1. **GENESIS OF JAVA**

The genesis of Java can be traced back to the early 1990s. It was conceived by James Gosling, Patrick Naughton, Chris Warth, Ed Frank, and Mike Sheridan at Sun Microsystems, which was later acquired by Oracle Corporation. The project initially aimed to develop software for consumer electronics like set-top boxes. However, with the advent of the internet and the World Wide Web, Java's focus shifted towards developing applications for the internet.

**Oak Language: Initially**, Java was named "Oak" after the tree outside Gosling's office. The team developed a new programming language with a syntax similar to C/C++, but with modifications to make it more suitable for embedded systems.

**Internet Focus:** As the World Wide Web gained popularity, the team recognized the potential for Oak in creating interactive and dynamic web content. They rebranded Oak as "Java" in 1995 and showcased its capabilities as a platform-independent language suitable for web development.

**Write Once, Run Anywhere (WORA):** One of the revolutionary features of Java was its ability to compile code into bytecode, which could run on any device with a Java Virtual Machine (JVM). This concept of platform independence was encapsulated in the slogan "Write Once, Run Anywhere."

**Introduction of Java Applets:** Java applets were small programs that could be embedded within web pages to provide interactive content. They enabled developers to create dynamic web experiences, which was a significant advancement at the time.

**Official Release:** Java 1.0 was officially released by Sun Microsystems in 1996. It included core libraries, development tools, and the Java Virtual Machine necessary to run Java applications.

**Growth and Adoption:** Java quickly gained popularity among developers due to its platform independence, robustness, and security features. It became the language of choice for enterprise software development, web applications, and mobile applications.

**Expansion of the Java Ecosystem:** Over the years, the Java ecosystem expanded with the introduction of frameworks (such as Spring and Hibernate), application servers (like Apache Tomcat and JBoss), and development tools (including Eclipse and IntelliJ IDEA). These additions further solidified Java's position as a leading programming language.

Today, Java remains one of the most widely used programming languages globally, powering a vast array of applications across various domains, including enterprise software, web development, mobile development (with Android), and big data processing. Despite the emergence of new languages and technologies, Java continues to evolve, with regular updates and enhancements to meet the changing needs of the software development industry.

A timeline of a computer program

Description automatically generated with medium confidence

**Fig 1: Genesis of JAVA**

1. **BYTE CODE**

Byte Code can be defined as an intermediate code generated by the compiler after the compilation of source code(JAVA Program). This intermediate code makes Java a platform-independent language. Compiler converts the source code or the Java program into the Byte Code(or machine code), and secondly, the Interpreter executes the byte code on the system. The Interpreter can also be called JVM(Java Virtual Machine). The byte code is the common piece between the compiler(which creates it) and the Interpreter (which runs it).

Whenever we write any program, it is not written in machine code. We write it in a high-level language like JAVA, C++, Python, etc. But the computer understands only the machine code. So when we execute our program, it is first converted into machine code or Byte code by the compiler and then executed by the Interpreter.

This intermediate code or the byte can run on any platform making, JAVA a platform-independent language.

**Bytecode and Internet**

**Platform Independence:** Java bytecode is the intermediate representation of Java source code compiled by the Java compiler. Instead of compiling directly to machine code for a specific hardware platform, Java code is compiled into bytecode that can run on any device with a Java Virtual Machine (JVM). This feature enables "Write Once, Run Anywhere" (WORA), making Java applications platform-independent. This was particularly advantageous in the early days of the internet when there was a diverse range of devices and operating systems accessing web content.

**Java Applets:** Java applets were a significant early use case for Java bytecode on the internet. Applets were small Java programs that could be embedded within web pages to provide interactive content. When a user visited a webpage containing a Java applet, the bytecode for the applet was downloaded to the user's browser and executed by the JVM running within the browser. This allowed developers to create dynamic and interactive web content, which was revolutionary at the time.

**Client-Server Communication:** Java bytecode facilitated client-server communication over the internet. Java-based client applications could communicate with server-side applications using various network protocols such as HTTP, TCP/IP, and RMI (Remote Method Invocation). The platform independence of Java bytecode meant that client applications could run on different devices and operating systems while communicating with server-side components seamlessly.

