**What is Java?**

 Java is a high-level, object-oriented programming language that was developed by Sun Microsystems (acquired by Oracle Corporation) in the mid-1990s. It is designed to be platform-independent, meaning that Java programs can run on any device that has a Java Virtual Machine (JVM). The JVM interprets and executes Java bytecode, which is generated by compiling Java source code.  
  
Key features of Java include:  
  
1. Write Once, Run Anywhere (WORA): Java programs are compiled into an intermediate form called bytecode, which can be executed on any device with a compatible JVM. This makes Java highly portable and platform-independent.  
  
2. Object-Oriented: Java follows the principles of object-oriented programming (OOP), which involves organizing code into classes and objects. This approach promotes code reusability, modularity, and maintainability.  
  
3. Automatic Memory Management: Java uses a garbage collector to automatically manage memory, freeing developers from the burden of manual memory management. This helps prevent memory leaks and makes Java programs more robust.  
  
4. Security: Java has built-in security features to protect against various security threats. The Java Runtime Environment (JRE) includes a security manager and various APIs to create secure applications.  
  
5. Rich Standard Library: Java comes with a comprehensive standard library that provides a wide range of functionalities for tasks such as networking, input/output, data structures, and more. This helps developers write code more efficiently without having to reinvent the wheel.  
  
6.Multithreading: Java supports multithreading, allowing developers to create concurrent and parallel applications. This is crucial for developing applications that can efficiently handle tasks simultaneously, enhancing performance and responsiveness.  
  
Java is widely used for developing a variety of applications, including web applications, mobile applications (Android is built using Java), enterprise software, scientific applications, and more. It remains a popular choice for developers due to its versatility, portability, and the extensive ecosystem of libraries and frameworks available for Java development.

**Daily Notes**

 Advantages of Java:  
  
1. Platform Independence:  
- Advantage: Java's "Write Once, Run Anywhere" (WORA) principle allows developers to write code on one platform and run it on any other with a compatible JVM. This portability is a significant advantage.  
  
2. Object-Oriented Programming:  
Advantage: Java's strong support for object-oriented programming (OOP) promotes modular and reusable code, making it easier to design, develop, and maintain large-scale applications.  
  
3. Automatic Memory Management:  
Advantage: Java's garbage collector automatically manages memory, reducing the risk of memory leaks. Developers don't have to explicitly free memory, leading to more robust and stable applications.  
  
4. Rich Standard Library:  
Advantage: Java provides a comprehensive standard library with a wide range of built-in functionalities. This minimizes the need for developers to write extensive low-level code, increasing productivity.  
  
Disadvantages of Java:  
  
1. Performance Overhead:  
Disadvantage: Java programs may have a performance overhead due to the abstraction provided by the JVM. While this gap has been closing with advancements in the JVM, certain applications may still require fine-tuned performance.  
  
2. Verbosity:  
-Disadvantage: Java code can be verbose compared to some modern programming languages. This verbosity may result in more lines of code to accomplish tasks, potentially making the codebase harder to read and maintain.  
  
3. Limited Low-Level Access:  
- Disadvantage: Java, being a high-level language, might not be the best choice for systems programming or applications that require low-level access to hardware. It may not be as suitable for scenarios where performance optimization is critical.  
  
4. Slower Startup Time:  
- Disadvantage: Java applications may have a slower startup time compared to languages that compile directly to machine code. This can be a drawback for certain types of applications, especially those that require quick responsivenes

**Java Components**

 Java Code(.java) – this is the java code you have written, and its saved in the .java file(s)  
Javac compiler – it compiles the java source code files (.java file) into bytecode so that it can be executed by JVM. The bytecode is saved in a class file by compiler.  
Bytecode - this is what is produced by the javac complier after compiling java code.  
Java Virtual Machine (JVM)  
Generally referred as JVM, it’s the primary function is to execute the bytecode produced by compiler.  
  
Each operating system has different JVM, however the output they produce after execution of bytecode is same across all operating systems – hence Java is referred regarded as platform independent.  
  
Program execution phases follows this general sequence write the program, then compile the program and run the program.  
  
1) Writing of the program is of course done by the java programmer like you and me.  
  
2) Compilation of program is done by javac compiler, javac is the primary java compiler included in java development kit (JDK). It takes java program as input and generates java bytecode as output.  
  
3) In third phase, JVM executes the bytecode generated by compiler. This is called program run phase.  
  
Java Development Kit (JDK)  
This is a complete java development kit that includes JRE (Java Runtime Environment), compilers and various tools like JavaDoc, Java debugger etc.  
  
You would need JDK installed on your computer in order create, compile and run Java program(s)  
  
Java Runtime Environment (JRE)  
JRE allows you to run java programs, it includes JVM, browser plugins and applets support. When you only need to run a java program on your computer, you would only need JRE.  
  
Note: It is important to note JRE only runs a compiled java program.  
  
You cannot compile java programs with JRE you will need JDK.

**Main Java features**

 Java is a versatile and widely used programming language that incorporates several features, making it suitable for various types of application development.  
  
