* **C is procedure-oriented**, and **garbage collection in C is manual**.
* **Java is object-oriented**. **Everything in Java depends on objects**, data abstraction, and polymorphism.
* Java is **efficient and fast in execution**.
* The **class loader** is part of the **Java Runtime Environment** responsible for loading Java classes into the **Java Virtual Machine (JVM)**.
* It helps developers to **create, compile, package, and run Java applications**. It is essential for Java developers.
* The JVM interprets Java bytecode, making **Java applications platform-independent**.

**Concrete implementation** refers to a specific, fully defined implementation of an interface.

**Runtime instance** refers to an object that is created and exists in memory.

**Dynamic typing** in programming languages means that **type checking is performed at runtime**.

**Access modifiers in Java**:

* **public**: Accessible from any other class.
* **private**: Accessible only within the class in which it is declared.
* **protected**: Accessible within the same class, subclasses, and within the same package.

A **constructor** in Java:

* Has the same name as the class name.
* Does not have a return type.
* Can be overloaded but cannot be overridden.

**Method Overloading**:

* **Occurs in the same class**.
* Allows methods with the **same name but different parameters**.
* Java supports method overloading through two mechanisms: **different number of parameters** or **different types of parameters**.

**Method Overriding**:

* Occurs when a subclass provides a specific implementation of a method declared in one of its parent classes.
* Used to provide a **specific implementation of a method**.

**Destructor** is used to destroy objects.

**this keyword**:

* Refers to the **current object**.
* Used to differentiate **instance variables** from **local variables**.

**Member function**: Also known as a method, is a block of code within a class that performs a specific task.

**Shadowing**: Occurs when a **local variable** or **parameter** has the same name as an **instance variable**.

**Accessor (Getter)**: Retrieves the value of a private instance variable.

**Mutator (Setter)**: Modifies the value of a private instance variable.

**Inheritance Types**:

* Single level
* Multiple
* Multi-level
* Hierarchical
* Hybrid

**Encapsulation**:

* A data hiding mechanism that groups together data and methods acting on that data.
* Example: A capsule.
* Achieved by providing only getter and setter methods, making the class read-only or write-only.
* Improves unit testing.

**Polymorphism**:

* Performing a single action in different ways.
* **Types**:
  + **Runtime polymorphism**: Determined by the **virtual function & pointer**.
  + **Compile-time polymorphism**: Determined by the **compiler**, achieved through **operator overloading and function overloading**.

**Abstraction**:

* Hiding implementation details and showing only functionality.
* Achieved through **abstract classes** (0-100% abstraction) and **interfaces** (100% abstraction).

**Static keyword**:

* **Static variable**: Belongs to the class rather than any specific instance.
* **Static method**: Part of the class, accessible by every instance of the class.
* **Static block**: A block marked with **static** keyword.

**Composition**: A design technique to implement **has-a relationship**.

**Aggregation**: A class having a reference to another class. Represents a **has-a relationship**.

**Containment**: Also known as composition or has-a relationship.

**super keyword**:

* Refers to the immediate parent class object.
* Used to invoke immediate parent class methods or constructors.

**Overloading vs Overriding**:

* **Overloading**:
  1. Done in the same class.
  2. Useful for program design.
  3. Different methods are overloaded.
* **Overriding**:
  1. Done in inherited classes.
  2. Message is the same but implementation is specific to the derived class.
  3. Methods have to be the same in the derived class.

**Protected keyword**:

* An access modifier accessible within the package and outside the package through inheritance only.
* Cannot be assigned to outer classes or interfaces.

**Final keyword**:

* Used to restrict the user.
* **Final variable**: Cannot be changed.
* **Final method**: Cannot be overridden.
* **Final class**: Cannot be extended.

**Object class in Java**:

* The parent class of all Java classes.
* Methods include **toString()**, **equals()**, **hashCode()**, **getClass()**, **finalize()**, **clone()**, **wait(long timeout)**, **notify()**, and **notifyAll()**.

**Concrete class**:

* Can be instantiated.
* Implements all methods of its superclasses or interfaces.

**Interface**:

* A blueprint of a class with static constants and abstract methods.
* Achieves abstraction and multiple inheritance.
* **Functional interface**: An interface with only one abstract method. Examples include **Runnable**, **ActionListener**, and **Comparable**.

**Lambda expressions**:

* Introduced in Java 8.
* Provide a clear way to represent a single-method interface.

**String**:

* A collection of characters.
* Immutable.
* Methods: **compareTo()**, **equals()**.

**Thread-safe code**:

* Ensures shared data is accessed and modified safely using synchronization.