**Servlets and JSP:** As web development evolved, Java bytecode became integral to server-side programming as well. Java Servlets and JavaServer Pages (JSP) emerged as popular technologies for building dynamic web applications. Servlets are Java classes that extend the functionality of web servers to handle client requests, while JSP allows developers to embed Java code directly into HTML pages. Servlets and JSP are compiled into bytecode and executed on the server-side within a JVM, enabling the dynamic generation of web content.

**Java Web Frameworks:** Numerous Java web frameworks have been developed to streamline web application development. These frameworks, such as Spring MVC, Struts, and JavaServer Faces (JSF), abstract away low-level details and provide libraries and tools to facilitate the development of web applications. Under the hood, these frameworks leverage Java bytecode to achieve platform independence and enable robust, scalable web solutions.

Overall, Java bytecode has been instrumental in shaping the internet landscape by enabling platform-independent, secure, and scalable applications across a wide range of devices and environments. While the prominence of Java applets has diminished over time, Java bytecode continues to play a crucial role in modern web development, powering server-side applications, web services, and enterprise-grade solutions.

A screenshot of a computer code

Description automatically generated

1. **BUZZWORDS IN JAVA**
2. **Simple:**

Java is designed as a simple programming language. When Java was developed, the Java team wanted it to be simple for the professional programmer to learn and use effectively. This is because it has to work on various electronic devices or on small machines where less memory is available. At that time, the size of basic interpreter and class support is only 40 KB.

First of all, several difficult, clumsy, and confusing features of other programming languages such as C and C++ had been omitted in Java.

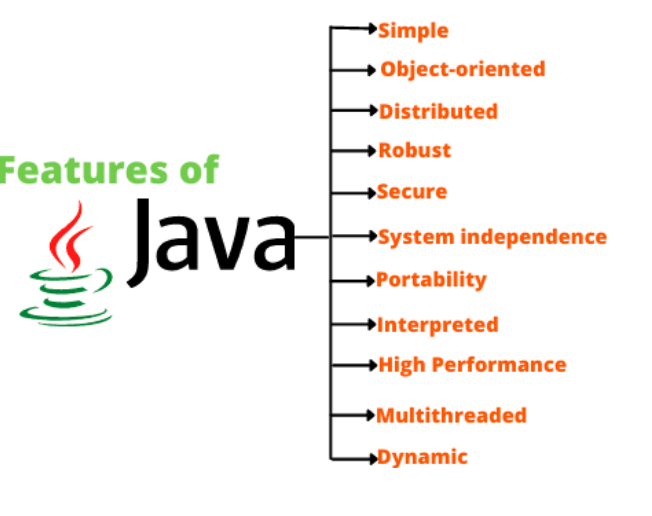
For example, the feature of pointers that is very difficult for both learners and programmers. Therefore, this feature had been completely eliminated from Java. Second, the JavaSoft team maintained the same syntax of C and C++ in Java so that a programmer who knows C or C++ will find Java already familiar and require very little effort in learning.

**Why Pointer feature is eliminated from Java?**

1. The concept of Pointer leads to confusion for a programmer.

2. Pointer may break a program easily. For example, when we add two pointers, the program can crash immediately.

3. The use of Pointer feature can break security because harmful programs like virus and other hacking programs can be developed using pointer.



**Fig 2: JAVA buzzwords**

1. **Object-oriented:**

Java is a purely object-oriented programming language (OOP). This means that Java program is developed by using classes and objects. In other words, to write a program in java, we need at least a class or an object. An object is anything that exists in the real world. For example, every human being, a book, a tree, pen, pencil, and so on. Every object has properties and shows certain behavior (action). Let us understand this statement with the help of an example.

**3. Distributed:**

Java is designed to support the distributed environment of the Internet because it handles TCP/IP and UDP protocols. Using Java, we can make a program to get information and can distribute it on various computers on a network. Java also supports Remote Method Invocation (RMI) feature that enable a program to distribute methods across a network.

