Classes & Objects

**Classes**

A **class** is like a blueprint or template for creating objects. It defines a set of attributes and methods that the created objects will have. Think of a class as a general concept or category.

For example, consider a class called Car. This class might have attributes like color, make, model, and year, and methods like start(), stop(), and drive().

**Objects**

An **object** is an instance of a class. When you create an object, you are creating a specific example of the class with actual values. Using the Car class example, an object could be a specific car like a red 2020 Toyota Corolla.

Here’s a simple example in Java:

**Java**

// Define the Car class

public class Car {

// Attributes

String color;

String make;

String model;

int year;

// Method

void start() {

System.out.println("Car is starting");

}

}

// Create an object of the Car class

public class Main {

public static void main(String[] args) {

Car myCar = new Car(); // Create a new Car object

myCar.color = "Red";

myCar.make = "Toyota";

myCar.model = "Corolla";

myCar.year = 2020;

// Use the object's method

myCar.start(); // Output: Car is starting

}

}

AI-generated code. Review and use carefully. [More info on FAQ](https://www.bing.com/new#faq).

In this example:

* Car is the class.
* myCar is an object of the Car class.
* The attributes color, make, model, and year are assigned specific values for myCar.
* The method start() is called on the myCar object.

**Key Points**

* **Class**: A blueprint for objects. Defines attributes and methods.
* **Object**: An instance of a class. Has specific values for the attributes defined by the class.

**The Stack & Heap**

Stack Memory

* Purpose: Used for static memory allocation and the execution of a thread.
* Contents: Stores primitive data types (e.g., int, char, float) and references to objects in the heap.
* LIFO Order: Operates in a Last-In-First-Out (LIFO) manner. Each time a method is called, a new block (stack frame) is created on top of the stack to hold local variables and references.
* Scope and Lifetime: Variables in the stack exist only as long as the method that created them is running. When the method finishes, the stack frame is removed, and the memory is freed.
* Speed: Access to stack memory is very fast.
* Thread Safety: Each thread has its own stack, making it inherently thread-safe.

Heap Memory

* Purpose: Used for dynamic memory allocation for Java objects and JRE classes at runtime.
* Contents: Stores all objects and arrays. When an object is created, it is always stored in the heap space.
* Global Access: Objects in the heap can be accessed globally, meaning any part of the application can access them.
* Garbage Collection: Java’s automatic garbage collector manages heap memory, freeing up space by removing objects that are no longer in use.
* Generations: The heap is divided into generations to optimize garbage collection:
  + Young Generation: Where new objects are allocated and aged.
  + Old (Tenured) Generation: Where long-surviving objects are stored.
  + Permanent Generation (Metaspace in Java 8 and later): Stores metadata for the runtime classes and methods.

Key Differences

* Memory Allocation: Stack memory is used for static memory allocation, while heap memory is used for dynamic memory allocation.
* Management: Stack memory is automatically managed (allocated and deallocated), whereas heap memory requires manual management through garbage collection.
* Performance: Stack memory access is faster compared to heap memory due to its LIFO order and smaller size.
* Scope: Stack memory is limited to the scope of the method, while heap memory is accessible throughout the application.

Example in Java

Here’s a simple example to illustrate the difference:

Java

public class MemoryExample {

public static void main(String[] args) {

int stackVar = 10; // Allocated on the stack

Car myCar = new Car(); // Allocated on the heap

myCar.color = "Red";

myCar.make = "Toyota";

myCar.model = "Corolla";

myCar.year = 2020;

}

}

class Car {

String color;

String make;

String model;

int year;

}

In this example:

* stackVar is a primitive variable allocated on the stack.
* myCar is a reference variable allocated on the stack, but the actual Car object is allocated on the heap.

**Garbage Collection**

Garbage collection in Java is an automatic process that helps manage memory by reclaiming memory occupied by objects that are no longer in use. This process is handled by the Java Virtual Machine (JVM) and ensures efficient memory utilization, preventing memory leaks and optimizing performance.

How Garbage Collection Works

1. Object Creation: When an object is created in Java, it is allocated memory on the heap.
2. Reference Tracking: The JVM keeps track of references to objects. An object is considered “reachable” if it can be accessed through any reference.
3. Unreachable Objects: When an object is no longer referenced, it becomes eligible for garbage collection. The garbage collector identifies these unreachable objects and reclaims their memory.

Types of Garbage Collection

Java uses several garbage collection algorithms, each suited for different scenarios:

1. Serial Garbage Collector: Suitable for single-threaded environments. It uses a single thread to perform all garbage collection tasks.
2. Parallel Garbage Collector: Uses multiple threads to speed up the garbage collection process. Ideal for multi-threaded applications.
3. Concurrent Mark-Sweep (CMS) Collector: Aims to minimize pause times by performing most of the garbage collection work concurrently with the application threads.
4. G1 Garbage Collector: Designed for applications with large heaps. It divides the heap into regions and performs garbage collection incrementally to reduce pause times.

