR");  
        } else {  
            System.out.println("Condition not met with OR");  
        }  
        System.out.println("Global Counter: " + globalCounter);  
    }  
    private static boolean performSideEffect() {  
        globalCounter++;  
        System.out.println("Side effect executed");  
        return true;  
    }  
}
```

Condition not met with AND
Condition met with OR
Global Counter: 1

```

public class ShortCircuitExample {
    private static int globalCounter = 0;

    public static void main(String[] args) {
        boolean conditionA = false;
        boolean conditionB = false;

        // Using short-circuit OR (||)
        if (conditionA || performSideEffect()) {
            System.out.println("Condition met with OR");
        } else {
            System.out.println("Condition not met with OR");
        }

        // Print the final value of globalCounter
        System.out.println("Global Counter: " + globalCounter);
    }

    private static boolean performSideEffect() {
        // Side effect: modifying a global variable
        globalCounter++;
        System.out.println("Side effect executed");
        return true;
    }
}

```

Avoids Unintended State Changes

If the right-hand side of a logical expression (&& or ||) has side effects, and the left-hand side of the expression already determines the result, the right-hand side might not be executed.

This avoids unintended changes or operations

Side effect executed
Condition met with OR
Global Counter: 1

If conditionA is true, performSideEffect() is not called due to short-circuit evaluation.
This prevents the global variable globalCounter from being incremented if it's not necessary

```

public class ShortCircuitExample {
    public static void main(String[] args) {
        boolean isValid = true;

        // Using short-circuit AND (&&)
        if (isValid && expensiveComputation()) {
            System.out.println("Condition met with AND");
        } else {
            System.out.println("Condition not met with AND");
        }
    }

    private static boolean expensiveComputation() {
        // Simulate an expensive computation
        System.out.println("Expensive computation started...");
        try {
            Thread.sleep(1000); // 1 second delay
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Expensive computation finished.");
        return false;
    }
}

```

```

Expensive computation started...
Expensive computation finished.
Condition not met with AND

```

Avoids expensive computation

Short-circuit evaluation can enhance performance by avoiding expensive or time-consuming operations when the result of the expression is already known.

short-circuit AND (&&) avoid an expensive computation when it is not necessary.

If isValid were false, the expensiveComputation() method would not be called at all, which is a key benefit of using short-circuit logical operators; the output is

```
Condition not met with AND
```

```

public class ShortCircuitExample {
    public static void main(String[] args) {
        String str = null;

        // Using short-circuit AND (&&) to avoid NullPointerException
        if (str != null && str.length() > 5) {
            System.out.println("String is longer than 5 characters");
        } else {
            System.out.println("String is null or not longer than 5
characters");
        }
    }
}

```

String is null or not longer than 5 characters

```

public class ShortCircuitExample {
    public static void main(String[] args) {
        String str = null;

        // Using short-circuit AND (&&) to avoid
        NullPointerException
        if (str != null || str.length() > 5) {
            System.out.println("String is longer than 5 characters");
        } else {
            System.out.println("String is null or not longer than 5
characters");
        }
    }
}

```

Exception in thread "main"
java.lang.NullPointerException: Cannot invoke
"String.length()" because "str" is null
at
ShortCircuitExample.main(ShortCircuitExample.java:10)

Avoid Exceptions:

Short-circuit operators can prevent exceptions by ensuring that the second operand is only evaluated if necessary.

The Assignment Operator

- The *assignment operator* is the single equal sign, =. The assignment operator works in Java much as it does in any other computer language. It has this general form:
- *var = expression;*
- Here, the type of *var* must be compatible with the type of *expression*.
- `int x, y, z;`

```
x = y = z = 100; // set x, y, and z to 100
```

The ? Operator

- Java includes a special ternary (three-way) operator that can replace certain types of if-then-else statements.
- This operator is the ?. It can seem somewhat confusing at first, but the ? can be used very effectively once mastered. The ? has this general form:

condition ? expression1 : expression2

- condition: An expression that evaluates to a boolean value.
 - expression1: Evaluated and returned if condition is true.
 - expression2: Evaluated and returned if condition is false
- The result of the ? operation is that of the expression evaluated. Both expression2 and expression3 are required to return the same (or compatible) type, which can't be void.
 - Here is an example of the way that the ? is employed:

```
ratio = denom == 0 ? 0 : num / denom;
```


The ?: Operator

```
public class TernaryExample {  
    public static void main(String[] args) {  
        int i1 = 10; // Example 1  
        int i2 = -10; // Example 2  
  
        int absValue1 = (i1 < 0) ? -i1 : i1;  
        int absValue2 = (i2 < 0) ? -i2 : i2;  
  
        System.out.println("Absolute value of " + i1 + " is " + absValue1);  
        System.out.println("Absolute value of " + i2 + " is " + absValue2);  
    }  
}
```

```
Absolute value of 10 is 10  
Absolute value of -10 is 10
```

Operator Precedence

- In Java, operators have a specific order of precedence, determining how expressions are evaluated.
- The order of precedence from highest to lowest is as follows.
- Although [], (), and . are technically separators, they also function as operators with the highest precedence when used for array access, method calls, and field access.
- Binary Operations: Evaluated from left to right.
- Assignment Operators: Evaluated from right to left

Operator Precedence

Highest

++ (postfix)	-- (postfix)					
++ (prefix)	-- (prefix)	~	!	+ (unary)	-(unary)	(Type-cast)
*	/	%				
+	-					
>>	<<	>>>				
>	>=	<	<=	instanceof		
==	!=					
&						
^						
&&						
?:						
->						
=	op= (+=, -=, *=, /=, %=, &=, =, ^=, <<=, >>=, and >>>=)					

Lowest

```

public class OperatorPrecedence {
    public static void main(String[] args) {
        int a = 5;
        int b = 10;
        int c = 2;
        // Postfix and prefix operators
        System.out.println(a++ + " " + a);
        // Prefix increment
        System.out.println(++b + " " + b);
        // Unary operators
        System.out.println("\nUnary operators:");
        System.out.println("Bitwise complement of 5: " + ~a);
        System.out.println("Logical NOT of true: " + !true);
        System.out.println("Unary plus: " + +a);
        System.out.println("Unary minus: " + -a);
        // Type casting
        System.out.println("\nType casting:");
        System.out.println("Int to double: " + (double)a);
    }
}

```

```

System.out.println("\nMultiplication, division, and modulus:");
// Multiplication and division have same precedence
System.out.println("a * b / c = " + (a * b / c));
// Parentheses change the order
System.out.println("a * (b / c) = " + (a * (b / c)));
// Modulus has same precedence as multiplication
System.out.println("a * b % c = " + (a * b % c));
// Addition and subtraction
System.out.println("\nAddition and subtraction:");
// Left to right evaluation
System.out.println("a + b - c = " + (a + b - c));
// precedence across rows
System.out.println("\n different precedence levels:");
// Prefix increment, then multiplication, then addition
System.out.println(++a * b + c = " + (++a * b + c));
// Multiplication before addition
System.out.println("a + b * c = " + (a + b * c));
// Parentheses change the order
System.out.println("(a + b) * c = " + ((a + b) * c));
    }
}

```

5 6

11 11

Unary operators:

Bitwise complement of 5: -7

Logical NOT of true: false

Unary plus: 6

Unary minus: -6

Type casting:

Int to double: 6.0

Multiplication, division, and modulus:

$a * (b / c) = 30$

$a * b \% c = 0$

Addition and subtraction:

$a + b - c = 15$

different precedence levels:

$++a * b + c = 79$

$a + b * c = 29$

$(a + b) * c = 36$

```
public class OperatorPrecedence {  
    public static void main(String[] args) {  
        int a = 5;  
        int b = 10;  
        int c = 2;  
        // Postfix and prefix operators  
        System.out.println(a++ + " " + a);  
        // Prefix increment  
        System.out.println(++b + " " + b);  
        // Unary operators  
        System.out.println("\nUnary operators:");  
        System.out.println("Bitwise complement of 5: " + ~a);  
        System.out.println("Logical NOT of true: " + !true);  
        System.out.println("Unary plus: " + +a);  
        System.out.println("Unary minus: " + -a);  
        // Type casting  
        System.out.println("\nType casting:");  
        System.out.println("Int to double: " + (double)a);  
    }  
}
```

5 6
11 11

Unary operators:

Bitwise complement of 5: -7

Logical NOT of true: false

Unary plus: 6

Unary minus: -6

Type casting:

Int to double: 6.0

Multiplication, division, and modulus:

$a * (b / c) = 30$

$a * b \% c = 0$

Addition and subtraction:

$a + b - c = 15$

different precedence levels:

$++a * b + c = 79$

$a + b * c = 29$

$(a + b) * c = 36$