1. Platform Independence (Write Once, Run Anywhere):  
- Java programs are compiled into bytecode, which can be executed on any device with a Java Virtual Machine (JVM). This platform independence is a key feature, allowing Java applications to run on diverse systems without modification.  
  
2. Object-Oriented Programming (OOP):  
- Java is designed around the principles of object-oriented programming, emphasizing concepts like classes, objects, encapsulation, inheritance, and polymorphism. This promotes code organization, reusability, and modularity.  
  
3. Automatic Memory Management (Garbage Collection):  
- Java incorporates a garbage collector that automatically manages memory, deallocating objects that are no longer in use. This feature helps prevent memory leaks and simplifies memory management for developers.  
  
4. Multithreading:  
- Java supports multithreading, allowing developers to write programs that can perform multiple tasks concurrently. This is essential for creating efficient and responsive applications, particularly in scenarios with parallel processing requirements.  
  
5. Rich Standard Library  
- Java comes with a robust and extensive standard library that provides pre-built packages and classes for various common tasks. This includes networking, input/output operations, data structures, utilities, and more, reducing the need for developers to write code from scratch.  
  
6. Security:  
- Java incorporates various security features, including a security manager and a set of APIs that facilitate the development of secure applications. Applets, which were a popular Java feature for web applications, ran in a restricted environment for enhanced security.  
  
7. Dynamic and Extensible:  
- Java supports dynamic loading of classes, allowing classes to be loaded on-demand during runtime. This dynamic and extensible nature enables the development of flexible and adaptable applications.  
  
8. High Performance:  
- While Java is not as low-level as some languages, modern JVMs provide sophisticated optimization techniques, resulting in competitive performance for a wide range of applications. Just-In-Time (JIT) compilation is one such optimization technique used by JVMs.

**Setting up the environment**

 Setting up the Java development environment involves a few key steps, including installing the Java Development Kit (JDK), configuring the environment variables, and selecting an Integrated Development Environment (IDE) if desired. Below are the general steps for setting up a Java development environment:  
  
### 1. \*\*Install the Java Development Kit (JDK):\*\*  
  
Visit the official Oracle website or an alternative distribution (such as OpenJDK) to download the latest version of the JDK. Follow the installation instructions provided for your operating system:  
  
- [Oracle JDK](https://www.oracle.com/java/technologies/javase-downloads.html)  
- [OpenJDK](https://adoptopenjdk.net/)  
  
### 2. \*\*Configure Environment Variables:\*\*  
  
After installing the JDK, you need to set up the following environment variables:  
  
\*\*On Windows:\*\*  
- Right-click on "This PC" or "Computer" and select "Properties."  
- Click on "Advanced system settings" on the left.  
- Click the "Environment Variables" button.  
- Under "System variables," find and edit the "Path" variable. Add the path to the "bin" directory of your JDK installation (e.g., `C:\Program Files\Java\jdk1.8.0\_181\bin`).  
  
\*\*On macOS and Linux:\*\*  
- Edit the `~/.bash\_profile`, `~/.bashrc`, or `~/.zshrc` file, depending on your shell.  
- Add the following line (replace `<path-to-jdk>` with the actual path to your JDK installation):  
```bash  
export PATH=$PATH:/path/to/jdk/bin  
```  
- Save the file and run `source ~/.bash\_profile` (or the corresponding file) to apply the changes.  
  
### 3. \*\*Verify Java Installation:\*\*  
  
Open a command prompt or terminal window and run the following commands to verify that Java is installed:  
  
```bash  
java -version  
```  
  
This command should display the installed Java version.  
  
```bash  
javac -version  
```  
  
This command should display the installed Java compiler version.  
  
### 4. \*\*Choose an Integrated Development Environment (IDE):\*\*  
  
While not strictly necessary, using an IDE can significantly enhance your development experience. Popular Java IDEs include:  
  
- [Eclipse](https://www.eclipse.org/)  
- [IntelliJ IDEA](https://www.jetbrains.com/idea/)  
- [NetBeans](https://netbeans.apache.org/)  
  
Download and install your preferred IDE, following the instructions provided on their respective websites.  
  
### 5. \*\*Create a Simple Java Program:\*\*  
  
Now that your environment is set up, create a simple Java program to test your installation. Use a text editor to write a Java source file (e.g., `HelloWorld.java`):  
  
```java  
public class HelloWorld {  
public static void main(String[] args) {  
System.out.println("Hello, World!");  
}  
}  
```  
  
Save the file and compile it using the command:  
  
```bash  
javac HelloWorld.java  
```  
  
Run the compiled program with:  
  
```bash  
java HelloWorld  
```  
  
If you see the "Hello, World!" output, your Java development environment is successfully set up.  
  
These steps provide a basic setup for Java development. Depending on your specific needs, you may explore additional tools, libraries, or build tools to enhance your development workflow.

**Daily Notes**

 Certainly! Let's delve into how the presence of computers and the internet has influenced and contributed to making Java a better programming language.  
  
