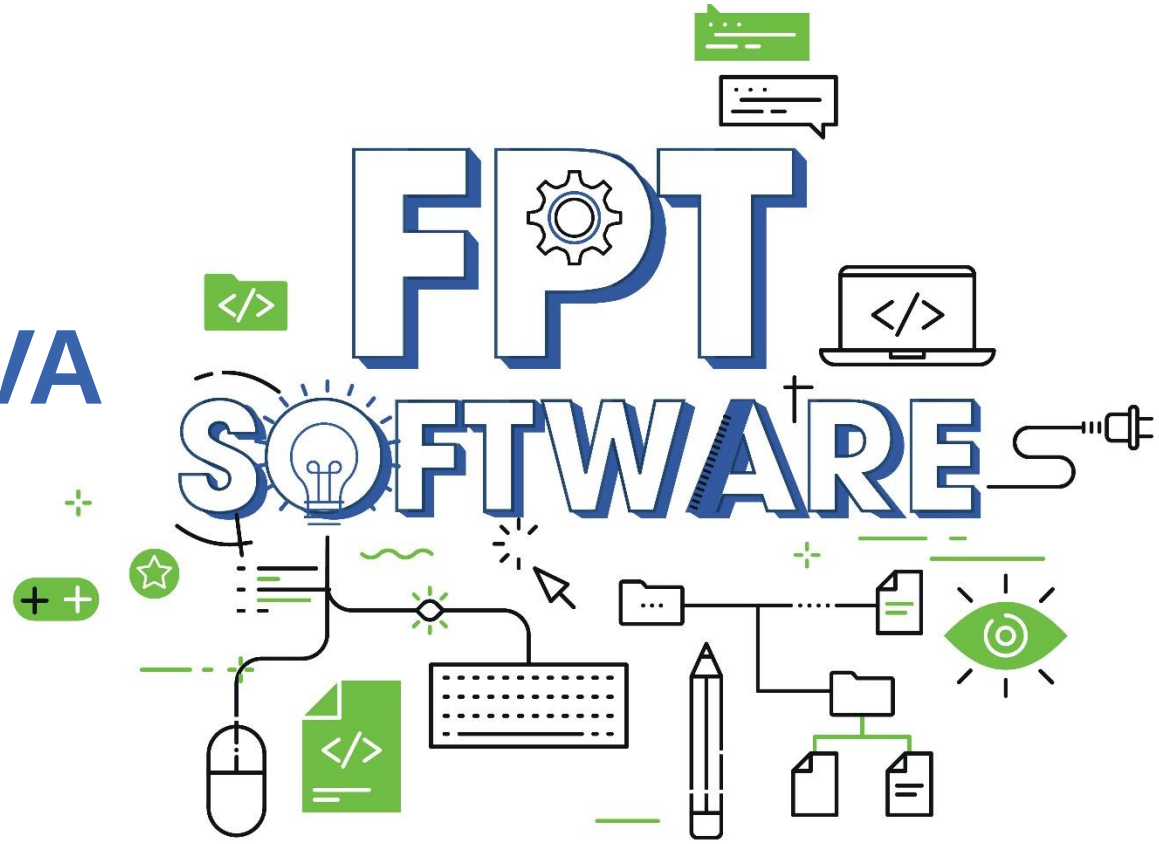
ADVANCE OOP WITH JAVA



Instructor: DieuNT1

Agenda

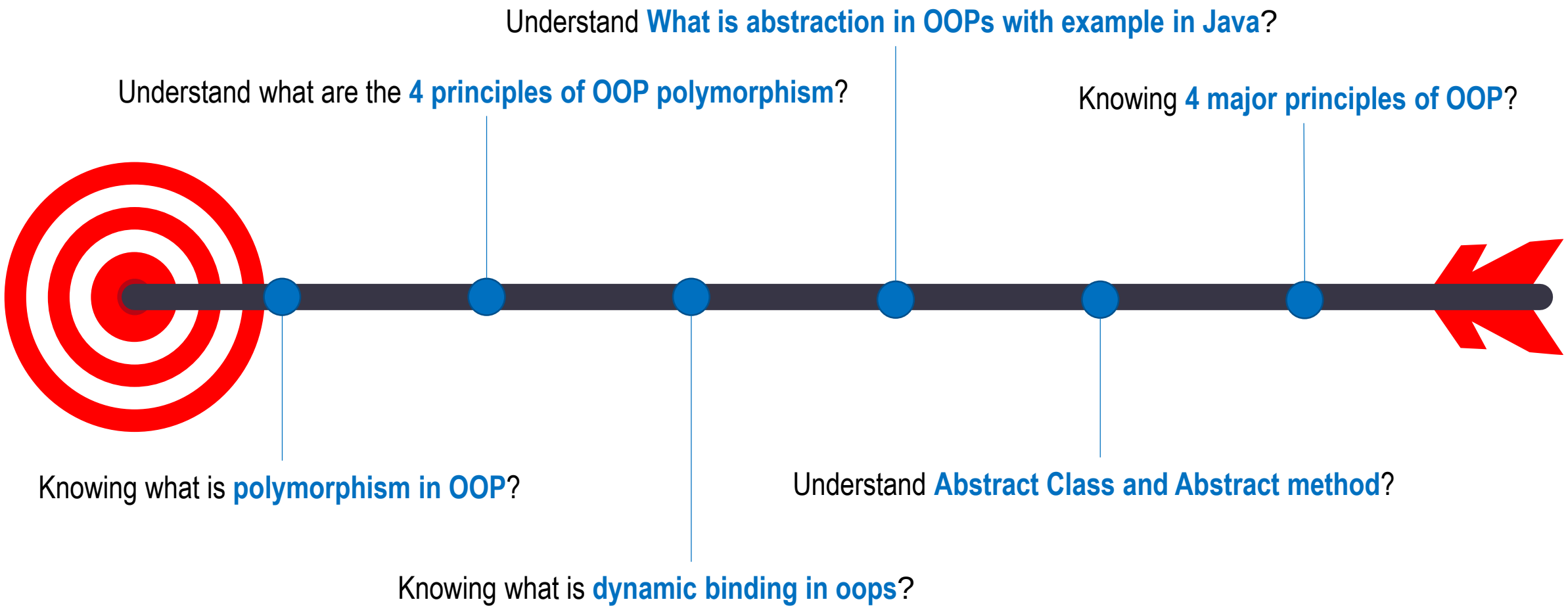
01. Polymorphism

- ✓ Types of Polymorphism
- ✓ Method Overloadding
- ✓ Method Overriding
- ✓ Static and Dynamic binding