```
// Multiplication, division, and modulus
System.out.println("\nMultiplication, division, and modulus:");
// Multiplication and division have same precedence
System.out.println("a * b / c = " + (a * b / c));
// Parentheses change the order
System.out.println("a * (b / c) = " + (a * (b / c)));
// Modulus has same precedence as multiplication
System.out.println("a * b % c = " + (a * b % c));
    // Addition and subtraction
    System.out.println("\nAddition and subtraction:");
    // Left to right evaluation
    System.out.println("a + b - c = " + (a + b - c));
    // Demonstrating precedence across rows
    System.out.println("\ndifferent precedence levels:");
    // Prefix increment, then multiplication, then addition
    System.out.println("++a * b + c = " + (++a * b + c));
    // Multiplication before addition
    System.out.println("a + b * c = " + (a + b * c));
    // Parentheses change the order
    System.out.println("(a + b) * c = " + ((a + b) * c));
}
}
```

Control statements

- To cause the flow of execution to advance and branch based on changes to the state of a program.
- Java's program control statements can be put into the following categories:
 - Selection
 - Iteration
 - Jump

Control statements

- Selection statements:
 - To choose different paths of execution based upon the outcome of an expression or the state of a variable.
- Iteration statements:
 - To enable program execution to repeat one or more
- Jump statements:
 - To allow your program to execute in a nonlinear fashion.

Java's Selection Statements

- To control the flow of your program's execution based upon conditions known only during run time.
- Java supports two selection statements:
 - if and switch.

Java's Selection Statements

- If statement
 - The if statement is Java's conditional branch statement.
 - Used to route program execution through two different paths.

```
if (condition)
    statement1;
else
    statement2;
```

- Here, each statement may be a single statement, or a compound statement enclosed in curly braces (that is, a block).
- The condition is any expression that returns a boolean value.
- The else clause is optional.
- The **if** statements are executed from the top down. As soon as one of the conditions controlling the **if** is **true**, the statement associated with that **if** is executed, and the rest of the ladder is bypassed.
- The final **else** acts as a default condition; that is, if all other conditional tests fail, then the last **else** statement is performed.
- If there is no final **else** and all other conditions are **false**, then no action will take place.

```
Nested If
if(condition)
    statement;
else if(condition)
    statement;
else if(condition)
    statement;
.
.
.
else
    statement;
```

Selection statement: If statement

```
class ConditionalExamples {
    public static void main(String args[]) {
        // First example
        int bytesAvailable = 10; // Example value
        int n = 5; // Example value
        if (bytesAvailable > 0) {
            processData();
            bytesAvailable -= n;
        } else {
            waitForMoreData();
            bytesAvailable = n;
        }
        // Second example
        boolean dataAvailable = true;
        if (dataAvailable)
            processData();
        else
            waitForMoreData();
    }
}
```

```
        // Third example
        int month = 4; // April
        String season;
        if (month == 12 || month == 1 || month == 2)
            season = "Winter";
        else if (month == 3 || month == 4 || month == 5)
            season = "Spring";
        else if (month == 6 || month == 7 || month == 8)
            season = "Summer";
        else if (month == 9 || month == 10 || month == 11)
            season = "Autumn";
        else
            season = "Bogus Month";
        System.out.println("April is in the " + season + ".");
    }
    private static void processData() {
        System.out.println("Processing data...");
    }
    private static void waitForMoreData() {
        System.out.println("Waiting for more data...");
    }
}
```

Selection statement: Switch

- The switch statement is Java's multiway branch statement.
- It provides an easy way to dispatch execution to different parts of your code based on the value of an expression.
- A better alternative than a large series of if-else-if statements.
- Expression must resolve to type byte, short, int, char, String or an enumeration.
- Duplicate case values are not allowed. The type of each value must be compatible with the type of expression.
- The expression is evaluated once.
- The break statement is used to exit the switch statement. Without break, the program continues to the next case.
- If no cases match, the default block is executed (if it exists).
- Switch can handle byte, short, char, int, String, enum primitive data types and Wrapper classes (Integer, Byte, Short, Character)