### 1. \*\*Computers and Java:\*\*  
  
\*\*a. Portability:\*\*  
- Java's platform independence is a crucial advantage in the context of computers. The ability to write code once and run it on any device with a Java Virtual Machine (JVM) enhances portability. This is especially beneficial in diverse computing environments, where Java applications can seamlessly run on different operating systems and hardware architectures.  
  
\*\*b. Cross-Platform Development:\*\*  
- The ubiquity of computers with different operating systems (Windows, macOS, Linux) necessitates a programming language that can cater to a wide audience. Java's cross-platform capabilities allow developers to create applications that are not bound to a specific operating system, contributing to its popularity in desktop application development.  
  
\*\*c. Enterprise Applications:\*\*  
- Java's suitability for developing large-scale enterprise applications is amplified by the prevalence of computers in business environments. The platform independence, scalability, and robustness of Java make it an ideal choice for building server-side applications, middleware, and backend systems.  
  
### 2. \*\*Internet and Java:\*\*  
  
\*\*a. Web Development:\*\*  
- The internet has revolutionized the way applications are delivered and accessed. Java's role in web development, particularly on the server side, has been significant. Java-based technologies like Servlets and JavaServer Pages (JSP) have been instrumental in building dynamic and interactive web applications.  
  
\*\*b. Applets (Historical Significance):\*\*  
- In the early days of the internet, Java Applets played a key role in enhancing web interactivity. Applets allowed developers to embed Java programs within web pages, providing functionality beyond what HTML alone could offer. While Applets are less common today, their historical role influenced Java's position in web development.  
  
\*\*c. Java for Backend Services:\*\*  
- The internet's evolution has led to an increased demand for robust and scalable backend services. Java's role in server-side programming, supported by technologies like Java EE (Enterprise Edition) and Spring Framework, has made it a preferred choice for building backend services that power web applications.  
  
\*\*d. Networking and Distributed Computing:\*\*  
- The internet relies on networking, and Java's built-in support for networking facilitates the development of applications that communicate over the web. Java's Remote Method Invocation (RMI) and other technologies enable the creation of distributed systems, a crucial aspect of internet-based applications.  
  
### 3. \*\*Overall Impact:\*\*  
  
\*\*a. Standardization and Interoperability:\*\*  
- Java's adoption in internet-related development has contributed to standardization and interoperability. The language's widespread use has led to the creation of a vast ecosystem of libraries and frameworks that adhere to common conventions, fostering interoperability between different components of the software stack.  
  
\*\*b. Security:\*\*  
- Given the security challenges posed by internet-based applications, Java's focus on security features, such as the Java Security Manager, has become increasingly important. Java's design considerations for networked environments contribute to creating secure applications on the internet.  
  
\*\*c. Community Collaboration:\*\*  
- The internet facilitates collaboration and information sharing. Java's community thrives on online forums, collaborative development platforms, and open-source contributions. The internet has played a pivotal role in fostering a vibrant Java community that shares knowledge, addresses challenges, and contributes to the continuous improvement of the language.  
  
In summary, the widespread presence of computers and the internet has greatly influenced the development and evolution of Java. Java's features, such as platform independence, network support, and security measures, align well with the demands of modern computing and internet-based applications, making it a robust and versatile programming language for a wide range of use cases

**Writing First java Program**

 public class HelloWorld {  
public static void main(String[] args) {  
System.out.println("Hello, World!");  
}  
}

**Typical Structure of a Java program**

 Typical Structure of a Java program  
A typical structure of a Java program contains the following elements:  
  
Package declaration  
Import statements  
Comments  
Class definition  
Attributes  
Methods/Behaviours  
Package declaration  
  
A class in Java can be placed in different directories/packages based on the module they are used. For all the classes that belong to a single parent source directory, a path from source directory is considered as package declaration. Keyword package is used on package declaration statement  
  
Import statements  
  
There can be classes written in other folders/packages of our working java project and also there are many classes written by individuals, companies, etc which can be useful in our program. To use them in a class, we need to import the class that we intend to use. Many classes can be imported in a single program and hence multiple import statements can be written. Keyword import is used on import statement  
  
Comments  
  
The comments in Java can be used to provide information about the variable, method, class or any other statement. There are two ways to write comments in Java:  
  
Single line comment - used to comment single line of code. It uses double forward slash (//)  
e.g. // this is a single line comment  
Multiline comment -used to comment multiple lines of code  
e.g. /\*This is multiline  
comment \*/  
  
Class Definition  
  
A name should be given to a class in a java file. This name is used while creating an object of a class, in other classes/programs. Keyword class is used on Class Definition  
  
Main Method  
  
Execution of a Java application starts from the main method. In other words, it’s an entry point for the class or program that starts in Java Run-time.  
  
Methods/Behaviours  
  
A set of instructions which form a purposeful functionality that can be required to run multiple times during the execution of a program. To not repeat the same set of instructions when the same functionality is required, the instructions are enclosed in a method. A method’s behaviour can be exploited by passing variable values to a method.

