Abstraction in Java is a core **OOP concept** used to **hide the complex implementation details** and **show only the essential information** to the user. It focuses on *what* an object does rather than *how* it does it.

You can achieve abstraction in Java using **abstract classes** and **interfaces**.

**1. Abstraction with Abstract Classes**

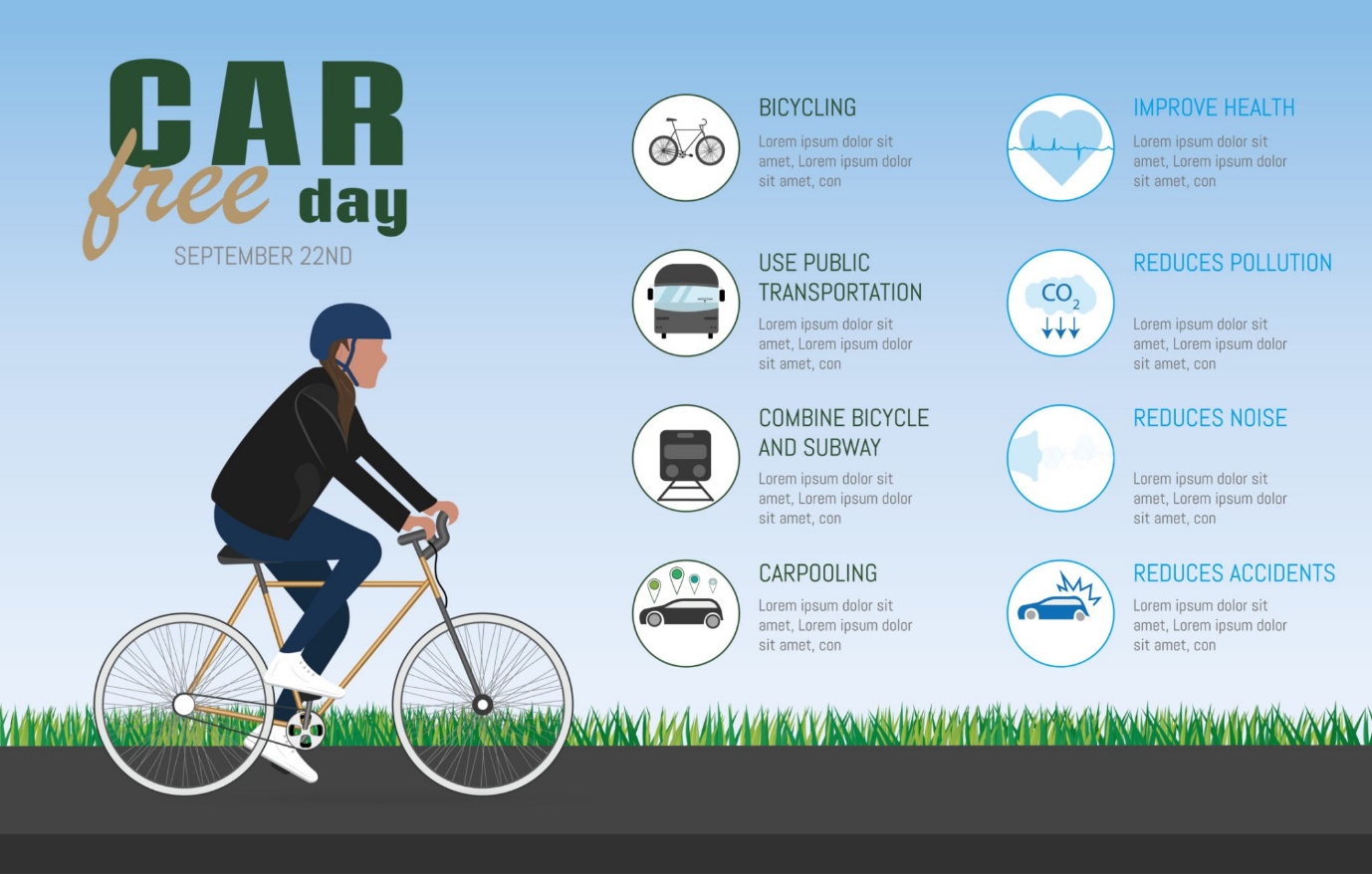
An **abstract class** is a class that is declared with the abstract keyword. It can have abstract methods (methods without a body) and concrete methods (methods with a body). You **cannot create objects** of an abstract class.

**Key Concepts:**

* **Abstract Method:** A method declared with the abstract keyword and no implementation (body). Subclasses *must* provide the implementation for these methods.
* **Inheritance:** A subclass extends the abstract class and provides the implementation for all its abstract methods.

**Example: The Vehicle Class**

Imagine a Vehicle class



Getty Images

Explore

where every vehicle must have a way to start and stop, but the way they do it differs (e.g., a car uses an ignition key, a motorcycle uses a kick-starter).