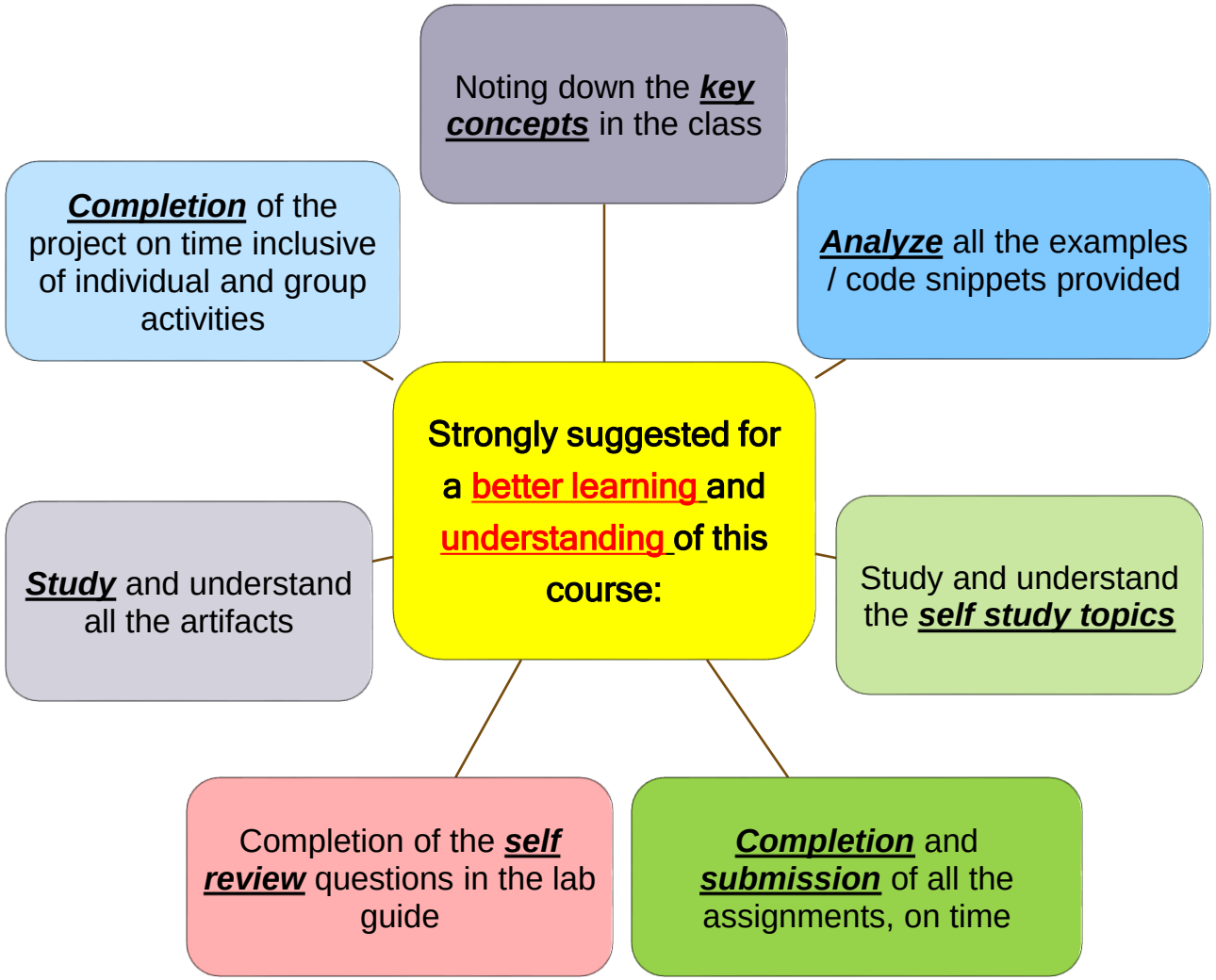
02. Abstraction

- ✓ Abstract class
- ✓ Interfaces

Lesson Objectives



Learning Approach



Section 1

POLYMORPHISM

Polymorphism Overview

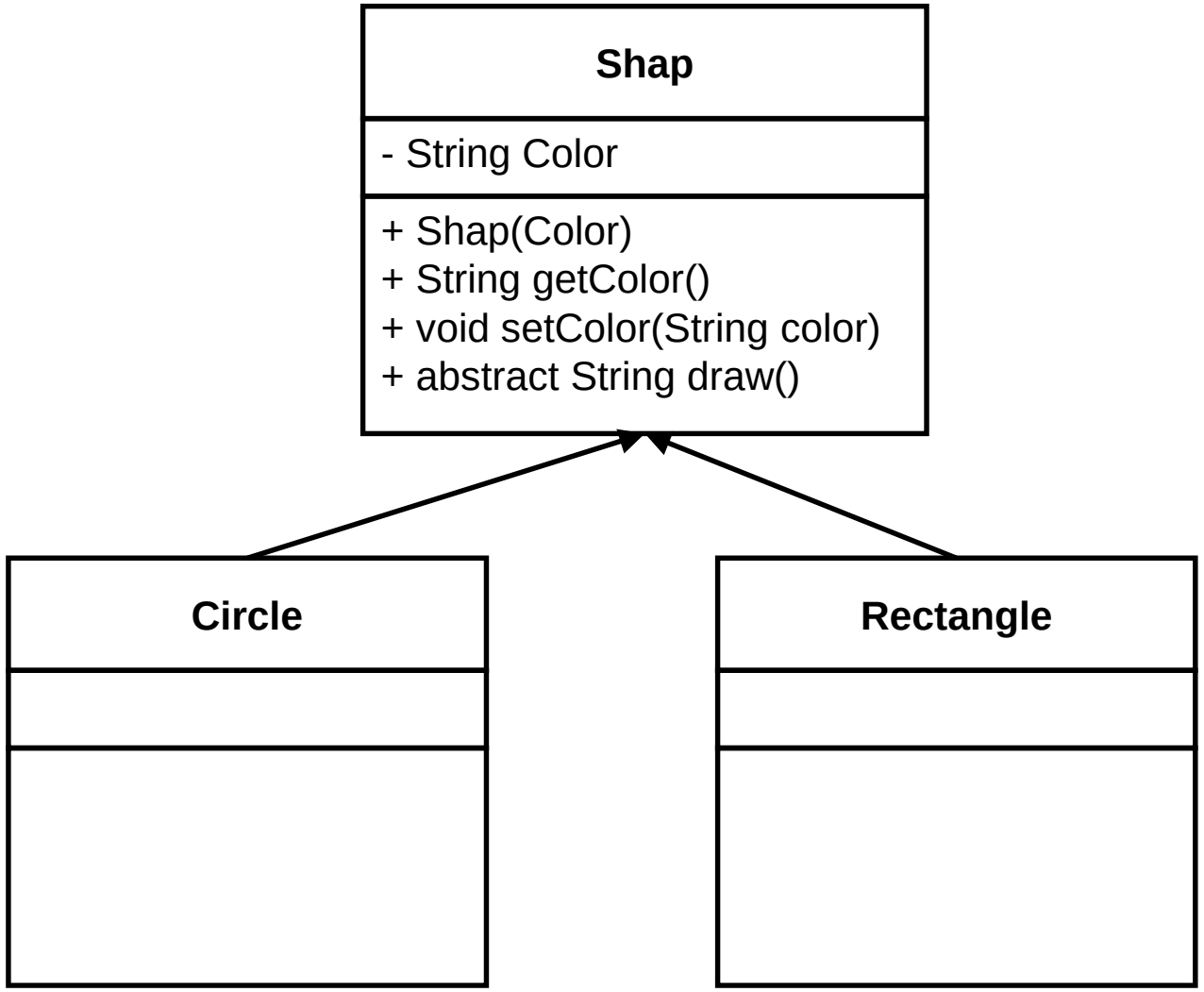
Polymorphism, meaning "many forms," is another key concept in Java that goes hand-in-hand with abstraction.

It allows you to *treat objects of different types in the same way*, *without needing to know their exact details*. This makes your code more flexible and powerful.

- There are two types of polymorphism in java:
  - ✓ compile time polymorphism: method overloading.
  - ✓ runtime polymorphism: method overriding.

one name, many forms.

Polymorphism Example



Polymorphism Example

```java
public class Shape {
    private String color;

    public Shape(String color) {
        this.setColor(color);
    }

    public String getColor() {
        return color;
    }

    public void setColor(String color) {
        this.color = color;
    }

    public void draw() {
     return "I'm a " + this.color + " shape.";
    }
}
```

```java
public class Circle extends Shape {
    public Circle(String color) {
        super(color);
    }
    @Override
    public String draw() {
        return "I'm a " + this.getColor() + " circle.";
    }
}
```

```java
public class Rectangle extends Shape {
    public Rectangle(String color) {
        super(color);
    }

    @Override
    public String draw() {
        return "I'm a " + this.getColor() + " rectangle.";
    }
}
```