```
switch(expression){  
    case value1:  
        //statement  
        break;  
    case value2:  
        //statement  
        break;  
    .  
    .  
    .  
    case valueN:  
        //statement  
        break;  
    default:  
        //statement  
        break;  
}
```

```

class SwitchExample {
    public static void main(String args[]) {
        byte b = 1; short s = 100; char c = 'A'; int i = 10;
        switch (b) { // Example with byte
            case 1:
                System.out.println("Byte value is 1");
                break;
            default:
                System.out.println("Byte value is neither 1 nor 2");
                break;
        }

        switch (s) { // Example with short
            case 100:
                System.out.println("Short value is 100");
                break;
            default:
                System.out.println("Short value is neither 100 nor 200");
                break;
        }

        switch (c) { // Example with char
            case 'A':
                System.out.println("Char value is A");
                break;
            default:
                System.out.println("Char value is neither A nor B");
                break;
        }
    }
}

```

```

switch (i) {
    // Example with int
    case 10:
        System.out.println("Int value is 10");
        break;
    default:
        System.out.println("Int value is neither 10 nor 20");
        break;
}

// Example with String
String str = "hello";
switch (str) {
    case "hello":
        System.out.println("String is 'hello'");
        break;
    case "java":
        System.out.println("JamesGosling");
        break;
    default:
        System.out.println("neither 'hello' nor 'JamesGosling'");
        break;
}

```

```
// Example with enum
Day day = Day.MONDAY;
switch (day) {
    case MONDAY:
        System.out.println("It's Monday");
        break;
    case TUESDAY:
        System.out.println("It's Tuesday");
        break;
    default:
        System.out.println("It's neither Monday nor Tuesday");
        break;
}

// Enum type for the days of the week
enum Day {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,
    SATURDAY, SUNDAY
}
```

Byte value is 1
Short value is 100
Char value is A
Int value is 10
String is 'hello'
It's Monday

case 1: statement in the inner switch does not conflict with the case 1: statement in the outer switch.

```
class NestedSwitch {  
    public static void main(String[] args) {  
        int mode = 2, option = 1;  
  
        switch (mode) {  
            case 1 -> switch (option) {  
                case 1 -> System.out.println("Mode 1, Option 1");  
                case 2 -> System.out.println("Mode 1, Option 2");  
                case 3 -> System.out.println("Mode 1, Option 3");  
            };  
            case 2 -> switch (option) {  
                case 1 -> System.out.println("Mode 2, Option 1");  
                case 2 -> System.out.println("Mode 2, Option 2");  
                case 3 -> System.out.println("Mode 2, Option 3");  
            };  
        }  
    }  
}
```

```
class NestedSwitch {  
    public static void main(String[] args) {  
        int mode = 2, option = 1;  
  
        switch (mode) {  
            case 1:  
                switch (option) {  
                    case 1: System.out.println("Mode 1, Option 1"); break;  
                    case 2: System.out.println("Mode 1, Option 2"); break;  
                    case 3: System.out.println("Mode 1, Option 3"); break;  
                }  
                break;  
            case 2:  
                switch (option) {  
                    case 1: System.out.println("Mode 2, Option 1"); break;  
                    case 2: System.out.println("Mode 2, Option 2"); break;  
                    case 3: System.out.println("Mode 2, Option 3"); break;  
                }  
                break;  
        }  
    }  
}
```


Selection statement: Switch

```
import java.util.Scanner;

public class SimpleCalculator {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        double num1, num2, result;
        char operation;
        System.out.println("Simple Calculator");
        System.out.println("Enter first number:");
        num1 = scanner.nextDouble();
        System.out.println("Enter an operation (+, -, *, /):");
        operation = scanner.next().charAt(0);
        System.out.println("Enter second number:");
        num2 = scanner.nextDouble();
        switch (operation) {
            case '+':
                result = num1 + num2;
                System.out.printf("%.2f + %.2f = %.2f", num1, num2, result);
                break;
```