Phases of Garbage Collection

1. Marking: The garbage collector identifies all reachable objects.
2. Deletion: Unreachable objects are deleted, and their memory is reclaimed.
3. Compaction: The heap is compacted to eliminate fragmentation and make memory allocation more efficient.

Example

Here’s a simple example to illustrate garbage collection in Java:

Java

public class GarbageCollectionExample {

public static void main(String[] args) {

Car myCar = new Car(); // Object created on the heap

myCar = null; // The object is now eligible for garbage collection

// Requesting JVM to run Garbage Collector

System.gc();

}

}

class Car {

@Override

protected void finalize() throws Throwable {

System.out.println("Car object is garbage collected");

}

}

AI-generated code. Review and use carefully. [More info on FAQ](https://www.bing.com/new#faq).

In this example:

* The Car object is created and then dereferenced by setting myCar to null.
* The System.gc() method is called to suggest that the JVM performs garbage collection.
* The finalize() method is overridden to provide a custom cleanup action before the object is garbage collected.

Key Points

* Automatic Memory Management: Java handles memory management automatically, reducing the risk of memory leaks.
* Garbage Collector: A daemon thread that runs in the background to reclaim memory.
* Efficiency: Different garbage collection algorithms optimize performance based on application needs.

**Classes - Members (Fields, Constructors, Methods)**

Fields

Fields, also known as attributes or properties, are variables that hold the data or state of an object. They are defined within a class and can have different access levels (e.g., private, protected, public).

Example in Java:

Java

public class Car {

// Fields

private String color;

private String make;

private String model;

private int year;

}

Constructors

Constructors are special methods that are called when an object is instantiated. They initialize the object’s fields and set up any necessary state. Constructors have the same name as the class and do not have a return type.

Example in Java:

Java

public class Car {

private String color;

private String make;

private String model;

private int year;

// Constructor

public Car(String color, String make, String model, int year) {

this.color = color;

this.make = make;

this.model = model;

this.year = year;

}

}

Methods

Methods define the behavior of an object. They are functions that can perform actions, manipulate fields, and return values. Methods can also have different access levels and can be overloaded (multiple methods with the same name but different parameters).

Example in Java:

Java

public class Car {

private String color;

private String make;

private String model;

private int year;

public Car(String color, String make, String model, int year) {

this.color = color;

this.make = make;

this.model = model;

this.year = year;

}

// Method

public void start() {

System.out.println("Car is starting");

}

public void drive() {

System.out.println("Car is driving");

}

}

Putting It All Together

Here’s a complete example that includes fields, a constructor, and methods:

Java

public class Car {

private String color;

private String make;

private String model;

private int year;

// Constructor

public Car(String color, String make, String model, int year) {

this.color = color;

this.make = make;

this.model = model;

this.year = year;

}

// Methods

public void start() {

System.out.println("Car is starting");

}

public void drive() {

System.out.println("Car is driving");

}

public void displayInfo() {

System.out.println("Car Info: " + color + " " + make + " " + model + " " + year);

}

public static void main(String[] args) {

Car myCar = new Car("Red", "Toyota", "Corolla", 2020);

myCar.start();

myCar.drive();

myCar.displayInfo();

}

}

In this example:

* Fields: color, make, model, and year store the state of the Car object.
* Constructor: Initializes the fields when a new Car object is created.
* Methods: start(), drive(), and displayInfo() define the behavior of the Car object.

**Access Modifiers**

Access modifiers in Java control the visibility and accessibility of classes, methods, constructors, and fields. They help in encapsulating the data and ensuring that only the intended parts of a program can access certain components. Here are the four main access modifiers in Java:

**1. Private**

* **Scope**: Accessible only within the same class.
* **Usage**: Typically used to hide the internal details of a class.
* **Example**:

**Java**

public class Car {

private String color;

private String make;

private String model;

private int year;

private void displayInfo() {

System.out.println("Car Info: " + color + " " + make + " " + model + " " + year);

}

}

**2. Default (Package-Private)**

* **Scope**: Accessible only within the same package. If no access modifier is specified, it defaults to package-private.
* **Usage**: Useful for grouping related classes that should interact with each other but not with classes outside the package.
* **Example**:

**Java**

class Car {

String color;

String make;

String model;

int year;

void displayInfo() {

System.out.println("Car Info: " + color + " " + make + " " + model + " " + year);

}

}

**3. Protected**

* **Scope**: Accessible within the same package and by subclasses in other packages.
* **Usage**: Often used in inheritance to allow subclasses to access certain fields or methods.
* **Example**:

**Java**

public class Car {

protected String color;

protected String make;

protected String model;

protected int year;

protected void displayInfo() {

System.out.println("Car Info: " + color + " " + make + " " + model + " " + year);

}

}

**4. Public**

* **Scope**: Accessible from any other class.
* **Usage**: Used for classes, methods, and fields that need to be accessible from other parts of the program.
* **Example**:

**Java**

public class Car {

public String color;

public String make;

public String model;

public int year;

public void displayInfo() {

System.out.println("Car Info: " + color + " " + make + " " + model + " " + year);

}

}

**Summary Table**

**Table**

| **Modifier** | **Class** | **Package** | **Subclass** | **World** |
| --- | --- | --- | --- | --- |
| **Private** | Y | N | N | N |
| **Default** | Y | Y | N | N |
| **Protected** | Y | Y | Y | N |
| **Public** | Y | Y | Y | Y |

**Non-Access Modifiers**

Non-access modifiers in Java provide additional information about the behavior and characteristics of classes, methods, and variables. Here are the main non-access modifiers:

**1.**static

* **Purpose**: Indicates that a member belongs to the class itself rather than to any specific instance.
* **Usage**: Commonly used for constants, utility methods, and shared resources.
* **Example**:

**Java**

public class Car {

static int numberOfCars = 0; // Shared among all instances

public Car() {

numberOfCars++;

}

public static void displayTotalCars() {

System.out.println("Total cars: " + numberOfCars);

}

}

**2.**final

* **Purpose**: Prevents modification. When applied to a variable, it makes it a constant. When applied to a method, it prevents overriding. When applied to a class, it prevents inheritance.
* **Usage**: Used to define constants, secure methods from being overridden, and prevent classes from being subclassed.
* **Example**:

**Java**

public final class Car {

final int maxSpeed = 200; // Constant

public final void displayInfo() {

System.out.println("Max speed: " + maxSpeed);

}

}

**3.**abstract

* **Purpose**: Indicates that a class cannot be instantiated and may contain abstract methods that must be implemented by subclasses.
* **Usage**: Used to define abstract classes and methods that provide a template for subclasses.
* **Example**:

**Java**

public abstract class Vehicle {

abstract void start(); // Abstract method

public void stop() {

System.out.println("Vehicle stopped");

}

}

public class Car extends Vehicle {

@Override

void start() {

System.out.println("Car is starting");

}

}

**4.**synchronized

* **Purpose**: Ensures that a method or block of code is accessed by only one thread at a time.
* **Usage**: Used in multi-threaded applications to prevent concurrent access issues.
* **Example**:

**Java**

public class BankAccount {

private int balance = 1000;

public synchronized void deposit(int amount) {

balance += amount;

}

public synchronized void withdraw(int amount) {

balance -= amount;

}

}

**5.**volatile

* **Purpose**: Indicates that a variable’s value will be modified by different threads.
* **Usage**: Used to ensure visibility of changes to variables across threads.
* **Example**:

**Java**

public class SharedData {

private volatile boolean flag = true;

public void setFlag(boolean flag) {

this.flag = flag;

}

public boolean getFlag() {

return flag;

}

}

**6.**transient

* **Purpose**: Prevents a field from being serialized.
* **Usage**: Used in serialization to exclude certain fields.
* **Example**:

**Java**

import java.io.\*;

public class Car implements Serializable {

private String model;

private transient int year; // Will not be serialized

public Car(String model, int year) {

this.model = model;

this.year = year;

}

}

**7.**native

* **Purpose**: Indicates that a method is implemented in native code using another programming language (e.g., C or C++).
* **Usage**: Used for performance-critical operations or to access system-specific features.
* **Example**:

**Java**

public class NativeExample {

public native void nativeMethod();

static {

System.loadLibrary("NativeLib");

}}

Static Members

Static members in Java are class-level members that are shared among all instances of the class. They can be fields (variables) or methods. Let’s break down the key aspects of static members:

**Static Fields**

* **Shared Across Instances**: A static field is shared by all instances of the class. There is only one copy of the static field, regardless of how many objects are created.
* **Class-Level Variable**: It belongs to the class itself rather than any specific instance.
* **Initialization**: Static fields can be initialized when they are declared or in a static block.

Example:

**Java**

public class Car {

// Static field

public static int numberOfCars = 0;

// Instance fields

private String color;

private String make;

// Constructor

public Car(String color, String make) {

this.color = color;

this.make = make;

numberOfCars++; // Increment the static field

}

}

**Static Methods**

* **Class-Level Methods**: Static methods belong to the class rather than any instance. They can be called without creating an instance of the class.
* **Accessing Static Fields**: Static methods can access static fields and other static methods directly.
* **No**this**Keyword**: Static methods cannot use the this keyword because they are not associated with any instance.