**Daily Notes**

 Elements Present:  
1. Package Statement: `package com.cititraining.edu.firstjava;`  
2. Import Statement: `import System.io.\*;`  
3. Class Definition: `public class HelloWorld {`  
4. Main Method: `public static void main(String[] args) {`  
5. Method Definition (`printHelloWorld`): `public void printHelloWorld() {`  
6. Comments: Both multiline and single-line comments are present.  
7. Print Statement: `System.out.println("Hello World");`  
  
Elements Missing:  
1. The `System` class should be in lowercase: `import system.io.\*;` instead of `import System.io.\*;`  
2. The `main` method should be declared within the class. It is already present in the provided code.  
3. The closing brace for the class: `}` is present.  
4. The code snippet is missing the closing brace for the main method (`main`) and the class (`HelloWorld`). It should end with `}`.

**Daily Notes**

 public class Dog {  
  
// Void method to make the dog bark  
public void bark() {  
System.out.println("Woof! Woof!");  
}  
  
// Main method to demonstrate the Dog class  
public static void main(String[] args) {  
// Creating an instance of the Dog class  
Dog myDog = new Dog();  
  
// Calling the bark method  
myDog.bark();  
}  
}

**What is a variable?**

 A variable is a name given to a memory location. It is the basic unit of storage in a program.  
The value stored in a variable can be changed during program execution.  
A variable is only a name given to a memory location; all the operations done on the variable effects that memory location.  
In Java, all the variables must be declared before used

**Data types in Java**

 In Java, data types define the type of data that a variable can hold. Java has two categories of data types: primitive data types and reference data types.  
  
### Primitive Data Types:  
  
1. \*\*Integral Types:\*\*  
- `byte`: 8-bit signed integer.  
- `short`: 16-bit signed integer.  
- `int`: 32-bit signed integer.  
- `long`: 64-bit signed integer.  
  
2. \*\*Floating-Point Types:\*\*  
- `float`: 32-bit floating-point.  
- `double`: 64-bit floating-point.  
  
3. \*\*Character Type:\*\*  
- `char`: 16-bit Unicode character.  
  
4. \*\*Boolean Type:\*\*  
- `boolean`: Represents true or false values.  
  
### Reference Data Types:  
  
1. \*\*Objects:\*\*  
- Any class you create is a reference data type.  
- Examples include `String`, `Scanner`, `Random`, etc.  
  
2. \*\*Arrays:\*\*  
- A collection of similar types.  
- Example: `int[] numbers;`  
  
3. \*\*Enumerations (Enums):\*\*  
- A special data type that enables for a variable to be a set of predefined constants.  
- Example:  
```java  
enum Day {  
SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY  
}  
```  
  
4. \*\*Interfaces:\*\*  
- Reference data types representing a contract for classes to implement.  
  
5. \*\*Others:\*\*  
- There are other types like `BigInteger` and `BigDecimal` in the `java.math` package for arbitrary-precision integers and decimals, respectively.  
  
### Default Values:  
  
- Primitive data types have default values (e.g., `0` for numeric types, `false` for `boolean`).  
- Reference data types default to `null` (until an object is explicitly assigned).  
  
### Example:  
  
```java  
public class DataTypesExample {  
public static void main(String[] args) {  
// Primitive Data Types  
int age = 25;  
double height = 5.9;  
char grade = 'A';  
boolean isStudent = true;  
  
// Reference Data Types  
String name = "John";  
int[] numbers = {1, 2, 3};  
Day today = Day.MONDAY;  
}  
}  
```  
  
Remember that Java is a statically-typed language, meaning that you must declare the data type of a variable before using it.

**Daily Notes**

 In Java, casting refers to the process of converting a value from one data type to another. This is necessary when you want to assign a value of one data type to a variable of another data type. Casting is generally categorized into two types: widening (implicit) casting and narrowing (explicit) casting. These types of casting are associated with primitive data types.  
  
1. \*\*Widening (Implicit) Casting:\*\*  
- Widening casting happens automatically when you are trying to assign a value of a smaller data type to a variable of a larger data type.  
- Java performs widening casting implicitly because there is no loss of data when moving from a smaller data type to a larger one.  
- For example:  
```java  
int intValue = 10;  
double doubleValue = intValue; // Widening casting from int to double  
```  
  
2. \*\*Narrowing (Explicit) Casting:\*\*  
- Narrowing casting, on the other hand, is performed explicitly, and it requires the use of parentheses and a cast operator.  
- This type of casting is necessary when you are trying to assign a value of a larger data type to a variable of a smaller data type, as this involves the potential loss of data.  
- For example:  
```java  
double doubleValue = 10.5;  
int intValue = (int) doubleValue; // Narrowing casting from double to int  
```  
  
It's important to note that when narrowing casting is performed, there is a risk of losing precision or information. For instance, if you cast a `double` to an `int`, the fractional part will be truncated.  
  
Example with potential loss of precision:  
```java  
double doubleValue = 10.8;  
int intValue = (int) doubleValue; // intValue will be 10 (fractional part is truncated)  
```  
  
It's always a good practice to be cautious when using narrowing casting and to ensure that potential loss of information is acceptable for the specific application or use case.