```
            case '-':
                result = num1 - num2;
                System.out.printf("%.2f - %.2f = %.2f", num1, num2, result);
                break;
            case '*':
                result = num1 * num2;
                System.out.printf("%.2f * %.2f = %.2f", num1, num2, result);
                break;
            case '/':
                if (num2 != 0) {
                    result = num1 / num2;
                    System.out.printf("%.2f / %.2f = %.2f", num1, num2, result);
                } else {
                    System.out.println("Error: Division by zero!");
                }
                break;
            default:
                System.out.println("Error: Invalid operation!");
        }
    }
}
```

Simple Calculator

Enter first number:

10

Enter an operation (+, -, *, /):

#

Enter second number:

2

Error: Invalid operation!

Simple Calculator

Enter first number:

10

Enter an operation (+, -, *, /):

*

Enter second number:

2

10.00 * 2.00 = 20.00

Selection statement: Switch

- Three important features of the switch statement to note:
- The switch differs from the if in that switch can only test for equality, whereas if can evaluate any type of Boolean expression. That is, the switch looks only for a match between the value of the expression and one of its case constants.
- No two case constants in the same switch can have identical values. Of course, a switch statement and an enclosing outer switch can have case constants in common.
- A switch statement is usually more efficient than a set of nested ifs in terms of execution speed and code clarity, particularly when there are many cases to evaluate

Selection statement: switch vs if

- To select among a large group of values, a **switch** statement will run much faster than the equivalent logic coded using a sequence of **if-elses**.
- The compiler can do this because it knows that the **case** constants are all the same type and simply must be compared for equality with the **switch** expression.
- The compiler has no such knowledge of a long list of **if** expressions.

Iteration Statements:

- Java's iteration statements are
 - for,
 - while, and
 - do-while.
- These statements are commonly call loops.
- A loop repeatedly executes the same set of instructions until a termination condition is met.

Iteration statement: While

- The while loop is Java's most fundamental loop statement. It repeats a statement or block while its controlling expression is true.

```
while(condition) {  
    // body of loop  
}
```

- The condition can be any Boolean expression.
- The body of the loop will be executed as long as the conditional expression is true.
- When condition becomes false, control passes to the next line of code immediately following the loop.
- The curly braces are unnecessary if only a single statement is being repeated.

```
class NoBody {  
    public static void main(String[] args) {  
        int i, j;  
  
        i = 100;  
        j = 200;  
  
        // Find midpoint between i and j  
        while (++i < --j); // no body in this loop  
  
        System.out.println("Midpoint is " + i);  
    }  
}
```

Midpoint is 150

```
class SimpleWhileLoop {  
    public static void main(String[] args) {  
        int count = 1; // Initialize the counter  
        while (count <= 5) {  
            System.out.println("Count is: " + count);  
            count++; // Increment the counter  
        }  
    }  
}
```

Count is: 1
Count is: 2
Count is: 3
Count is: 4
Count is: 5

Iteration statement: do-while

- Each iteration of the do-while loop first executes the body of the loop and then evaluates the conditional expression.
- If this expression is true, the loop will repeat. Otherwise, the loop terminates.
- Java's loops, condition must be a Boolean expression.

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

```
import java.util.Scanner;
// validate user input

class DoWhileLoopExample {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int number;
        do {
            System.out.print("Please enter a number between 1 and 10: ");
            number = scanner.nextInt();

            if (number < 1 || number > 10) {
                System.out.println("Invalid input. Try again.");
            }
        } while (number < 1 || number > 10);

        System.out.println("You entered: " + number);
    }
}
```

Please enter a number between 1 and 10: 5
You entered: 5

Please enter a number between 1 and 10: 11
Invalid input. Try again.

Iteration statement: While vs do While

- While loop
 - The conditional expression controlling a while loop is initially false, then the body of the loop will not be executed at all.
- Do while loop
 - Execute the body of a loop at least once, even if the conditional expression is false to begin with.
 - To test the termination expression at the end of the loop rather than at the beginning.
 - The do-while loop always executes its body at least once.

Iteration statement:for

- There are two forms of the for loop.
 - for loop
 - Requires manual management of the index and loop bounds.
 - Indexing required
 - Read and Write access

```
for(initialization; condition; increment){  
    // body of the loop  
}
```

- for each loop
 - iterates directly over elements in arrays or collections
 - No indexing required
 - Read-only access

```
for (Type element : Array or collection) {  
    // body of the loop  
}
```

Iteration statement: for

