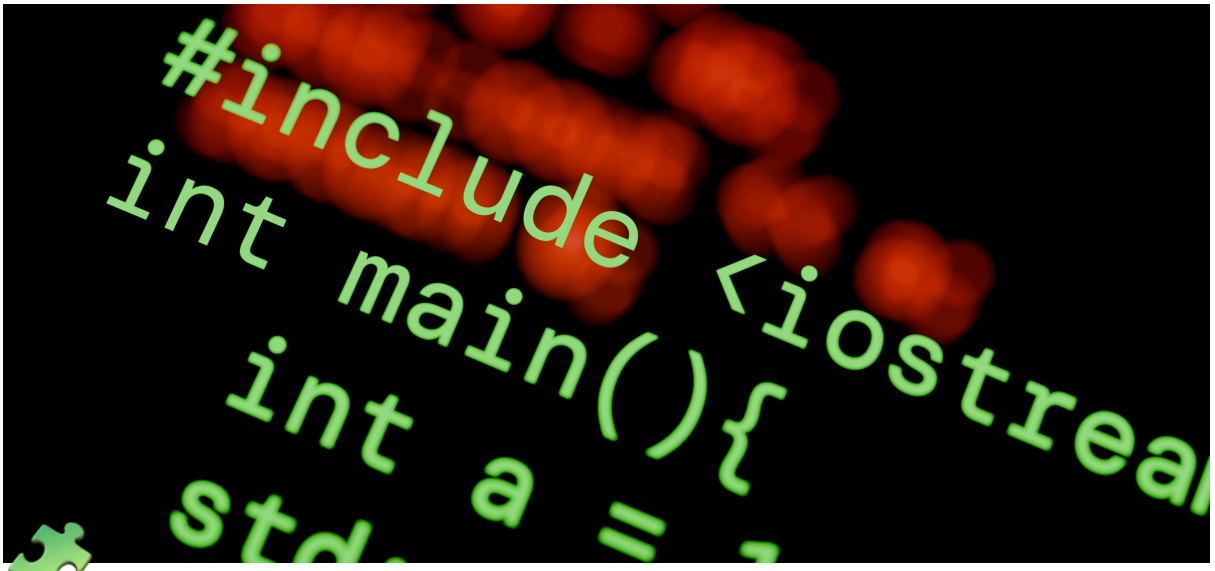

Variable + Polymorphism + Abstraction 24 Jan

Variable

In Java, a **variable** is a container that holds data values during program execution. Every variable has a **type** (e.g., `int`, `String`) that defines the kind of data it can store.

Types of Variables in Java

Java categorizes variables based on where they are declared and how they are used. Here's an overview:

---

1. Static Variables (Class Variables)

- Declared with the `static` keyword.
- Belong to the **class** rather than any specific object.
- Shared across all objects of the class.
- Initialized only once when the class is loaded into memory.

Example:

```
public class Example {
    static int staticVariable = 10; // Static variable

    public static void main(String[] args) {
        System.out.println("Static Variable: " + staticVariable);
    }
}
```

Key Points:

- Shared among all instances.
- Can be accessed using the class name (`ClassName.staticVariable`).
- Lifetime: Throughout the program execution.

---

2. Instance Variables

- Declared inside a class but **outside any method**, **constructor**, or **block**.
- Belong to a specific instance (object) of the class.
- Each object has its **own copy** of instance variables.

Example:

```
public class Example {
    int instanceVariable = 20; // Instance variable

    public static void main(String[] args) {
        Example obj = new Example();
        System.out.println("Instance Variable: " + obj.instanceVariable);
    }
}
```

Key Points:

- Initialized when the object is created.
- Accessed using the object reference (`obj.instanceVariable`).
- Lifetime: As long as the object exists.

---

3. Local Variables

- Declared **inside a method**, **constructor**, or **block**.
- Only accessible within the scope where they are declared.
- Must be initialized before use (no default value).

Example:

```
public class Example {
    public static void main(String[] args) {
        int localVariable = 30; // Local variable
        System.out.println("Local Variable: " + localVariable);
    }
}
```

Key Points:

- Limited to the block in which they're declared.
- No default values; you must initialize them explicitly.
- Lifetime: Until the block of code finishes executing.

