The main difference is that the **JVM (Java Virtual Machine)** is the **engine** that runs your Java code, while the **JRE (Java Runtime Environment)** is the **entire environment** that contains the JVM and all the necessary libraries to run a Java application.

JVM (Java Virtual Machine) ⚙️

The JVM is the core component that enables Java's "write once, run anywhere" philosophy. It's a virtual machine that executes Java bytecode, which is the compiled output of Java source code. The JVM acts as an interpreter, translating the bytecode into the native machine code for the specific operating system and hardware it's running on.

- **Function:** Executes Java bytecode.
- **Role:** An abstract machine that provides a runtime environment for bytecode.
- **Key Feature:** Platform independence. A JVM implementation exists for every major operating system (Windows, macOS, Linux), ensuring that the same .class file can run on any of them.

JRE (Java Runtime Environment) 📦

The JRE is a complete package required to run Java programs. It includes everything a user needs to execute a Java application, but it does **not** contain tools for development.

- **Function:** Provides the runtime environment for Java applications.
- **Components:**
  - **JVM:** The engine that runs the code.
  - **Java Class Libraries:** A set of standard libraries (e.g., java.lang, java.util) that provide essential functionality like I/O, networking, and data structures.

- **Role:** The JRE is what end-users typically install on their computers to run Java applications. It is the minimal set of files required to run a program, but you can't use it to compile code.

Summary: JVM vs. JRE vs. JDK

Think of it like this:

- **JVM:** The engine in a car. It makes the car run.
- **JRE:** The entire car, but without a mechanic's tool kit. You can drive it, but you can't fix or modify it. It includes the engine, wheels, and all other necessary parts.
- **JDK (Java Development Kit):** The car, plus the mechanic's tool kit. It includes the JRE and all the development tools like a compiler (javac), debugger, and more. The JDK is for developers.

The JRE is a superset of the JVM. The JDK is a superset of the JRE.

Polymorphism, one of the three core pillars of object-oriented programming (OOP), is the ability of an object to take on many forms. 🎭 In Java, it allows you to treat objects of different subclasses in a uniform way through a common superclass or interface. This enables you to write code that's more flexible, reusable, and easier to maintain.

There are two main types of polymorphism in Java:

1. Static (Compile-Time) Polymorphism

This type of polymorphism is resolved at compile time. It is achieved through **method overloading**, which means having multiple methods with the same name in the same class, but with different parameters (number, type, or order of parameters). The compiler decides which method to call based on the arguments provided during the call.

Example:

Java

```java
class Calculator {
    // Overloaded method to add two integers
    public int add(int a, int b) {
        return a + b;
    }

    // Overloaded method to add two doubles
    public double add(double a, double b) {
        return a + b;
    }
}
```

In this example, the add method is overloaded. The compiler can tell the difference and choose the correct version based on whether you pass integers or doubles.

---

2. Dynamic (Runtime) Polymorphism

This is the more common type of polymorphism in OOP and is resolved at runtime. It is achieved through **method overriding**, where a subclass provides its own specific implementation of a method that is already defined in its superclass. The decision of which method to execute is made at runtime based on the actual object type, not the reference type.

Example:

Java

```
class Animal {
    public void makeSound() {
        System.out.println("The animal makes a sound.");
    }
}

class Dog extends Animal {
    // Overriding the makeSound method
    @Override
    public void makeSound() {
        System.out.println("The dog barks.");
    }
}

class Cat extends Animal {
    // Overriding the makeSound method
    @Override
    public void makeSound() {
        System.out.println("The cat meows.");
    }
}
```

In the code above, the makeSound() method is overridden in the Dog and Cat subclasses. You can then use a single reference type, Animal, to call the correct makeSound() method for each specific object.

Java

```
Animal myDog = new Dog();
Animal myCat = new Cat();


myDog.makeSound(); // Outputs: The dog barks.
myCat.makeSound(); // Outputs: The cat meows.
```

Even though both myDog and myCat are declared as type Animal, the Java Virtual Machine (JVM) knows at runtime that myDog is actually a Dog object and myCat is a Cat object, and it calls the correct overridden method accordingly.

Interfaces and abstract classes are both used to achieve abstraction in Java, but they serve different purposes and are used in different scenarios. The choice between them depends on the specific design requirements of your program.

When to use an Interface 🤝

Use an interface when you need to define a **contract** for a set of behaviors that multiple, unrelated classes will implement. Interfaces are ideal for creating a common standard for different classes that do not share a parent-child relationship.

- **To model capabilities:** An interface defines what a class **can do**. For example, a Flyable interface could be implemented by both an Airplane and a Bird class, even though they have no common superclass other than Object.
- **To support multiple inheritance of types:** A class can implement multiple interfaces, allowing it to conform to different contracts. This provides a way to achieve a form of multiple inheritance, which is not directly supported for classes in Java.
- **To decouple components:** Interfaces allow for loose coupling. You can write code that works with a generic interface type, and at runtime, it will work with any class that implements that interface. This makes the code more flexible and easier to test.

When to use an Abstract Class 🏛️

Use an abstract class when you want to create a common base for a group of related classes. Abstract classes are best for creating a blueprint for a family of objects that share a similar state (fields) and behavior, some of which are already implemented.

- **To share code among subclasses:** An abstract class can provide a default implementation for methods, which its subclasses can inherit and use directly. This avoids code duplication. For example, an Animal abstract class could have a concrete eat() method that all subclasses can use, but an abstract makeSound() method that each subclass must implement.

- **To provide a base with common state:** Abstract classes can have instance variables (fields), which are shared by all subclasses. Interfaces cannot have instance fields.
- **To tightly couple related classes:** The classes that extend an abstract class are part of a rigid hierarchy. This is best for a set of objects that are variations of a single concept, like Car, Motorcycle, and Truck all extending an AbstractVehicle class.

Summary Table

Feature	Interface	Abstract Class
Methods	All methods are abstract by default (until Java 8). Can also have default and static methods.	Can have a mix of abstract and concrete (implemented) methods.
Variables	Can only have public static final constants.	Can have any type of variable (instance, static, final).
Inheritance	A class can **implement** multiple interfaces.	A class can only **extend** one abstract class.
Constructor	Cannot have a constructor.	Can have a constructor.
Primary Use	Defining a contract or capability.	Providing a common base with shared code and state.

A **lambda expression** is a concise, anonymous function that provides a concrete implementation of a method. A **function** (specifically in the context of functional programming in Java) refers to the abstract concept of a method that takes input and produces output, and in Java, it is often represented by a **functional interface**.

The distinction is that a **lambda expression is the code block**, while a **Function is a functional interface** that serves as the type for that code block.