```
for(initialization; condition; increment){  
    // body of the loop  
}
```

- First, the initialization portion of the loop is executed only once when the loop starts.
- Next, Boolean expression condition is evaluated.
 - If this expression is true, then the body of the loop is executed.
 - If it is false, the loop terminates.
- Next, the iteration portion of the loop is executed.
- The loop then iterates, first evaluating the conditional expression, then executing the body of the loop, and then executing the iteration expression with each pass. This process repeats until the controlling expression is false.

Iteration statement:for

- There are two forms of the for loop.
 - for loop

```
String[] javaDevelopers = {"James", "Patrick", "Mike"};
for (int i = 0; i < javaDevelopers.length; i++) {
    System.out.println(javaDevelopers[i]);
}
```

- for each loop

```
String[] javaDevelopers = {"James", "Patrick", "Mike"};
for (String name : javaDevelopers) {
    System.out.println(name);
}
```

Basic for Loop

```
class BasicForLoop {  
    public static void main(String[] args) {  
        // Print squares of numbers from 1 to 5  
        for (int i = 1; i <= 5; i++) {  
            System.out.println("Square of " + i + " is " +  
                (i * i));  
        }  
    }  
}
```

Square of 1 is 1
Square of 2 is 4
Square of 3 is 9
Square of 4 is 16
Square of 5 is 25

For loop with variable declared inside:

```
class VariableInsideForLoop {  
    public static void main(String[] args) {  
        // Print Fibonacci sequence up to 100  
        int prev = 0;  
        System.out.print("Fibonacci sequence: ");  
        for (int current = 1; current <= 100; ) {  
            System.out.print(current + " ");  
            int next = prev + current;  
            prev = current;  
            current = next;  
        }  
    }  
}
```

Fibonacci sequence: 1 1 2 3 5 8 13 21 34 55 89

For loop with boolean condition:

```
class BooleanConditionForLoop {  
    public static void main(String[] args) {  
        int sum = 10;  
        boolean reachedTarget = false;  
        int num ;  
        for ( num=1; !reachedTarget; num++) {  
            sum += num;  
            if (sum > 100) {  
                reachedTarget = true;  
            }  
        }  
        System.out.println("Sum exceeded 100 after adding " + num+ " numbers");    }  
}
```

Sum exceeded 100 after adding 14 numbers

For loop with some parts empty:

```
class EmptyPartsForLoop {  
    public static void main(String[] args) {  
        int[] numbers = {1, 2, 3, 4, 5};  
        int index = 0;  
  
        for (; index < numbers.length;) {  
            System.out.println("Element at index " + index + ": " +  
numbers[index]);  
            index++;  
        }  
    }  
}
```

Element at index 0: 1
Element at index 1: 2
Element at index 2: 3
Element at index 3: 4
Element at index 4: 5

```

class EmptyPartsForLoop {
    public static void main(String[] args) {
        int[] numbers = {1, 2, 3, 4, 5};
        int index = 0;
        Boolean complete=false;
        for (; !complete;) {
            System.out.println("Element at index " + index + ": " + numbers[index]);
            if(index==4) complete=true;
            index++;
        }
    }
}

```

Element at index 0: 1
 Element at index 1: 2
 Element at index 2: 3
 Element at index 3: 4
 Element at index 4: 5

An infinite loop. This loop will run forever because there is no condition under which it will terminate.

```

class InfiniteForLoop {
    public static void main(String[] args) {
        int counter = 0;
        for (;;) {
            System.out.println("This is iteration " + (++counter));
            if (counter == 5) {
                System.out.println("Breaking out of the infinite loop");
                break;
            }
        }
    }
}

```

```

class InfiniteWhileLoop {
    public static void main(String[] args) {
        int counter = 0;
        while (true) {
            System.out.println("This is iteration " + (++counter));
            if (counter == 5) {
                System.out.println("Breaking out of the infinite loop");
                break;
            }
        }
    }
}

```

An infinite loop. This loop will run forever because there is no condition under which it will terminate.