**StringBuffer**:

* Used to create mutable string objects.
* Thread-safe.
* Methods: **append**, **insert**, **replace**, **delete**, **reverse**, **capacity**, **length**, **substring**.

**StringBuilder**:

* Represents a mutable sequence of characters.
* Not synchronized, hence not thread-safe.
* Introduced in JDK 1.5.

**Marker interface**:

* An interface without methods, fields, or constants.
* Provides runtime information to the JVM.
* Examples: **Serializable**, **Cloneable**.

**Inner class instantiation**:

java

Copy code

OuterClass.InnerClass innerObject = outerObject.new InnerClass();

**Anonymous inner class**:

* An inner class without a name, used to override a method of a class or interface.

**Functional interface**:

* An interface with only one abstract method.
* Supports lambda expressions.
* Examples: **Runnable**, **ActionListener**, **Comparable**.

**String methods**:

* **compareTo()**: Compares two strings lexicographically.
* **equals()**: Checks if the content of two strings is the same.

**StringBuffer vs StringBuilder**:

* **StringBuffer** is thread-safe but less efficient.
* **StringBuilder** is not thread-safe but more efficient.

**Encapsulation**:

* Wrapping code and data together.
* Achieved by providing getter and setter methods.

**Polymorphism**:

* Runtime polymorphism: Also known as dynamic binding.
* Compile-time polymorphism: Also known as static binding.

**Abstraction**:

* Hiding implementation details and showing only the essential features.

**Concrete class**:

* A class that can be instantiated.

**Interface**

Interface in Java is a blueprint of a class. It can contain static constants and abstract methods. The interface in Java is a mechanism to achieve abstraction and multiple inheritance.

Since Java 8, interfaces can have default and static methods. Since Java 9, interfaces can also have private methods.

**Key Points:**

* Loose coupling: Interfaces help achieve loose coupling in code.
* Constants and Abstract Methods: The interface is essentially a collection of constants and abstract methods.
* Programming by Contract: Data members in an interface are always public, static, and final.
* A subclass can only have a single superclass in Java, but a class can implement multiple interfaces.
* If a class implements an interface, it must implement all the methods of the interface.

**Syntax:**

* class-to-class: class A extends B
* interface-to-interface: interface A extends B
* class-to-interface: class A implements B

**Default Methods:**

* Before Java 8, interfaces could only have abstract methods. If a new method was added to an interface, its implementation had to be provided in the implementing classes.
* To overcome this issue, Java introduced default methods, which allow methods to have an implementation in the interface.

**Marker Interface:**

* An interface that does not contain any methods, fields, or constants is known as a marker interface or tag interface.
* It provides runtime information to the JVM and compiler.
* Examples: Serializable, Cloneable.

**Cloneable Interface:**

* Used to create an exact copy of an object.
* Exists in the java.lang package.
* A class must implement the Cloneable interface if it wants to create a clone of its objects.

**Example:**

**java**

**Copy code**

OuterClass.InnerClass innerObject = outerObject.new InnerClass();

**Anonymous Inner Class:**

* An anonymous inner class is an inner class without a name, for which only a single object is created.
* Used to override a method of a class or interface.

**Functional Interface:**

* A functional interface has only one abstract method.
* From Java 8 onwards, lambda expressions can be used to represent the implementation of a functional interface.
* A functional interface can have any number of default and static methods but only one abstract method.
* Also known as SAM (Single Abstract Method) interfaces.
* Examples: Runnable, ActionListener, Comparable.

**Lambda Expressions:**

* Provide a clear and consistent way to represent a single-method interface using an expression.
* No need to define a separate implementation for the method.

**String:**

* A collection of characters.
* Immutable.
* **Methods:**
  + **compareTo():** Compares two strings lexicographically.
  + **equals():** Checks if the content of two strings is the same.

**Thread Safety:**

* Ensures that shared data is accessible and modified in a way that prevents data corruption and unexpected behaviour.
* Typically achieved using synchronization mechanisms.

**StringBuffer:**

* Used to create mutable string objects.
* **Thread-safe:** Multiple threads cannot access the same instance simultaneously, ensuring order and safety.
* **Methods:** append, insert, replace, delete, reverse, capacity, length, substring.
  + **append():** Concatenates the given argument with the existing string.

**StringBuilder:**

* Represents a mutable sequence of characters.
* Not synchronized, hence not thread-safe.
* Introduced in JDK 1.5.
* More efficient than StringBuffer when thread safety is not required.

**Comparison:**

* **StringBuffer:**
  + Introduced in Java 1.0.
  + Thread-safe.
* **StringBuilder:**
  + Introduced in Java 1.5.
  + Not synchronized, hence not thread-safe.