**4. Robust:**

The meaning of Robust is “strong”. This means that Java programs are strong because they do not crash easily like C or C++ programs. There are mainly three reasons to be robust. Firstly, Java has an excellent inbuilt feature named exception handling. An exception is an abnormal condition or error that occurs at run time. If an exception occurs in a java program, the program terminates abruptly and rises to problems like loss of data. To overcome such types of problems, we can use exception handling mechanism. This means that even though an exception occurs in a program, no harm (i.e. loss of data) will happen.

Second reason is that Java has a robust memory management feature. Most of the C and C++ programs crash in the middle because of not allocating sufficient space in the memory for a program. Such problems will not occur in Java because we do not need to allocate or deallocate the memory in Java. Everything is taken care of by JVM only. For example, JVM allots the required memory by a Java program. Third reason is that Java supports an error checking feature at various stages: Early checking at compile-time and dynamic checking at run time.

**5. Secure:**

Since Java supports the distributed environment of the internet, it also provides multiple security features. Security problems like tampering, impersonation, and virus threats can be eliminated or reduced by using Java on Internet.

**6. System independence (Architecture neutral):**

Java compiler compiles the source code into byte code that is independent of any machine architecture. In other words, the byte code of Java program is not machine-dependent. It can be run on any system machine with any processor and any operating system that implements Java Virtual Machine. The goal of JavaSoft was to “write once; run anywhere, any time, forever”. To a great extent, JavaSoft accomplished this goal.

**7. Portability:**

If a program gives the same result on every system machine, that program is called portable. Java programs are portable.

**8. Interpreted:**

During compilation, Java compiler converts the source code of the program into byte code. This byte code can be executed on any system machine with the help of Java interpreter in JVM. If we take any other programming language, only a compiler or an interpreter is used to run programs. But in Java, we use both compiler and interpreter for the execution of the program.

**9. High Performance:**

The speed of interpreter inside JVM to execute a program is slow. To overcome this problem, JavaSoft team has introduced JIT (Just In Time) compiler which improves the performance of interpreting byte code by caching interpretations. Due to which the speed of execution of java program is enhanced. So, both interpreter and JIT compiler in JVM work together to run the program.

**10. Multithreaded:**

Java supports multi-threading programming that allows to write programs to do several works simultaneously. A thread is an individual process to execute a group of statements. JVM utilizes multiple threads to execute different blocks of code. Creating multiple threads is called ‘multithreaded’ in Java.

**11. Dynamic:**

Before the development of Java, only static text was displayed in the browser. But using applet program, we can also create animation dynamically on the Internet.

**4. EVOLUTION OF JAVA**

The evolution of Java spans several decades, marked by significant milestones, language enhancements, and technological advancements. Here's an overview of the key stages in the evolution of Java:

**Java 1.0 (1996):** Java was officially released by Sun Microsystems in 1996. It introduced core features such as the Java Virtual Machine (JVM), Java Applets for web development, and the Java Development Kit (JDK) for developers.

**Java 1.1 (1997):** This release focused on improving performance, stability, and adding new features such as the AWT event model and inner classes.

**Java 2 (J2SE 1.2, 1998):** A major release that introduced the "Java 2" branding. Introduced Swing GUI toolkit, Collections Framework, and Java Naming and Directory Interface (JNDI). Also introduced the "assert" keyword, strictfp modifier, and the Java Plug-in.

**J2SE 1.3 (2000):** Enhanced performance, stability, and introduced features like Java Sound API, Java Naming and Directory Interface (JNDI), and HotSpot JVM.

**J2SE 1.4 (2002):** Introduced significant language enhancements such as assert keyword, regular expressions, and the Java Web Start technology. Added the NIO (New I/O) package, XML processing APIs (JAXP), and integrated XML parser.

**J2SE 5.0 (2004):** A landmark release introducing major language enhancements under the codename "Tiger." Key features included generics, metadata annotations, enumerated types, autoboxing/unboxing, and enhanced for loop. Also introduced the concurrency utilities (java.util.concurrent) and the Scanner class.

**Java SE 6 (2006):** Focus on performance improvements, including compiler and runtime optimizations. Introduced scripting support with the inclusion of the Java Compiler API and scripting engine. Also introduced the Desktop API, XML digital signature API, and improvements to the Java Virtual Machine.