**Operators in Java**

 In Java, operators are special symbols that perform operations on variables and values. They are the building blocks of expressions and are used to manipulate data. Here are some of the common operators in Java:  
  
1. \*\*Arithmetic Operators:\*\*  
- `+` (Addition)  
- `-` (Subtraction)  
- `\*` (Multiplication)  
- `/` (Division)  
- `%` (Modulus, gives the remainder of a division)  
  
```java  
int a = 10;  
int b = 20;  
int sum = a + b; // sum = 30  
int difference = b - a; // difference = 10  
int product = a \* b; // product = 200  
int quotient = b / a; // quotient = 2  
int remainder = b % a; // remainder = 0  
```  
  
2. \*\*Relational Operators:\*\*  
- `==` (Equal to)  
- `!=` (Not equal to)  
- `>` (Greater than)  
- `<` (Less than)  
- `>=` (Greater than or equal to)  
- `<=` (Less than or equal to)  
  
```java  
int x = 5;  
int y = 10;  
boolean isEqual = (x == y); // false  
boolean isNotEqual = (x != y); // true  
boolean isGreaterThan = (x > y); // false  
boolean isLessThan = (x < y); // true  
```  
  
3. \*\*Logical Operators:\*\*  
- `&&` (Logical AND)  
- `||` (Logical OR)  
- `!` (Logical NOT)  
  
```java  
boolean condition1 = true;  
boolean condition2 = false;  
  
boolean result1 = condition1 && condition2; // false  
boolean result2 = condition1 || condition2; // true  
boolean result3 = !condition1; // false  
```  
  
4. \*\*Assignment Operators:\*\*  
- `=` (Assignment)  
- `+=` (Addition assignment)  
- `-=` (Subtraction assignment)  
- `\*=` (Multiplication assignment)  
- `/=` (Division assignment)  
- `%=` (Modulus assignment)  
  
```java  
int num = 5;  
num += 3; // num is now 8 (equivalent to num = num + 3)  
num -= 2; // num is now 6 (equivalent to num = num - 2)  
```  
  
5. \*\*Increment and Decrement Operators:\*\*  
- `++` (Increment by 1)  
- `--` (Decrement by 1)  
  
```java  
int count = 10;  
count++; // count is now 11  
count--; // count is now 10  
```  
  
6. \*\*Conditional (Ternary) Operator:\*\*  
- `? :` (Conditional operator)  
  
```java  
int a = 5;  
int b = 10;  
int max = (a > b) ? a : b; // max is 10  
```  
  
These are some of the fundamental operators in Java. They are essential for performing various operations and comparisons in your programs.

**Java Classes and Objects**

 Everything in Java is associated with classes and objects, along with its attributes and methods. For example: in real life, a car is an object. The car has attributes, such as weight and color, and methods, such as drive and brake.  
  
A Class is like an object constructor, or a "blueprint” or a factory for creating objects. This means without a class no object can be created.  
  
What is a class in Java?  
Class − A class can be defined as a template/blueprint that describes the behaviour/state that the object of its type support.  
  
Creating a class  
  
To create a class, use the keyword class  
  
A class is made up of:  
  
Constructor  
  
A constructor is a method called when creating an object from a class e.g public Person()in the example. If we do not explicitly write a constructor for a class, the Java compiler builds a default constructor for that class. Some constructors may accept parameter and some may not.  
  
Each time a new object is created, at least one constructor will be invoked. The main rule of constructors is that they should have the same name as the class. A class can have more than one constructor.  
  
Methods  
  
A class can have any number of methods to access the value of various kinds of methods. In the example given below run, walk and sleep are methods  
  
Variables/Properties  
  
Usually these are attributes the describe objects that a created from that class.  
What is an Object in Java?  
Object is an instance of a class, i.e. they are created from classes. Objects have states/attributes and behaviours. Example: A dog has states - colour, name, breed as well as behaviours – wagging the tail, barking, eating.  
  
Creating an object  
  
In Java, an object is created from a class. We have already created the class named Person, so now we can use this to create objects.  
  
To create an object of Person, specify the class name, followed by the object name, and use the keyword new:

**What is OOP?**

 OOP stands for Object-Oriented Programming, and Java is an object-oriented programming language. In object-oriented programming, the key concept is to organize code into objects, which are instances of classes. Objects can have attributes (fields or properties) and behaviors (methods or functions). The four main principles of OOP are encapsulation, inheritance, polymorphism, and abstraction. Let's briefly discuss these concepts in the context of Java:  
  
1. \*\*Classes and Objects:\*\*  
- \*\*Class:\*\* A class is a blueprint or template for creating objects. It defines the properties and behaviors that objects created from the class will have.  
- \*\*Object:\*\* An object is an instance of a class. It represents a real-world entity and has both state (attributes) and behavior (methods).  
  