Java

// Abstract Class

abstract class Vehicle {

String brand = "Generic Vehicle";

// 1. Abstract method (no body)

// Subclasses MUST implement this.

abstract void startEngine();

// 2. Concrete method (with body)

// Shared functionality for all vehicles.

public void stop() {

System.out.println(brand + " has stopped.");

}

}

// Subclass inheriting from the abstract class

class Car extends Vehicle {

// Constructor to set the brand

public Car() {

this.brand = "Toyota";

}

// Implementation of the abstract method

@Override

void startEngine() {

System.out.println("Car started with a key ignition.");

}

}

// Another subclass

class Motorcycle extends Vehicle {

public Motorcycle() {

this.brand = "Honda";

}

// Implementation of the abstract method

@Override

void startEngine() {

System.out.println("Motorcycle started with a kick-starter.");

}

}

public class AbstractionDemo {

public static void main(String[] args) {

// We can't create an object of Vehicle: Vehicle myVehicle = new Vehicle(); // Error!

// We create objects of the concrete subclasses

Car myCar = new Car();

Motorcycle myMotorcycle = new Motorcycle();

// Calling the methods

myCar.startEngine(); // Hides the complexity of \*how\* the car starts

myCar.stop(); // Uses the concrete method from the abstract class

myMotorcycle.startEngine(); // Hides the complexity of \*how\* the motorcycle starts

myMotorcycle.stop();

}

}

**Output:**

Car started with a key ignition.

Toyota has stopped.

Motorcycle started with a kick-starter.

Honda has stopped.

**2. Abstraction with Interfaces**

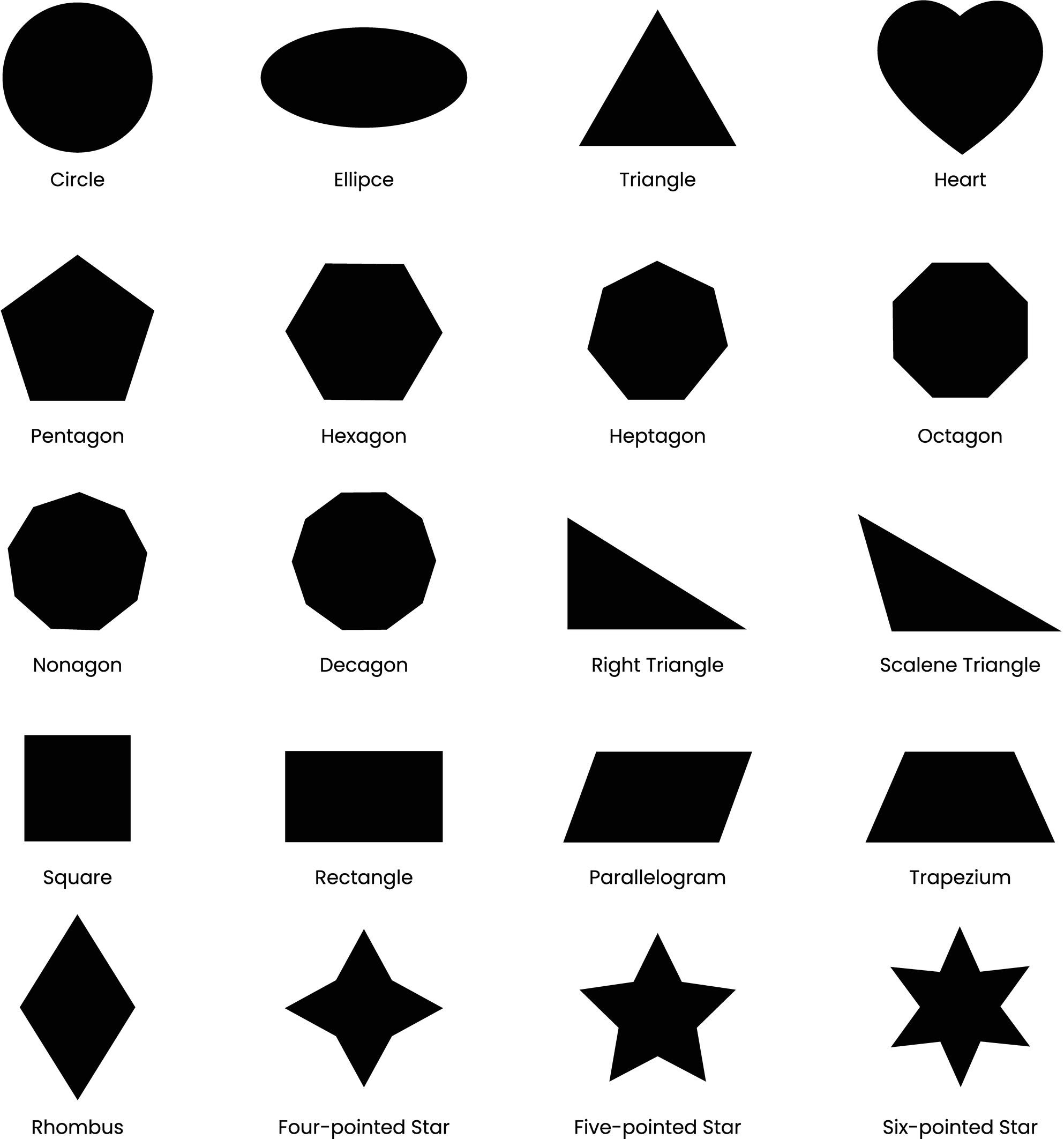
An **interface** is a blueprint of a class. It contains only abstract methods and constants (since Java 8, it can also have default and static methods). Interfaces achieve **100% abstraction** from a pure method-definition perspective (prior to Java 8) and support **multiple inheritance** of type.