Polymorphism Example

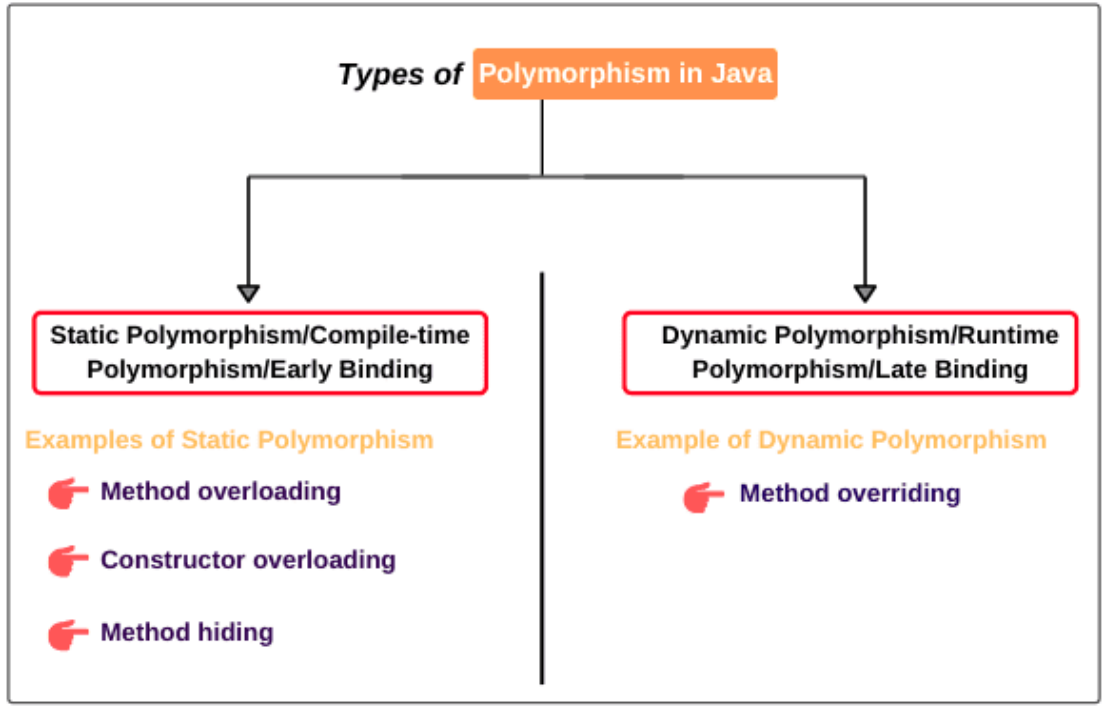
```
public class PolymorphismExample {
    private List<Shape> shapes = new ArrayList<Shape>();

    public PolymorphismExample() {

        Shape shape = new Shape("Yellow");
        Shape myFirstCircle = new Circle("Red");
        Circle mySecondCircle = new Circle("Blue");
        Rectangle myFirstRectangle = new Rectangle("Green");
        shapes.add(shape);
        shapes.add(myFirstCircle);
        shapes.add(mySecondCircle);
        shapes.add(myFirstRectangle);
    }
    public List<Shape> getShapes() {
         return shapes;
    }

    public static void main(String[] args) {
        PolymorphismExample example = new PolymorphismExample();
        for (Shape shape : example.getShapes()) {
             System.out.println(shape.draw());
        }
    }
}
```

Output:

```
I'm a Yellow shape.
I'm a Red circle.
I'm a Blue circle.
I'm a Green rectangle.
```