Example:

**Java**

public class Car {

public static int numberOfCars = 0;

public Car(String color, String make) {

this.color = color;

this.make = make;

numberOfCars++;

}

// Static method

public static void displayTotalCars() {

System.out.println("Total cars: " + numberOfCars);

}

}

**Static Blocks**

* **Initialization Block**: Static blocks are used to initialize static fields. They are executed when the class is loaded into memory.
* **Multiple Static Blocks**: A class can have multiple static blocks, which are executed in the order they appear.

Example:

**Java**

public class Car {

public static int numberOfCars;

// Static block

static {

numberOfCars = 0;

System.out.println("Static block executed");

}

public Car(String color, String make) {

this.color = color;

this.make = make;

numberOfCars++;

}

}

**Key Points**

* **Single Copy**: Only one copy of static fields exists, shared among all instances.
* **Class-Level Access**: Static methods and fields can be accessed using the class name.
* **Initialization**: Static fields can be initialized in static blocks or at the point of declaration.

**Example Usage**

Here’s a complete example demonstrating static fields, methods, and blocks:

**Java**

public class Car {

public static int numberOfCars;

static {

numberOfCars = 0;

System.out.println("Static block executed");

}

private String color;

private String make;

public Car(String color, String make) {

this.color = color;

this.make = make;

numberOfCars++;

}

public static void displayTotalCars() {

System.out.println("Total cars: " + numberOfCars);

}

public static void main(String[] args) {

Car car1 = new Car("Red", "Toyota");

Car car2 = new Car("Blue", "Honda");

Car.displayTotalCars(); // Output: Total cars: 2

}

}

In this example:

* The static field numberOfCars keeps track of the total number of Car objects created.
* The static method displayTotalCars() prints the total number of cars.
* The static block initializes the static field when the class is loaded.

**Interface and Abstract classes**

In Java, both interfaces and abstract classes are used to achieve abstraction, but they serve different purposes and have distinct characteristics. Let’s explore their differences and use cases.

Abstract Classes

An abstract class is a class that cannot be instantiated on its own and is meant to be subclassed. It can contain both abstract methods (without implementation) and concrete methods (with implementation).

Key Points:

* Abstract Methods: Must be implemented by subclasses.
* Concrete Methods: Can be used directly or overridden by subclasses.
* Fields: Can have instance variables and static variables.
* Constructors: Can have constructors to initialize fields.
* Inheritance: A class can extend only one abstract class.

Example:

Java

public abstract class Shape {

String color;

// Constructor

public Shape(String color) {

this.color = color;

}

// Abstract method

abstract double area();

// Concrete method

public void displayColor() {

System.out.println("Color: " + color);

}

}

public class Circle extends Shape {

double radius;

public Circle(String color, double radius) {

super(color);

this.radius = radius;

}

@Override

double area() {

return Math.PI \* radius \* radius;

}

}

Interfaces

An interface is a reference type in Java, similar to a class, that can contain only abstract methods (until Java 8, which introduced default and static methods). Interfaces are used to specify a set of methods that a class must implement.

Key Points:

* Abstract Methods: All methods are abstract by default (until Java 8).
* Default and Static Methods: Can have default and static methods (since Java 8).
* Fields: Can only have static and final variables (constants).
* Multiple Inheritance: A class can implement multiple interfaces.
* No Constructors: Interfaces cannot have constructors.

Example:

Java

public interface Drawable {

void draw();

}

public class Rectangle implements Drawable {

@Override

public void draw() {

System.out.println("Drawing a rectangle");

}

}

Differences Between Abstract Classes and Interfaces

Table

| Feature | Abstract Class | Interface |
| --- | --- | --- |
| Methods | Can have both abstract and concrete methods | Only abstract methods (default and static methods since Java 8) |
| Fields | Can have instance variables | Only static and final variables |
| Constructors | Can have constructors | Cannot have constructors |
| Inheritance | Can extend one class | Can implement multiple interfaces |
| Access Modifiers | Can have any access modifier | Methods are implicitly public |

Use Cases

* Abstract Classes: Use when you have a base class that should not be instantiated and you want to share code among related classes.
* Interfaces: Use when you want to define a contract that multiple classes can implement, regardless of their place in the class hierarchy.

Example Combining Both

Java

public interface Movable {

void move();

}

public abstract class Vehicle implements Movable {

String name;

public Vehicle(String name) {

this.name = name;

}

public void displayInfo() {

System.out.println("Vehicle: " + name);

}

}

public class Car extends Vehicle {

public Car(String name) {

super(name);

}

@Override

public void move() {

System.out.println("Car is moving");

}

}

In this example:

* Movable is an interface with an abstract method move().
* Vehicle is an abstract class that implements Movable and provides a concrete method displayInfo().
* Car is a concrete class that extends Vehicle and implements the move() method.