J**ava SE 7 (2011):** Introduced several language enhancements including the try-with-resources statement, diamond operator (<>), and switch statement with strings. Also introduced the Fork/Join framework for parallel programming, NIO.2 (java.nio) for asynchronous I/O operations, and the invokedynamic instruction for dynamic language support.

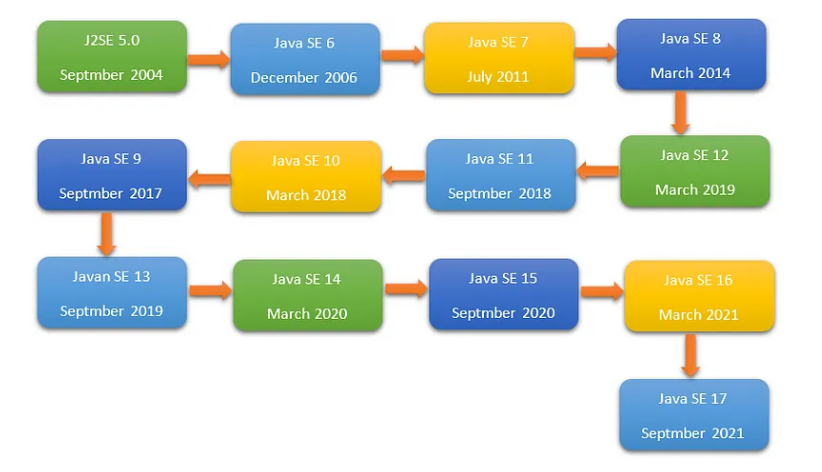
**Java SE 8 (2014):** A significant release introducing lambda expressions, the Stream API for functional-style programming, and the java.time package for date and time manipulation. Also introduced the Nashorn JavaScript engine, Compact Profiles for smaller runtime deployments, and improvements to the JVM (Metaspace replacing PermGen).

**Java SE 9 (2017):** Introduced the Java Platform Module System (JPMS) for modular development. Other features include the jshell tool for interactive Java programming, improvements to the Process API, and enhancements to the HTTP client.

**Java SE 10, 11, 12, 13, 14, 15, 16 (2018-2021):** These releases focused on incremental improvements, language enhancements, and performance optimizations. Features included local-variable type inference (var), enhancements to the garbage collector, improvements to the HttpClient, and preview features like switch expressions and text blocks.

**Java SE 17 (2021):** A long-term support (LTS) release, introducing features such as sealed classes and interfaces, pattern matching for switch, and enhancements to the garbage collectors. Also marked the end of the six-month release cadence, transitioning to a new release cycle with LTS releases every three years.

Throughout its evolution, Java has remained a popular and influential programming language, powering a wide range of applications from enterprise systems to mobile devices and the internet of things (IoT). It continues to adapt to changing technological landscapes while maintaining its core principles of portability, security, and robustness.



**5. ADVANTAGES AND DISADVANTAGES OF JAVA**

**Advantages:**

**Platform Independence:** Java programs can run on any device with a Java Virtual Machine (JVM), making them platform-independent. This "Write Once, Run Anywhere" (WORA) capability is one of Java's most significant advantages.

**Object-Oriented:** Java is a fully object-oriented programming language, which promotes modular design, code reusability, and easier maintenance.

**Rich API:** Java provides a vast standard library (Java API) covering areas such as networking, I/O, utilities, database access, and more. This extensive API reduces development time and effort by providing ready-to-use components.

**Strong Community Support:** Java has a large and active community of developers, offering abundant resources, tutorials, forums, and libraries to support development efforts.

**Automatic Memory Management:** Java's garbage collection mechanism automatically handles memory allocation and deallocation, reducing the risk of memory leaks and pointer-related errors.

**High Performance:** With advancements in JIT (Just-In-Time) compilation and optimization techniques, Java applications can achieve high performance comparable to native languages in many cases.

**Security:** Java has built-in security features such as bytecode verification, sandboxing, and a robust security manager, making it suitable for developing secure applications, especially in web and enterprise environments.