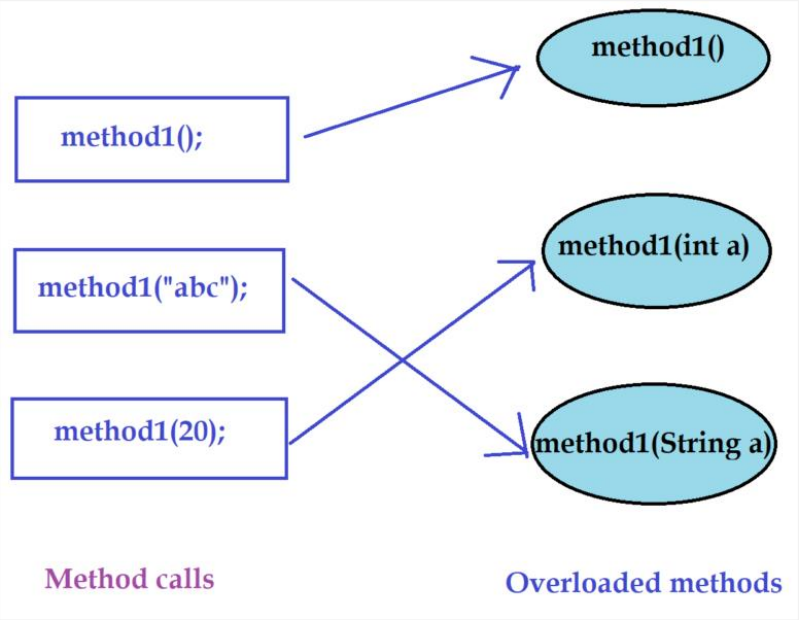
Types of Polymorphism

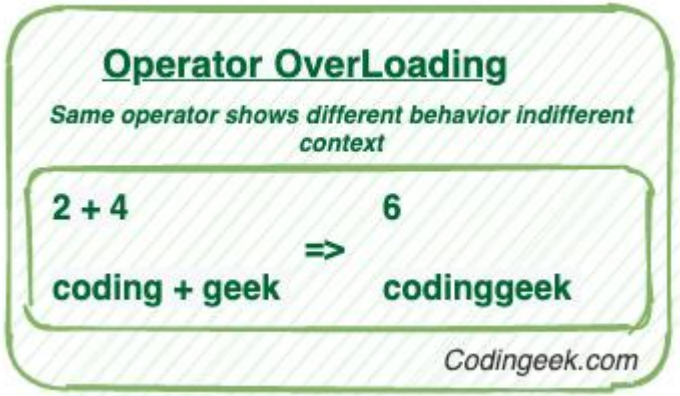
- **There are two types of polymorphism in Java**:
  - ✓**Static Polymorphism** also known as **compile time** polymorphism
  - ✓**Dynamic Polymorphism** also known as **runtime** polymorphism



Types of Polymorphism

- **Compile time Polymorphism** (or Static polymorphism)
  - ✓ Polymorphism that is resolved during compiler time is known as static polymorphism.
  - ✓ *Java compiler* binds method calls with method definition/body during compilation. Therefore, this type of polymorphism is also called **compile-time polymorphism in Java**.
  - ✓ **Method overloading and Operator Overloading** are examples of compile time polymorphism

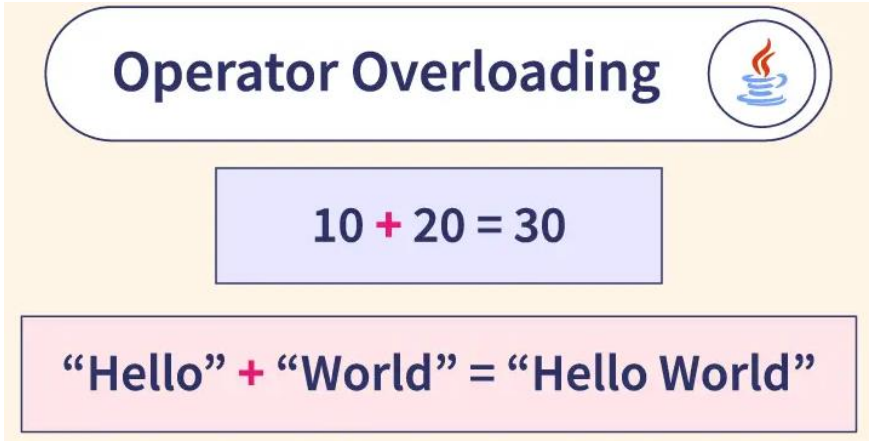




Operator Overloading

Operator overloading is the process of overloading an operator to perform different functions.

- Operator overloading refers to the ability to redefine the meaning of existing operators (like +, -, *, etc.) for custom data types, depending on the operands involved.
- Actually, **Java doesn't natively support operator overloading** in the classical sense: only the plus operator is allowed to have different functions.
  - ✓ Some operators, like "+" for String concatenation, already have built-in behavior for specific data types.



Operator Overloading

- However, Java takes a different approach to achieving similar functionality through. For example:

```
public class OperatorOverload {
	public void plusOperator(int num1, int num2) {
		System.out.println("The plus operator can add two integers! " + (num1 + num2));
	}

	public void plusOperator(String str1, String str2) {
		System.out.println("The plus operator can also concatenate two strings! " + (str1 + str2));
	}

	public static void main(String[] args) {
		OperatorOverload operator = new OperatorOverload();
		String str1 = "Data", str2 = "Flair";
		int num1 = 10, num2 = 14;
		operator.plusOperator(str1, str2);
		operator.plusOperator(num1, num2);
	}
}
```

- **Output**:

```
The plus operator can also concatenate two strings! DataFlair
The plus operator can add two integers! 24
```

Method Overloading

Overloaded methods let you reuse the same method name in a class, but with different arguments (and optionally, a different return type).

- **There are two ways to overload the method in java:**
  - ✓ By changing number of arguments
  - ✓ By changing the data type
- **Example**:

```
void func() { ... }
void func(int a) { ... }
float func(double a) { ... }
float func(int a, float b) { ... }
add(int a, int b) and add(double x, double y) for adding different data types.
print(String message) and print(Object obj) for printing different types of data.
sort(int[] arr) and sort(List<String> list) for sorting different data structures.
```

Method Overloading

- **Example:**

```
public class Shapes {
     public void area() {
          System.out.println("Find area ");
     }

     public void area(int r) {
          System.out.println("Circle area = " + 3.14 * r * r);
     }

     public void area(double b, double h) {
          System.out.println("Triangle area=" + 0.5 * b * h);
     }

     public void area(int l, int b) {
          System.out.println("Rectangle area=" + l * b);
     }
}
```

```
public class Main {
     public static void main(String[] args) {
          // Create a Shapes object
          Shapes myShape = new Shapes();
          myShape.area();
          myShape.area(5);
          myShape.area(6.0, 1.2);
          myShape.area(6, 2);
     }
}
```