Example in Java:  
```java  
// Defining a simple class  
public class Car {  
// Fields  
String brand;  
int year;  
  
// Methods  
void start() {  
System.out.println("Car is starting...");  
}  
}  
  
// Creating objects of the Car class  
Car myCar = new Car();  
myCar.brand = "Toyota";  
myCar.year = 2022;  
myCar.start();  
```  
  
2. \*\*Encapsulation:\*\*  
- Encapsulation is the concept of bundling the data (attributes) and methods that operate on the data into a single unit, i.e., a class. It helps in hiding the internal details of an object and protecting its state.  
  
3. \*\*Inheritance:\*\*  
- Inheritance allows a class (subclass or derived class) to inherit the properties and behaviors of another class (superclass or base class). This promotes code reusability and establishes a relationship between classes.  
  
Example in Java:  
```java  
// Inheritance example  
public class SportsCar extends Car {  
// Additional fields and methods specific to SportsCar  
void turbo() {  
System.out.println("Turbo mode activated!");  
}  
}  
  
// Creating an object of the SportsCar class  
SportsCar mySportsCar = new SportsCar();  
mySportsCar.brand = "Ferrari";  
mySportsCar.year = 2023;  
mySportsCar.start();  
mySportsCar.turbo();  
```  
  
4. \*\*Polymorphism:\*\*  
- Polymorphism allows objects of different classes to be treated as objects of a common superclass. It provides a way to perform a single action in different ways.  
  
Example in Java:  
```java  
// Polymorphism example  
public interface Engine {  
void start();  
}  
  
public class ElectricEngine implements Engine {  
public void start() {  
System.out.println("Electric engine starting...");  
}  
}  
  
public class GasEngine implements Engine {  
public void start() {  
System.out.println("Gas engine starting...");  
}  
}  
  
// Using polymorphism  
Engine myEngine = new ElectricEngine();  
myEngine.start();  
```  
  
5. \*\*Abstraction:\*\*  
- Abstraction involves simplifying complex systems by modeling classes based on the essential properties and behaviors they share. Abstract classes and interfaces are used to achieve abstraction.  
  
Example in Java:  
```java  
// Abstract class example  
public abstract class Shape {  
// Abstract method (no implementation)  
public abstract double area();  
}  
  
public class Circle extends Shape {  
// Implementation of the abstract method  
private double radius;  
  
public Circle(double radius) {  
this.radius = radius;  
}  
  
public double area() {  
return Math.PI \* radius \* radius;  
}  
}  
  
// Using abstraction  
Shape myCircle = new Circle(5.0);  
System.out.println("Area of the circle: " + myCircle.area());  
```  
  
In summary, Object-Oriented Programming in Java is based on the principles of encapsulation, inheritance, polymorphism, and abstraction, allowing for the creation of modular and reusable code. These concepts help in designing software systems that are scalable, maintainable, and easier to understand.

**Daily Notes**

 Object-oriented programming (OOP) is popular for several reasons, but it's important to note that the popularity of programming paradigms can vary depending on the context, application domain, and developer preferences. Here are some reasons why OOP is often favored over declarative and procedural programming:  
  
1. \*\*Modeling Real-World Entities:\*\*  
- OOP provides a natural way to model and represent real-world entities and their interactions. Objects in OOP map closely to entities in the problem domain, making it easier to understand and design systems.  
  
2. \*\*Encapsulation:\*\*  
- The encapsulation feature of OOP helps in organizing code by bundling data (attributes) and the methods that operate on that data into a single unit (class). This makes it easier to manage complexity and hide the internal details of an object.  
  
3. \*\*Code Reusability:\*\*  
- Inheritance in OOP allows the creation of new classes that are built upon existing classes. This promotes code reuse, as common functionalities can be inherited from a base class, reducing the need to duplicate code.  
  
4. \*\*Polymorphism:\*\*  
- Polymorphism allows objects to be treated as instances of their parent class, enabling flexibility and extensibility in the code. This makes it easier to adapt and modify systems without affecting existing code.  
  
5. \*\*Abstraction:\*\*  
- Abstraction in OOP involves creating abstract classes and interfaces that define the essential properties and behaviors of objects. This helps in simplifying complex systems and focusing on the key aspects of the problem domain.  
  
6. \*\*Modularity:\*\*  
- OOP promotes modularity by breaking down a system into smaller, manageable units (objects). This makes it easier to understand, maintain, and update code, as changes can be localized to specific objects or classes.  
  
7. \*\*Support for Large-Scale Systems:\*\*  
- OOP is well-suited for building large-scale, complex systems. The principles of OOP, such as encapsulation and abstraction, contribute to the scalability and maintainability of the codebase.  
  
8. \*\*Community and Industry Adoption:\*\*  
- OOP has been widely adopted in the software development industry, and many popular programming languages, including Java, C++, and Python, are object-oriented. This widespread adoption has led to a large community of developers with expertise in OOP, creating a positive feedback loop.  
  
9. \*\*Object-Oriented Analysis and Design (OOAD):\*\*  
- The use of OOAD methodologies, such as Unified Modeling Language (UML), has become a standard practice in software engineering. These methodologies provide effective tools for designing and documenting complex systems using object-oriented principles.  
  