**Multi-threading Support:** Java provides built-in support for multi-threading, allowing developers to write concurrent and parallel programs easily.

**Scalability:** Java's architecture supports scalable applications, making it suitable for both small-scale projects and large enterprise systems.

**Disadvantages:**

**Performance Overhead:** Despite optimizations, Java applications may have a performance overhead compared to natively compiled languages like C or C++. This overhead is mainly due to the JVM's runtime environment and garbage collection.

**Memory Consumption:** Java applications typically consume more memory compared to native applications due to the JVM's memory requirements and runtime overhead.

**Complexity:** Java, being a feature-rich language, can sometimes be perceived as complex, especially for beginners. Its extensive API and object-oriented concepts may require a steep learning curve for some developers.

**Startup Time: Java** applications may have longer startup times compared to applications written in interpreted languages like Python or JavaScript, as they need to initialize the JVM and load classes.

**Browser Plugin Deprecation:** The decline of browser support for Java applets has reduced one of Java's significant use cases in web development, limiting its relevance for client-side web programming.

**Lack of Real-time Capabilities:** While Java is suitable for a wide range of applications, it may not be the best choice for real-time systems with strict timing requirements due to its non-deterministic garbage collection and other runtime characteristics.

**Verbosity:** Java code can sometimes be verbose, requiring more lines of code compared to other languages for accomplishing similar tasks. This verbosity can lead to increased development time and maintenance effort.

Despite these disadvantages, Java remains one of the most widely used and influential programming languages, powering a vast array of applications across various domains, including enterprise software, web development, mobile development (with Android), and big data processing.



**Fig 4: Evolution of JAVA**

1. **OBJECT ORIENTED CONCEPTS OF JAVA**

Object-oriented programming (OOP) is a fundamental programming paradigm based on the concept of “objects”. These objects can contain data in the form of fields (often known as attributes or properties) and code in the form of procedures (often known as methods). The core concept of the object-oriented approach is to break complex problems into smaller objects. OOPs in Java is useful for improving code readability and reusability with an efficient definition of a program. The primary OOPs concepts in Java include abstraction, encapsulation, inheritance, and polymorphism. The different Java OOPs concepts enable us to establish working methods and variables.

A diagram of different colored circles

Description automatically generated

**Fig 1: OOP in JAVA**

In the world of Java programming, it’s essential to grasp the fundamental object-oriented concepts in Java. Object-Oriented Programming, or OOPs in Java, is not just a set of rules; it’s a powerful paradigm that drives Java’s design and development principles.

**Objects:**

In Java, an object is more than just a runtime entity; it’s the embodiment of a class. These objects mirror real-world entities, complete with both state and behavior. Think of a ‘Car’ object based on a ‘Car’ class. It possesses attributes like color, brand, and speed, along with behaviors like ‘accelerate’ and ‘brake’. This is where Java OOP concepts come to life.

**Classes:**

A class in Java serves as a blueprint for creating objects. It bundles data (attributes) and behavior (methods) that define the object. Consider a ‘Car’ class; it defines attributes like color and methods like ‘accelerate()’. Essentially, a class is a template that characterizes what can be instantiated as an object, exemplifying OOPs concepts.

**Abstraction:**

Abstraction in Java involves concealing complexity while exposing only the essential aspects. It’s realized through abstract classes and interfaces. For instance, a ‘Vehicle’ interface declares a ‘move()’ method, but the actual implementation is left to classes like ‘Car’ or ‘Bike’. This focus on “what an object does” rather than “how it does it” is a core OOPs concept in Java.

**Inheritance:**

Inheritance is a mechanism where a new class (subclass) derives attributes and methods from an existing class (superclass). This fosters code reusability and establishes a hierarchical relationship between classes. For example, an ‘ElectricCar’ class can inherit traits from the ‘Car’ class, showcasing the practicality of Java OOPs concepts.

**Polymorphism:**

Polymorphism allows Java methods and objects to assume multiple forms. It finds common use in method overloading (same method name, different parameters) and method overriding (same method name and parameters in subclass as in parent class). Consider a ‘draw()’ method implemented differently in ‘Circle’, ‘Square’, and ‘Triangle’ classes as a prime example of polymorphism in Java.