- **Output:**

```
Find area
Circle area = 78.5
Triangle area=3.5999999999999996
Rectangle area=12
```

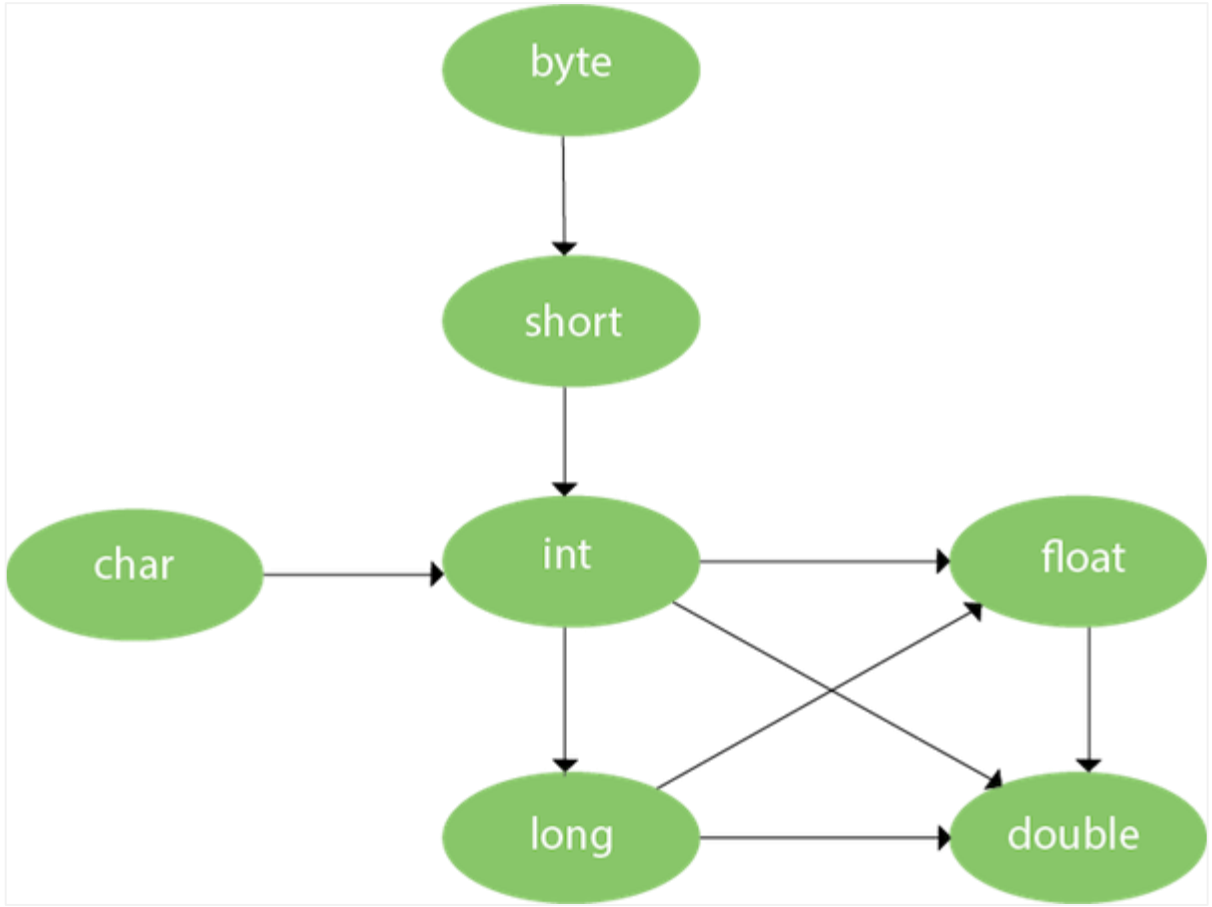
Method Overloading

- **Some main rules for overloading a method:**
  - ✓ Overloaded methods **must change the argument list**.
  - ✓ Overloaded methods **can change the return type**.
  - ✓ Overloaded methods **can change the access modifier**.
  - ✓ A method can be overloaded in the same class or in a subclass.
- **In a subclass,**
  - ✓ you can overload the methods inherited from the superclass.
  - ✓ such overloaded methods neither hide nor override the superclass methods, they are new methods, unique to the subclass.

Note: When I say method signature I am not talking about return type of the method, for example if two methods have same name, same parameters and have different return type, then this is not a valid method overloading example. This will throw compilation error.

Method Overloading and Type Promotion

- One type is promoted to another implicitly if no matching datatype is found:

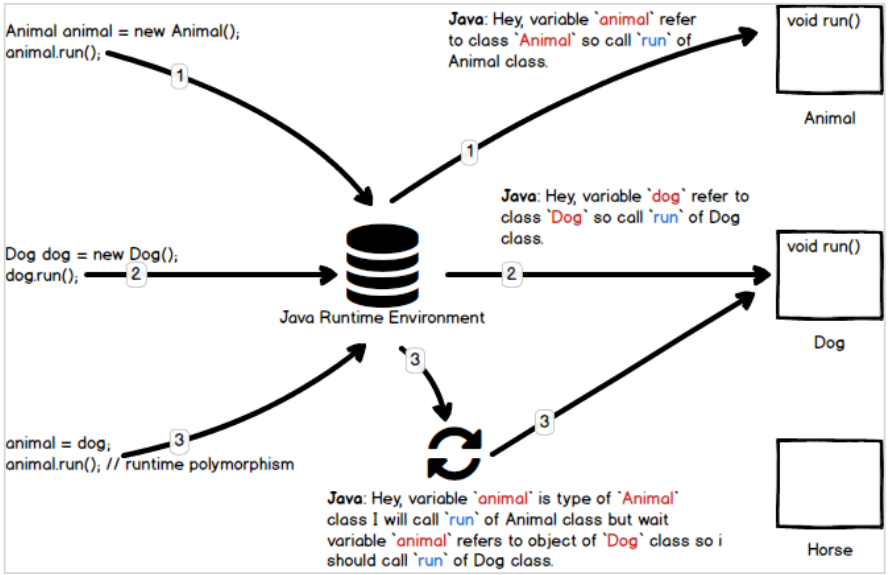


Practice time

- A program calculates and displays bonus amounts to pay various types of employees. There are 3 separate departments, numbered 1, 2, and 3.
  - ✓ ***Department 1*** employees are paid a bonus based on their sales: If their sales amount is over $5000 they get 5% of those sales, otherwise they get nothing.
  - ✓ ***Department 2*** employees are paid a bonus based on the number of units they sell: They get $100 per unit sold, and an extra $50 per unit if they sell more than 25 units; if they sell no units, they get nothing.
  - ✓ ***Department 3*** employees assemble parts in the plant and are paid a bonus of 10 cents per part if they reach a certain level: Part-time employees must assemble more than 250 parts to get the 10-cent-per-part bonus, and full-time employees must assemble more than 700.
- Write a set of 3 overloaded methods called **getBonus()** that works with the program below, according to the specifications described above.

Types of Polymorphism

- **Runtime Polymorphism (**or **Dynamic polymorphism):**
  - ✓ In dynamic polymorphism, the behavior of a method is decided at runtime,
  - ✓ **The JVM** (Java Virtual Machine) **binds the method call with method definition/body at runtime** and invokes the relevant method during runtime when the method is called.
    - The **Java compiler has no awareness** of the method to be called on an instance during compilation.
  - ✓ Dynamic or runtime polymorphism can be achieved/implemented in Java using **Method Overriding**.



Method Overriding

- Declaring a method in **subclass** which is already present in **parent class** is known as method overriding.
- **The main advantage of method overriding is:**
  - ✓ the class can give its own specific implementation to a inherited method *without even modifying the parent class* (base class).
- **Rules of method overriding in Java:**
  - ✓ Method name: method must have the same name as in the parent class.
  - ✓ Argument list: must be same as that of the method in parent class,
  - ✓ Access Modifier: cannot be more restrictive than the overridden method of parent class.

Method Overriding/Dynamic Bidning Examples

```
class Mammal {
    String makeNoise() {
        return "generic noise";
    }
}
class Zebra extends Mammal {
    @Override
    String makeNoise() {
        return "bray";
    }
}
public class ZooKeeper {
    public static void main(String[] args) {
        Mammal m = new Zebra();
        System.out.println(m.makeNoise());
    }
}
```

When you call a method through a reference variable referring to a parent class type, but the actual object at runtime belongs to a subclass, dynamic binding kicks in.

Runtime Lookup: Java looks up the most specific implementation of the method available in the inheritance hierarchy of the actual object's type.

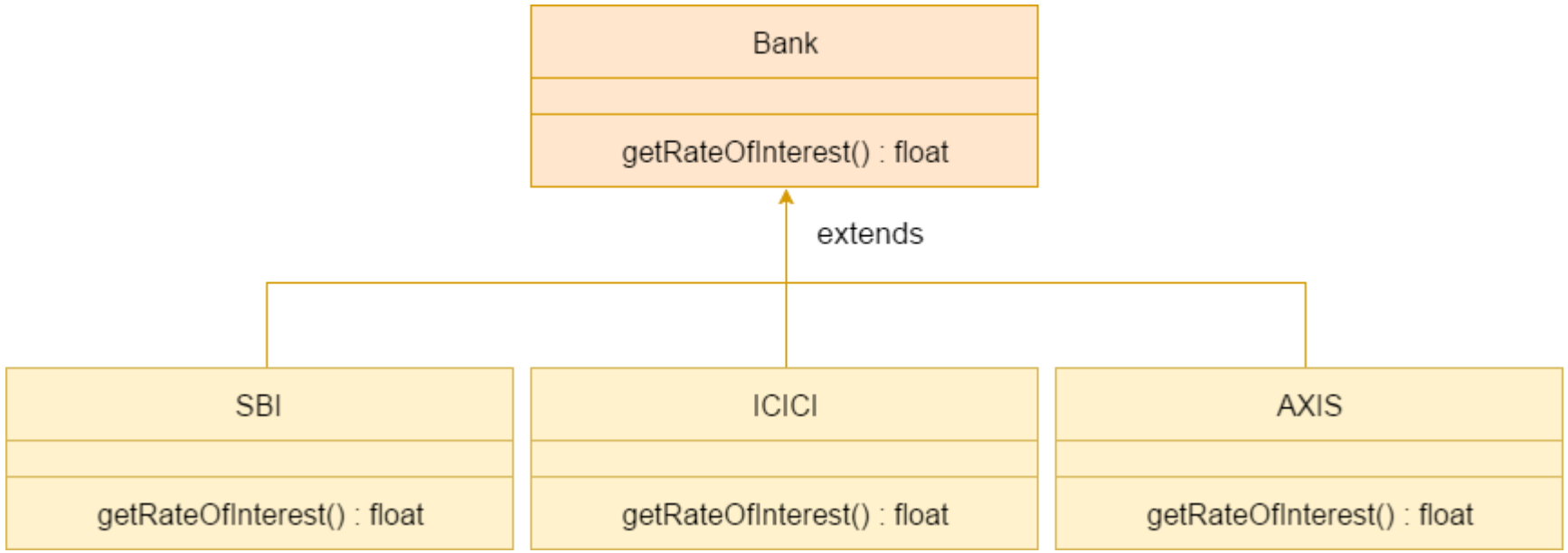
This "closest" override takes precedence and gets invoked at runtime.

