Team: Code Wizard
Members:

- Phanthira Kositjaroenkul (6630003)
- Kaung Khant Lin (6540131)
- Thit Lwin Win Thant (6540122)

Java supports all four types of polymorphism: **overloading**, **parameter coercion**, **parametric polymorphism**, and **subtype polymorphism**. Here's a critical breakdown of each.

1. Overloading (Ad Hoc Polymorphism)

Supported in Java: Yes

Mechanism: Java allows methods to be overloaded, meaning multiple methods can share the same name as long as their parameter lists (types, number, or order of parameters) are different. This form of polymorphism is resolved at compile time.

Example:

```java
class MathUtils {
    int square(int x) { return x * x; }
    double square(double x) { return x * x; }
}
```

Commentary: Overloading provides syntactic convenience, allowing developers to use a single, meaningful name for similar operations on different data types. However, Java does not support operator overloading (unlike C++), except for the `+` operator for string concatenation. Ambiguity can arise if the compiler cannot definitively choose a single overloaded method based on the provided arguments.

---

2. Parameter Coercion

Supported in Java: Yes, but limited

Mechanism: Java performs implicit type conversions, or **widening conversions**, automatically for primitive types. For instance, when a method expects a `double`, an `int` argument will be coerced to a `double` without an explicit cast.

Example:

```java
void printDouble(double x) {
    System.out.println(x);
}

printDouble(5);       // int 5 is coerced to double 5.0
```

Commentary: This feature adds flexibility, but it's restricted to widening conversions (e.g., `byte` to `int`, `int` to `double`). Java strictly avoids automatic **narrowing conversions** (e.g., `double` to `int`) to prevent data loss.

---

3. Parametric Polymorphism

Supported in Java: Yes (via Generics since Java 5)

Mechanism: Java's **generics** enable the creation of classes, interfaces, and methods that work with a wide range of types as parameters. This allows for type-safe, reusable code without being tied to a specific data type.

Example:

```java
class Box<T> {
    private T value;
    public void set(T v) { value = v; }
    public T get() { return value; }
}

Box<Integer> intBox = new Box<>();
Box<String> strBox = new Box<>();
```

Commentary: Java's generics are a powerful compile-time feature. However, they are implemented with **type erasure**, meaning the generic type information is removed at runtime. This prevents direct use with primitive types (e.g., `Box<int>` is invalid) and can lead to certain runtime type safety issues ("heap pollution").

---

4. Subtype Polymorphism (Inclusion Polymorphism)

Supported in Java: Yes, and it's a fundamental aspect of Java's object-oriented model.

Mechanism: This type of polymorphism allows a reference of a superclass or interface type to refer to an object of any of its subclasses. Method calls on this reference are resolved at runtime through **dynamic dispatch**, ensuring the correct method of the actual object is executed.

Example:

```java
class Animal {
    void speak() { System.out.println("Generic sound"); }
}
class Dog extends Animal {
    @Override
    void speak() { System.out.println("Woof"); }
}

Animal a = new Dog();
a.speak();  // Prints "Woof"
```

Commentary: This is a powerful form of polymorphism that supports the principle of "programming to an interface, not an implementation." It enables flexible and extensible designs, making it central to inheritance and interface-based programming in Java. Since any number of subtypes can be created, it is considered a universal polymorphism.