```
for (;;) {  
    // code  
    if (/* condition */) {  
        break; // Exit loop based on a condition  
    }  
}
```

```
while (true) {  
    // code  
    if (/* condition */) {  
        break; // Exit loop based on a condition  
    }  
}
```

```
class InfiniteForLoop {  
    public static void main(String[] args) {  
        int counter = 0;  
        for (;;) {  
            System.out.println("This is iteration " + (++counter));  
            if (counter == 5) {  
                System.out.println("Breaking out of the infinite loop");  
                break;  
            }  
        }  
    }  
}
```

This is iteration 1
This is iteration 2
This is iteration 3
This is iteration 4
This is iteration 5
Breaking out of the infinite loop

```
class InfiniteWhileLoop {  
    public static void main(String[] args) {  
        int counter = 0;  
        while (true) {  
            System.out.println("This is iteration " + (++counter));  
            if (counter == 5) {  
                System.out.println("Breaking out of the infinite loop");  
                break;  
            }  
        }  
    }  
}
```

This is iteration 1
This is iteration 2
This is iteration 3
This is iteration 4
This is iteration 5
Breaking out of the infinite loop

For loop with multiple variables:

To allow two or more variables to control a for loop, Java permits you to include multiple statements in both the initialization and iteration portions of the for. Each statement is separated from the next by a comma.

```
class MultipleVariablesForLoop {  
    public static void main(String[] args) {  
        // Print a countdown with days and hours  
        for (int days = 3, hours = 0; days >= 0; days--, hours += 6) {  
            System.out.println("Time remaining: " + days + " days and " + hours + " hours");  
        }  
    }  
}
```

Time remaining: 3 days and 0 hours

Time remaining: 2 days and 6 hours

Time remaining: 1 days and 12 hours

Time remaining: 0 days and 18 hours

Iteration statement: For-Each version of the for loop

- A for-each style loop is designed to cycle through a collection of objects, such as an array, in strictly sequential fashion, from start to finish.

```
//The general form of the for-each  
for(type itr-var : collection)  
statement-block
```

- type specifies the type
- itr-var specifies the name of an iteration variable that will receive the elements from a collection, one at a time, from beginning to end.

Basic for-each loop:

```
// Basic for-each style loop
class ForEachExample {
    public static void main(String[] args) {
        String[] fruits = {"Apple", "Banana", "Cherry", "Date", "berry"};
        int totalLength = 0;
        // for-each style to display and sum the length of fruit names
        for (String fruit : fruits) {
            System.out.println("Fruit name: " + fruit);
            totalLength += fruit.length();
        }
        System.out.println("Total length of all fruit names: " + totalLength);
    }
}
```

```
Fruit name: Apple
Fruit name: Banana
Fruit name: Cherry
Fruit name: Date
Fruit name: berry
Total length of all fruit names: 26
```

For-each loop with break

```
class ForEachWithBreak {  
    public static void main(String[] args) {  
        double[] prices = {10.99, 5.49, 15.99, 20.00, 7.99, 30.50};  
        double budget = 40.00;  
        double totalSpent = 0;  
        // for-each to sum prices until budget is exceeded  
        for (double price : prices) {  
            System.out.println("Checking item priced at: $" + price);  
            if (totalSpent + price > budget) {  
                break; // Stop if adding this item would exceed the budget  
            }  
            totalSpent += price;  
        }  
        System.out.println("Total spent within budget: $" + totalSpent);  
    }  
}
```

```
Checking item priced at: $10.99  
Checking item priced at: $5.49  
Checking item priced at: $15.99  
Checking item priced at: $20.0  
Total spent within budget: $32.47
```

For-each loop with 2D

```
class ForEachWith2D {  
    public static void main(String[] args) {  
        String[][] schedule = {  
            {"Monday", "Math", "History"},  
            {"Tuesday", "Science", "English"},  
            {"Wednesday", "Art", "Music"}  
        };  
  
        // Use for-each to display the schedule  
        for (String[] day : schedule) {  
            for (String subject : day) {  
                System.out.print(subject + "\t");  
            }  
            System.out.println();  
        }  
    }  
}
```

Monday	Math	History
Tuesday	Science	English
Wednesday	Art	Music

Using type inference in for loops