**Key Concepts:**

* **Contract:** An interface defines a contract. Any class that implements the interface must provide an implementation for all its abstract methods.
* **Multiple Inheritance:** A class can implement multiple interfaces, allowing it to inherit behavior from different sources.

**Example: The Drawable Interface**

Consider an interface for things that can be drawn



Shutterstock

.

Java

// Interface (Contract)

interface Drawable {

// Methods are implicitly public and abstract

void draw();

double calculateArea();

}

// Class implementing the interface

class Circle implements Drawable {

private double radius = 5.0;

@Override

public void draw() {

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

}

@Override

public double calculateArea() {

return Math.PI \* radius \* radius;

}

}

// Another class implementing the interface

class Rectangle implements Drawable {

private double length = 4.0;

private double width = 6.0;

@Override

public void draw() {

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

}

@Override

public double calculateArea() {

return length \* width;

}

}

public class InterfaceDemo {

public static void main(String[] args) {

Drawable shape1 = new Circle();

Drawable shape2 = new Rectangle();

// The main method only interacts with the 'Drawable' contract (abstraction)

shape1.draw();

System.out.println("Area: " + shape1.calculateArea());

shape2.draw();

System.out.println("Area: " + shape2.calculateArea());

}

}

**Output:**

Drawing a Circle.

Area: 78.53981633974483

Drawing a Rectangle.

Area: 24.0

**Summary of Abstraction**

| **Feature** | **Abstract Class** | **Interface** |
| --- | --- | --- |
| **Keyword** | abstract class | interface |
| **Inheritance** | Uses extends | Uses implements |
| **Object Creation** | Cannot be instantiated | Cannot be instantiated |
| **Methods** | Can have abstract *and* concrete methods. | Methods are implicitly public abstract (before Java 8). |
| **Multiple Inheritance** | Only one abstract class can be extended. | Multiple interfaces can be implemented. |
| **Variables** | Can have instance (non-final) variables. | Variables are implicitly public static final. |
| **Abstraction Level** | 0% to 100% | 100% (of methods, before Java 8). |

**Abstraction helps manage complexity** by letting you focus on the interface (what is available) without worrying about the implementation (how it works).

what is the mean of cannot create objects of an abstract class

That's a fantastic question that gets right to the heart of what an abstract class is for!

Here is the simple meaning of "You cannot create objects of an abstract class":

### 1. The Simple Meaning: No Physical Instance

When a class is declared as abstract, it means it is **incomplete** or **too general** to exist on its own as a physical thing.

* **You can't use the new keyword** with an abstract class.

#### Example: The Animal Blueprint

Imagine you have an abstract class named Animal.

Java

abstract class Animal {

// ... some methods

}

If you try to create an object:

Java

Animal myAnimal = new Animal(); // 💥 ERROR!

This fails because an "Animal" isn't a specific, runnable creature. What sound does a generic "Animal" make? How does it move? You can't say!

The class is meant only to serve as a **template** or **blueprint** for more specific, concrete things, like a Dog or a Cat.

### 2. Why is it Forbidden? (The Core Reason)

The main reason you can't instantiate an abstract class is because it might contain **abstract methods**—methods without a body.

If you could create an object of the Animal class, and that class had an abstract method called makeSound(), what would happen if you tried to call it?

Java

// If this were allowed...

Animal myAnimal = new Animal();

// ... and the Animal class had an abstract makeSound() method:

myAnimal.makeSound(); // ??? What sound would it make? Nothing is defined!

The Java compiler prevents this logical error by ensuring that **no object can exist until all its methods have a full, concrete implementation.**

### 3. What You *Can* Do: Inheritance

Instead of creating an object of the abstract class, you must create objects of its **concrete subclasses**.

You must create a class (like Dog) that **extends** the Animal class and **provides the missing implementations** for all the abstract methods.

Java

// Concrete Subclass

class Dog extends Animal {

// Provides the required implementation for makeSound()

void makeSound() {

System.out.println("Woof!");

}

}

public class Main {

public static void main(String[] args) {

// This is OK! We are creating an object of the concrete class.

Dog myDog = new Dog();

myDog.makeSound(); // Output: Woof!

}

}

**In short:** An abstract class is an incomplete idea. You can't build a functional object out of an incomplete idea. You must finish the idea first (by extending it) before you can use it.