While OOP is popular, it's essential to recognize that different programming paradigms, including declarative and procedural, have their own strengths and are better suited for certain types of problems. The choice of programming paradigm often depends on the specific requirements of a project and the preferences and expertise of the development team.

**Features of OOP – Core features**

 Abstraction is a process of hiding the implementation details from the user, only the functionality will be provided to the user. In other words, the user will have the information on what the object does instead of how it does it. In Java, abstraction is achieved using Abstract classes and interfaces.  
  
Encapsulation in Java is a mechanism of wrapping the data (variables) and code acting on the data (methods) together as a single unit. In encapsulation, the variables of a class will be hidden from other classes, and can be accessed only through the methods of their current class. Therefore, it is also known as data hiding.  
  
To achieve encapsulation in Java −  
  
Declare the variables of a class as private.  
Provide public setter and getter methods to modify and view the variables values.  
Inheritance can be defined as the process where one class acquires the properties (methods and fields) of another. With the use of inheritance, the information is made manageable in a hierarchical order.  
  
The class which inherits the properties of other is known as subclass (derived class, child class) and the class whose properties are inherited is known as superclass (base class, parent class).  
  
extends is the keyword used to inherit the properties of a class. In the example below class Sub inherits some of the properties from class Super.  
  
class Super {  
  
.....  
  
.....  
  
}  
  
class Sub extends Super {  
  
.....  
  
.....  
  
}  
  
Polymorphism means to process objects differently based on their data type. In other words, it means, one method with multiple implementations, for a certain class of action. And which implementation to be used is decided at runtime depending upon the situation (i.e., data type of the object)  
  
This can be implemented by designing a generic interface, which provides generic methods for a certain class of action and there can be multiple classes, which provides the implementation of these generic methods.  
  
polymorphism can be implemented in two ways:  
  
Overloading in simple words means more than one method having the same method name that behaves differently based on the arguments passed while calling the method. This called static because, which method to be invoked is decided at the time of compilation.  
Overriding means a derived class is implementing a method of its super class. The call to overriden method is resolved at runtime, thus called runtime polymorphism

**Features of OOP – Other features**

 Coupling  
Coupling refers to the knowledge or information or dependency of another class. It arises when classes are aware of each other. If a class has the details information of another class, there is strong coupling. In Java, we use private, protected, and public modifiers to display the visibility level of a class, method, and field. You can use interfaces for the weaker coupling because there is no concrete implementation.  
  
Cohesion  
Cohesion refers to the level of a component which performs a single well-defined task. A single well-defined task is done by a highly cohesive method. The weakly cohesive method will split the task into separate parts. The java.io package is a highly cohesive package because it has I/O related classes and interface. However, the java.util package is a weakly cohesive package because it has unrelated classes and interfaces.  
  
Association  
Association represents the relationship between the objects. Here, one object can be associated with one object or many objects. There can be four types of association between the objects:  
  
One to One  
  
One to Many  
  
Many to One, and  
  
Many to Many  
  
Let's understand the relationship with real-time examples.  
  
For examples:  
  
One country can have one president - one to one,  
president can have many cabinet ministers - one to many.  
Many cabinet ministers can have one prime minister - many to one  
Many cabinet ministers can have many departments - many to many.  
Association can be unidirectional or bidirectional.  
  
Aggregation  
Aggregation is a narrower kind of association. It occurs when there’s a one-way (HAS-A) relationship between the two classes you associate through their objects. For example, every Passenger has a Car but a Car doesn’t necessarily have a Passenger. When you declare the Passenger class, you can create a field of the Car type that shows which car the passenger belongs to. Then, when you instantiate a new Passenger object, you can access the data stored in the related Car as well.  
  
Aggregation in Java:  
  
One-directional association.  
Represents a HAS-A relationship between two classes.  
Only one class is dependent on the other.  
Composition  
Composition is a stricter form of aggregation. It occurs when the two classes you associate are mutually dependent on each other and can’t exist without each other. For example, take a Car and an Engine class. A Car cannot run without an Engine, while an Engine also can’t function without being built into a Car. This kind of relationship between objects is also called a PART-OF relationship.  
  
Composition in Java:  
  
A restricted form of aggregation  
Represents a PART-OF relationship between two classes  
Both classes are dependent on each other  
If one class ceases to exist, the other can’t survive alone

**Introduction to Decision control**

 Decision control is a fundamental concept in computer programming and refers to the ability of a program to make decisions and execute different sets of instructions based on certain conditions. This is essential for creating programs that can respond dynamically to different scenarios and user inputs. Decision control structures allow the flow of a program to be altered, directing it along different paths depending on whether specific conditions are true or false.  
  