Comparison Table

Feature	Static Variable	Instance Variable	Local Variable
Scope	**Class-level**	**Object-level**	**Method/block-level**
Lifetime	Program execution	Object's lifetime	Block/method execution
Default Value	Yes	Yes	No (must initialize)
Access Modifier	Can use access modifiers	Can use access modifiers	No access modifiers
Keyword	`static`	None	None
Shared	Yes (shared across objects)	No (unique for each object)	No

```
public class Example {
    static int staticVariable = 100;

    public static void main(String[] args) {
        // Accessing static variable via an object (not recommended)
        Example obj = new Example();
        System.out.println("Access via Object: " + obj.staticVariable);

        // Accessing static variable via class name (recommended)
        System.out.println("Access via Class Name: " + Example.staticVariable);
    }
}
```

Key Notes:

- Use **static variables** for shared data (e.g., constants, counters).
- Use **instance variables** for data unique to each object.
- Use **local variables** for temporary data within methods.

> The `final` keyword is useful when you want a variable to always store the same value, like PI (3.14159...).
>
> The `final` keyword is called a "modifier". You will learn more about these in the Java Modifiers Chapter.

What is Polymorphism?

Polymorphism in Java is the ability of an object to take many forms. It allows the same method or operation to behave differently depending on the object or data it is applied to.

Types of Polymorphism in Java

There are two main types:

1. **Compile-time Polymorphism (Method Overloading):**
    - Achieved by defining multiple methods with the same name but different parameters (number or type).

- Resolved at compile time.

2. **Runtime Polymorphism (Method Overriding):**
   - Achieved when a subclass provides a specific implementation of a method that is already defined in its parent class.
   - Resolved at runtime through dynamic method dispatch.

---

1. Compile-time Polymorphism (Method Overloading)

This is when multiple methods in the same class share the same name but differ in:

- Number of parameters
- Type of parameters

Example:

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

    // Method to add three integers
    public int add(int a, int b, int c) {
        return a + b + c;
    }

    // Method to add two floating-point numbers
    public float add(float a, float b) {
        return a + b;
    }
}

public class Main {
    public static void main(String[] args) {
        Calculator calc = new Calculator();

        // Calling overloaded methods
        System.out.println("Sum of two integers: " + calc.add(5, 10));      // Calls add(int, int)
        System.out.println("Sum of three integers: " + calc.add(1, 2, 3)); // Calls add(int, int, int)
        System.out.println("Sum of floats: " + calc.add(2.5f, 3.5f));      // Calls add(float, float)
    }
}
```

Output:

```
Sum of two integers: 15
Sum of three integers: 6
Sum of floats: 6.0
```

---

2. Runtime Polymorphism (Method Overriding)

This occurs when a subclass provides its own implementation of a method defined in the parent class. The method in the subclass overrides the one in the parent.

Key Points:

- Requires inheritance.
- Uses the concept of dynamic method dispatch.
- The overridden method in the subclass is called based on the actual object, not the reference type.

Example:

```
class Animal {
    public void sound() {
        System.out.println("Animals make different sounds");
    }
}

class Dog extends Animal {
    @Override
    public void sound() {
        System.out.println("Dog barks");
    }
}

class Cat extends Animal {
    @Override
    public void sound() {
        System.out.println("Cat meows");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal myAnimal;

        myAnimal = new Dog();
        myAnimal.sound(); // Calls Dog's sound()

        myAnimal = new Cat();
        myAnimal.sound(); // Calls Cat's sound()
    }
}
```

Output:

```
Dog barks
Cat meows
```

Differences Between Overloading and Overriding

Feature	Method Overloading	Method Overriding

Definition	Same method name, different parameter list.	Same method name and signature in parent and child class.
Type	Compile-time polymorphism	Runtime polymorphism
Inheritance	Not required.	Requires inheritance.
Binding	Resolved at compile time (static binding).	Resolved at runtime (dynamic binding).

Polymorphism with Interfaces

Polymorphism also works with interfaces. A reference variable of an interface can refer to any of its implementing classes.

Example:

```
interface Shape {
    void draw();
}

class Circle implements Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a Circle");
    }
}

class Rectangle implements Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a Rectangle");
    }
}

public class Main {
    public static void main(String[] args) {
        Shape shape;

        shape = new Circle();
        shape.draw(); // Calls Circle's draw()

        shape = new Rectangle();
        shape.draw(); // Calls Rectangle's draw()
    }
}
```

Output:

```
Drawing a Circle
Drawing a Rectangle
```

Key Benefits of Polymorphism

1. **Code Reusability**: Methods with the same name can be reused with different parameters or classes.

2. **Flexibility**: Objects can take many forms, allowing dynamic behavior.
3. **Readability**: Polymorphism makes code more readable and organized.

Practice Exercise

Write a Java program demonstrating the following:

1. Compile-time polymorphism using method overloading for a `Calculator` class.
2. Runtime polymorphism by overriding a method in a parent and child class (`Vehicle` → `Car`, `Bike`).

Abstraction in Java

Theory and Code: Demonstrating Abstraction in Java

This code demonstrates the concept of abstraction and how it differs when using an abstract class reference versus directly using a concrete class reference.

Key Points:

1. Abstract classes cannot be instantiated but can be used as references to subclass objects.
2. Using an abstract class reference ensures you can only call methods defined in the abstract class.
3. Using a concrete class reference allows access to both inherited methods and methods specific to the concrete class.

```
// Abstract class
abstract class Animal {
    // Abstract method (must be implemented by subclasses)
    abstract void makeSound();

    // Concrete method (shared by all subclasses)
    void breathe() {
        System.out.println("Breathing...");
    }
}

// Concrete class: Dog
class Dog extends Animal {
    // Implementation of abstract method
    @Override
    void makeSound() {
        System.out.println("Bark!");
    }

    // Subclass-specific method
    void wagTail() {
        System.out.println("Wagging tail...");
    }

    // Another implementation of abstract behavior
```

```
    @Override
    void move() {
        System.out.println("Runs on four legs.");
    }
}

// Concrete class: Bird
class Bird extends Animal {
    // Implementation of abstract method
    @Override
    void makeSound() {
        System.out.println("Chirp!");
    }

    // Subclass-specific method
    void fly() {
        System.out.println("Flying high in the sky...");
    }

    // Another implementation of abstract behavior
    @Override
    void move() {
        System.out.println("Flaps wings to move.");
    }
}

// Main class to demonstrate abstraction and concrete class behavior
public class AbstractionDemo {
    public static void main(String[] args) {
        // Using abstract class references
        Animal dog = new Dog();
        Animal bird = new Bird();

        // Demonstrating polymorphism
        System.out.println("Dog (Using Animal reference):");
        dog.breathe(); // Calls the concrete method from Animal
        dog.makeSound(); // Calls the Dog's implementation
        dog.move(); // Calls the Dog's implementation

        System.out.println();

        System.out.println("Bird (Using Animal reference):");
        bird.breathe(); // Calls the concrete method from Animal
        bird.makeSound(); // Calls the Bird's implementation
        bird.move(); // Calls the Bird's implementation

        System.out.println();

        // Using concrete class references
        Dog dog2 = new Dog();
        Bird bird2 = new Bird();
```

```
        System.out.println("Dog2 (Using Dog reference):");
        dog2.breathe(); // Calls the concrete method from Animal
        dog2.makeSound(); // Calls the Dog's implementation
        dog2.move(); // Calls the Dog's implementation
        dog2.wagTail(); // Specific to Dog class

        System.out.println();

        System.out.println("Bird2 (Using Bird reference):");
        bird2.breathe(); // Calls the concrete method from Animal
        bird2.makeSound(); // Calls the Bird's implementation
        bird2.move(); // Calls the Bird's implementation
        bird2.fly(); // Specific to Bird class
    }
}
```

Explanation:

Abstract Class (`Animal`):

- Contains an abstract method `makeSound()` that must be implemented by any subclass.
- Includes a concrete method `breathe()` that all subclasses inherit.

Concrete Class (`Dog` and `Bird`):

- `Dog` and `Bird` implement the `makeSound()` method and provide additional subclass-specific behaviors (`wagTail()` for `Dog` and `fly()` for `Bird`).

Main Class:

1. **Abstract Class Reference (`Animal dog = new Dog();`)**:
    - Demonstrates polymorphism.
    - Restricts access to methods defined in the `Animal` class.
2. **Concrete Class Reference (`Dog dog2 = new Dog();`)**:
    - Direct access to both inherited and subclass-specific methods.

Output:

The output will demonstrate the behaviors of `Dog` and `Bird` through both abstract and concrete references. This distinction is key to understanding abstraction and polymorphism in Java.