**Encapsulation:**

Encapsulation is a pivotal OOPs principle in Java. It entails bundling data (attributes) and code (methods) into a cohesive unit. Moreover, it limits direct access to an object’s components, preventing inadvertent interference. Techniques like using private variables and offering public getter and setter methods exemplify Java OOPs concepts put into practice.

Java’s strength lies in its adherence to the OOP principles. Understanding the characteristics of OOPs in Java is pivotal for crafting efficient and modular code. These Java OOPs concepts form the foundation of modern software development, enabling developers to create elegant and scalable solutions.

**Association**

The OOPs concept in Java that signifies the relationship between the objects is called association. This OOPs concept in Java ensures that an object can be associated with one single object or multiple objects.

**Aggregation**

Aggregation is the OOPs concept in Java that paves the path for achieving association. Aggregation is symbolic of the weak relationship between objects. This OOPs concept in Java is symbolic of the relationship where one object contains other objects as a part of the state.

**Composition**

In Java object oriented programming, multiple ways to achieve association are present. Composition is yet another way of achieving association. It is one of the Java OOPs concepts that represent the strong relationship between the dependent object and the containing object.

**Advantages of OOPs Concept (Object-Oriented Programming System)**

* Reusability: A reusable code is something that’s written once but used multiple times. The use of class ensures that a particular code can be used an infinite number of times.
* Data Redundancy: One of the greatest advantages of Java OOPs concepts is data redundancy. It can be created at the data storage by holding the same piece of data in two different places. When you wish to utilize similar functionality in multiple classes, you can write common class definitions by inheriting them.
* Code Maintenance: The OOPs concept in Java makes it easy to maintain existing code as new objects can be created with minor differences from the existing ones. It enables users to rework multiple times and alter the existing code by incorporating new changes.
* Security: The Java OOPs concepts like abstracting and data hiding allow filtering out limited exposure. Therefore, the features of OOPs make only necessary data visible to maintain security.
* Design Benefits: The OOPs in Java enable designers to have a more extensive design phase. It leads to the creation of better designs. The program reaches critical limits after a certain amount of time. At that point, it becomes easy to program all non-oops individually.
* Easy Troubleshooting: The OOPs concept in Java enables the use of self-constrained encapsulation objects. Therefore, developers can easily solve ongoing issues. The features of OOPs also prevent the issue of duplicate codes.

**Disadvantages of OOPs Concept (Object-Oriented Programming System)**

Once you understand what is object oriented programming in Java and its benefits, you should also learn about some of its drawbacks. The Java OOPs concepts require a lot of effort. Moreover, object oriented programming is slower than other programs. The size of OOPs is also bigger than others.

**Best Practices for OOP Concepts in Java**

The goal of the OOPs concept in Java is to save time without compromising ease of use and security. Some of the best practices of object oriented programming to achieve that goal are as follows:

* DRY or Don’t Repeat Yourself is a primacy concept in Java. Therefore, you should never use two blocks of identical code in two different places. You must be using a single method for different applications.
* If you want the Java code to alter in the future, you should encapsulate it by making every method and variable private at the outset. You can increase access to “protected” with time, if necessary. But don’t make it too public.
* Single responsibility is one of the best practices for Java OOPs concepts. It involves ensuring that a class has a single functionality. It ensures that the class can be used or extended on its own when new applications for it emerge. It will help eliminate coupling between multiple functionalities.
* The open-closed design is one of the best practices for the OOPs concept in Java. You will have to close all classes and methods for modification but keep them open for extensions. It will ensure that tried and tested codes of Java OOPs concepts remain static. But they can be easily modified to complete new tasks when required.

A diagram of a function

Description automatically generated

**Fig 2: Data and methods**

1. **ENCAPUSLATION**

Encapsulation is one of the four fundamental Object-Oriented Programming (OOP) concepts, along with inheritance, polymorphism, and abstraction. In Java, encapsulation refers to the bundling of data (variables) and methods that operate on the data into a single unit called a class. It helps in hiding the internal state of an object and only exposing the necessary functionality through methods.