```
class TypeInferenceInFor {  
    public static void main(String[] args) {  
        System.out.print("Powers of 2: ");  
        for (var i = 1; i <= 128; i *= 2) {  
            System.out.print(i + " ");  
        }  
        System.out.println();  
  
        var temperatures = new double[] {98.6, 100.4, 97.3, 99.1, 98.8};  
        System.out.print("Temperatures: ");  
        for (var temp : temperatures) {  
            System.out.printf("%.1f°F ", temp);  
        }  
        System.out.println();  
    }  
}
```

Powers of 2: 1 2 4 8 16 32 64 128

Temperatures: 98.6°F 100.4°F 97.3°F 99.1°F 98.8°F

Jump Statements

- Java supports three jump statements:
 - break,
 - continue, and
 - return.
- These statements transfer control to another part of your program.

Break

- Force immediate termination of a loop, bypassing the conditional expression and any remaining code in the body of the loop.
- When a break statement is encountered inside a loop, the loop is terminated and program control resumes at the next statement following the loop.

```
// Using break to exit a loop.
class BreakLoop {
    public static void main(String args[]) {
        for(int i=0; i<100; i++) {
            if(i == 10) break; // terminate loop if i is 10
            System.out.println("i: " + i);
        }
        System.out.println("Loop complete.");
    }
}
```

```
i: 0
i: 1
i: 2
i: 3
i: 4
i: 5
i: 6
i: 7
i: 8
i: 9
Loop complete..
```


Using break as labels

```
// Using break as a form of goto.
class Break {
    public static void main(String args[]) {
        boolean t = true;

        one: {
            two: {
                three: {
                    System.out.println("I am executable in three block.");
                    if(t) break three; // break out of three block
                    System.out.println("I won't execute");
                }
                System.out.println("I am executable at block two");
                if(t) break two; // break out of two block
            }
            System.out.println("This is at block one.");
        }
    }
}
```

```
I am executable in three block.
I am executable at block two
This is at block one.
```

```
// Using break to exit from nested loops
class BreakLoop4 {
    public static void main(String args[]) {
        outer: for(int i=0; i<10; i++) {
            System.out.print("outer loop " + i + ": ");
            for(int j=0; j<20; j++) {
                if(j == 10) break outer; // exit inner and outer loops
                System.out.print(j + " ");
            }
            System.out.println("I won't execute");
        }
        System.out.println("Outer Loops complete.");
    }
}
```

```
outer loop 0: 0 1 2 3 4 5 6 7 8 9 Outer Loops complete.
```

Using break as labels

```
// This program contains an error.
class BreakWithErr {
    public static void main(String args[]) {
        one: for(int i=0; i<10; i++) {
            System.out.print("Pass " + i + ": ");
        }

        for(int j=0; j<10; j++) {
            if(j == 10) break one; // error- one is not in this scope
            System.out.print(j + " ");
        }
    }
}
```

java: undefined label: one

Using continue

- In **while** and **do-while** loops, a **continue** statement causes control to be transferred directly to the conditional expression that controls the loop.
- In a **for** loop, control goes first to the iteration portion of the **for** statement and then to the conditional expression.
- For all three loops, any intermediate code is bypassed.

```
// Usage of continue.
class Continue {
    public static void main(String args[]) {
        for(int i=0; i<10; i++) {
            System.out.print(i + " ");
            if (i%2 == 0) continue;
            System.out.println("");
        }
    }
}
```

```
0 1
2 3
4 5
6 7
8 9
```

```
// Usage continue with a label.
class ContinueLabel {
    public static void main(String args[]) {
        outer: for (int i=0; i<10; i++) {
            for(int j=0; j<10; j++) {
                if(j > i) {
                    System.out.println();
                    continue outer;
                }
            }
            System.out.print(" " + (i + j));
        }
        System.out.println();    }
}
```

```
0
1 2
2 3 4
3 4 5 6
4 5 6 7 8
5 6 7 8 9 10
6 7 8 9 10 11 12
7 8 9 10 11 12 13 14
8 9 10 11 12 13 14 15 16
9 10 11 12 13 14 15 16 17 18
```

jump statement:Return

- The return statement is used to explicitly return from a method.
- That is, it causes program control to transfer back to the caller of the method. As such, it is categorized as a

```
// Demonstrate return.  
class Return {  
    public static void main(String args[]) {  
        boolean t = true;  
  
        System.out.println(" I will execute.");  
  
        if(t) return; // returning to caller  
  
        System.out.println("I won't execute.");  
    }  
}
```

I will execute.