There are two main types of decision control structures:  
  
1. \*\*Conditional Statements:\*\*  
- \*\*if statement:\*\* This is a basic conditional statement that allows a program to execute a certain block of code if a specified condition is true. If the condition is false, the associated block of code is skipped.  
```python  
if condition:  
# code to be executed if the condition is true  
```  
  
- \*\*if-else statement:\*\* This extends the basic if statement by providing an alternative block of code to be executed if the condition is false.  
```python  
if condition:  
# code to be executed if the condition is true  
else:  
# code to be executed if the condition is false  
```  
  
- \*\*if-elif-else statement:\*\* This allows for multiple conditions to be checked in sequence. The first true condition's block of code is executed, and if none are true, the else block is executed.  
```python  
if condition1:  
# code to be executed if condition1 is true  
elif condition2:  
# code to be executed if condition2 is true  
else:  
# code to be executed if none of the conditions are true  
```  
  
2. \*\*Looping Statements:\*\*  
- \*\*while loop:\*\* This loop continues to execute a block of code as long as a specified condition is true.  
```python  
while condition:  
# code to be executed while the condition is true  
```  
  
- \*\*for loop:\*\* This loop iterates over a sequence (such as a list or range) and executes a block of code for each item in the sequence.  
```python  
for variable in sequence:  
# code to be executed for each item in the sequence  
```  
  
Decision control is crucial for creating interactive and dynamic programs. It enables developers to write code that can adapt and respond to different situations, making programs more flexible and capable of handling a variety of scenarios.

**If Statement, IF ELSE statement, Nested If Statement**

 In Java, the `if` statement, `if-else` statement, and nested `if` statement are used for decision control. Here's an introduction to each:  
  
### 1. \*\*if Statement:\*\*  
The `if` statement is used to execute a block of code only if a specified condition is true. The basic syntax is as follows:  
  
```java  
if (condition) {  
// code to be executed if the condition is true  
}  
```  
  
Example:  
```java  
int x = 10;  
  
if (x > 5) {  
System.out.println("x is greater than 5");  
}  
```  
  
### 2. \*\*if-else Statement:\*\*  
The `if-else` statement allows you to provide an alternative block of code to be executed if the condition in the `if` statement is false. The basic syntax is as follows:  
  
```java  
if (condition) {  
// code to be executed if the condition is true  
} else {  
// code to be executed if the condition is false  
}  
```  
  
Example:  
```java  
int y = 3;  
  
if (y > 5) {  
System.out.println("y is greater than 5");  
} else {  
System.out.println("y is not greater than 5");  
}  
```  
  
### 3. \*\*Nested if Statement:\*\*  
A nested `if` statement is an `if` statement inside another `if` or `else` block. This allows for more complex decision-making scenarios. The basic syntax is as follows:  
  
```java  
if (outerCondition) {  
// outer code  
  
if (innerCondition) {  
// code to be executed if both outer and inner conditions are true  
}  
}  
```  
  
Example:  
```java  
int a = 5;  
int b = 7;  
  
if (a > 0) {  
System.out.println("a is positive");  
  
if (b > 0) {  
System.out.println("b is also positive");  
}  
}  
```  
  
In this example, the inner `if` statement is nested within the outer `if` statement. The inner block will only be executed if both the outer and inner conditions are true.  
  
These decision control structures in Java allow you to create flexible and responsive programs by directing the flow of execution based on specified conditions.

**Daily Notes**

 The `switch` statement and the ternary conditional operator (`? :`) are both decision-making constructs in Java that allow you to make choices in your code based on certain conditions.  
  
### 1. \*\*Switch Statement:\*\*  
The `switch` statement is used when you have a single expression that you want to compare against multiple possible values. It provides an alternative to using a series of `if-else if` statements. Here's the basic syntax:  
  
```java  
switch (expression) {  
case value1:  
// code to be executed if expression equals value1  
break;  
case value2:  
// code to be executed if expression equals value2  
break;  
// ... additional cases  
default:  
// code to be executed if none of the cases match  
}  
```  
  
Example:  
  
```java  
int dayOfWeek = 3;  
  
switch (dayOfWeek) {  
case 1:  
System.out.println("Monday");  
break;  
case 2:  
System.out.println("Tuesday");  
break;  
// ... additional cases  
default:  
System.out.println("Invalid day");  
}  
```  
  
### 2. \*\*Ternary Conditional Operator (`? :`):\*\*  
The ternary operator is a shorthand way of writing a simple `if-else` statement. It's a concise way to express a conditional check and assign values based on the result. The basic syntax is:  
  
```java  
variable = (condition) ? expression\_if\_true : expression\_if\_false;  
```  
  
Example:  
  
```java  
int x = 10;  
int y;  
  
// If x is greater than 5, assign y the value 1, otherwise assign y the value 0  
y = (x > 5) ? 1 : 0;  
```  
  
The `? :` operator evaluates the condition (`x > 5`). If the condition is true, it returns the value after the `?` (1 in this case), otherwise, it returns the value after the `:` (0 in this case).  
  
### Comparison:  
- The `switch` statement is suitable when you have multiple cases to check against a single value.  
- The ternary conditional operator is suitable for simple, one-line assignments based on a condition.  
  
In general, choose the construct that best fits the complexity and readability of your code. The `switch` statement is often more readable when you have many cases, while the ternary operator is more concise for simple conditions.