A close-up of a capsule

Description automatically generated

**Fig 3: Encapsulation**

**Advantages of Encapsulation:**

* Data Hiding: Encapsulation allows you to hide the internal state of an object from the outside world. This prevents direct access to the object's data, reducing the risk of unintended manipulation.
* Modularity: By bundling related data and methods together within a class, encapsulation promotes modularity and organizes code into manageable units. This enhances code readability, maintainability, and reusability.
* Flexibility: Encapsulation allows you to change the internal implementation of a class without affecting the external code that uses the class. Clients interact with the class through its public interface, not its internal details.
* Security: Encapsulation provides a level of security by controlling access to the data members of a class. By defining appropriate access modifiers (public, private, protected), you can restrict access to sensitive data and expose only the necessary functionality.

**Implementation of Encapsulation in Java:**

* Private Access Modifier: Java provides the private access modifier to restrict access to members (variables and methods) to within the same class. Private members are not accessible outside the class.

|  |
| --- |
| public class MyClass {  private int myNumber; // private data member  public void setMyNumber(int num) {  this.myNumber = num; // public method to set the value of myNumber  }  public int getMyNumber() {  return this.myNumber; // public method to get the value of myNumber  } } |

* Public Methods: Public methods are used to provide controlled access to private data members. These methods are typically referred to as getter and setter methods, allowing external classes to read and modify the encapsulated data.

|  |
| --- |
| MyClass obj = new MyClass();  obj.setMyNumber(10); // Set the value of myNumber using a public method  int number = obj.getMyNumber(); // Retrieve the value of myNumber using a public method |

* Access Modifiers: Java provides four access modifiers (public, private, protected, and default) to control access to class members. By choosing appropriate access modifiers, you can enforce encapsulation and regulate access to class members.

|  |
| --- |
| public class MyClass {  private int myNumber; // private data member  // public method to set the value of myNumber  public void setMyNumber(int num) {  this.myNumber = num;  }  // public method to get the value of myNumber  public int getMyNumber() {  return this.myNumber;  } } |

* Encapsulation is a key principle in Java programming and is used extensively to create robust, maintainable, and secure code. By encapsulating data and methods within classes, Java promotes code organization, modularity, and flexibility, facilitating the development of complex software systems.

1. **POLYMORPHISM**

Polymorphism is one of the fundamental concepts of Object-Oriented Programming (OOP), and it allows objects of different classes to be treated as objects of a common superclass. In Java, polymorphism is achieved through method overriding and method overloading.

**Method Overriding:**

Method overriding occurs when a subclass provides a specific implementation of a method that is already defined in its superclass. The subclass overrides the superclass method with its own implementation, but the method signature remains the same.

|  |
| --- |
| class Animal {  void makeSound() {  System.out.println("Animal makes a sound");  } }  class Dog extends Animal {  @Override  void makeSound() {  System.out.println("Dog barks");  } }  class Cat extends Animal {  @Override  void makeSound() {  System.out.println("Cat meows");  } }  public class Main {  public static void main(String[] args) {  Animal dog = new Dog();  Animal cat = new Cat();  dog.makeSound(); // Output: Dog barks  cat.makeSound(); // Output: Cat meows  } } |

In the above example, both Dog and Cat classes override the makeSound() method defined in the Animal class. When makeSound() is called on dog and cat objects, the overridden method in each subclass is executed.

**Method Overloading:**

Method overloading occurs when a class defines multiple methods with the same name but different parameter lists. The methods must have different signatures, which means they differ in the number or types of parameters.

|  |
| --- |
| class Calculator {  int add(int a, int b) {  return a + b;  }  double add(double a, double b) {  return a + b;  } }  public class Main {  public static void main(String[] args) {  Calculator calculator = new Calculator();  int sum1 = calculator.add(5, 10); // Invokes the int version of add method  double sum2 = calculator.add(3.5, 2.5); // Invokes the double version of add method  System.out.println("Sum1: " + sum1); // Output: Sum1: 15  System.out.println("Sum2: " + sum2); // Output: Sum2: 6.0  } } |