Output:

```
bray
```

A real example of Method Overriding

- Consider a scenario where **Bank** is a class that provides functionality to get the rate of interest. However, the rate of interest varies according to banks.
- For example, SBI, ICICI and AXIS banks could provide 8%, 7%, and 9% rate of interest.



A real example of Method Overriding

```
class Bank {
   float getRateOfInterest() {
    return 0;
   }
}

class SBI extends Bank {
   float getRateOfInterest() {
    return 8.4f;
   }
}
class ICICI extends Bank {
   float getRateOfInterest() {
    return 7.3f;
   }
}
class AXIS extends Bank {
   float getRateOfInterest() {
    return 9.7f;
   }
}
```

```
public class TestPolymorphism {
     public static void main(String args[]) {
       Bank b;
       b = new SBI();
       System.out.println("SBI Rate of Interest: " +
                                          b.getRateOfInterest());

       b = new ICICI();
       System.out.println("ICICI Rate of Interest: " +
                                          b.getRateOfInterest());

       b = new AXIS();
       System.out.println("AXIS Rate of Interest: " +
                                          b.getRateOfInterest());
     }
}
```

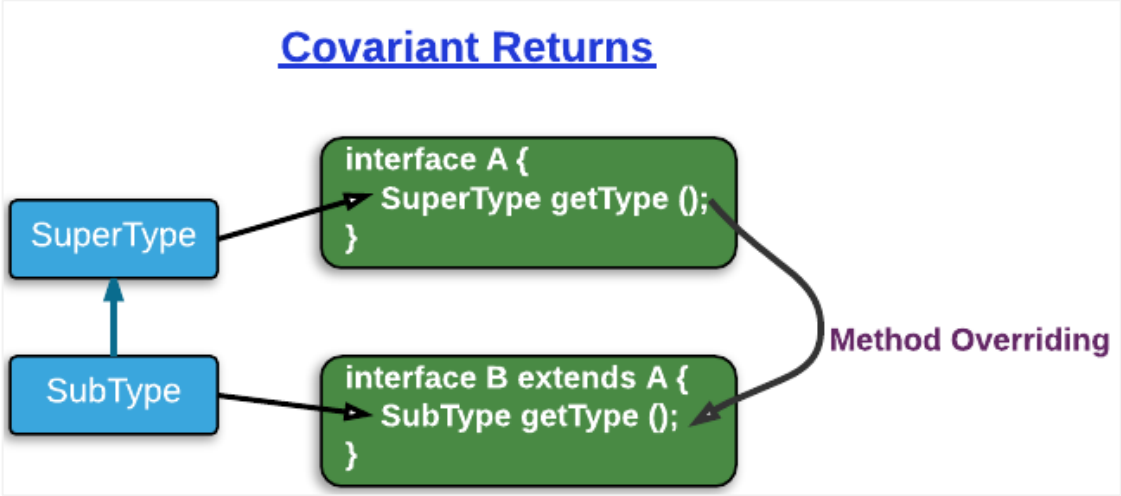
Output:

```
SBI Rate of Interest: 8.4
ICICI Rate of Interest: 7.3
AXIS Rate of Interest: 9.7
```

Method Overriding: Covariant Return Type

Covariant return types in method overriding allow a subclass to return a subtype of the return type declared in the parent class method.

- This provides greater flexibility and type safety when extending functionality in inheritance hierarchies.
- **When a subclass overrides a method from its parent class, it can:**
  - ✓ Have the **same return type** as the parent method (traditional overriding).
  - ✓ Have a **sub-type** of the parent method's return type (covariant overriding).
- This allows the subclass to return a **more specific version** of the data returned by the parent method.



Shadowing

- This is called shadowing—name in class Dog shadows name in class Animal

```
public class Animal {
    String name = "Animal";
     public void speak() {
        System.out.println("generic speak!");
     }
    public static void main(String args[]) {
           Animal animal1 = new Animal();
           Animal animal2 = new Dog();
           Dog dog = new Dog();
           System.out.println(animal1.name + " " + animal2.name +" " + dog.name);
    }
}
class Dog extends Animal {
    String name = "Dog";
     public void speak() {
        System.out.println("dog speak!");
     }
}
```

- **Output:**

```
Animal Animal Dog
```

Static and Dynamic binding

Static binding

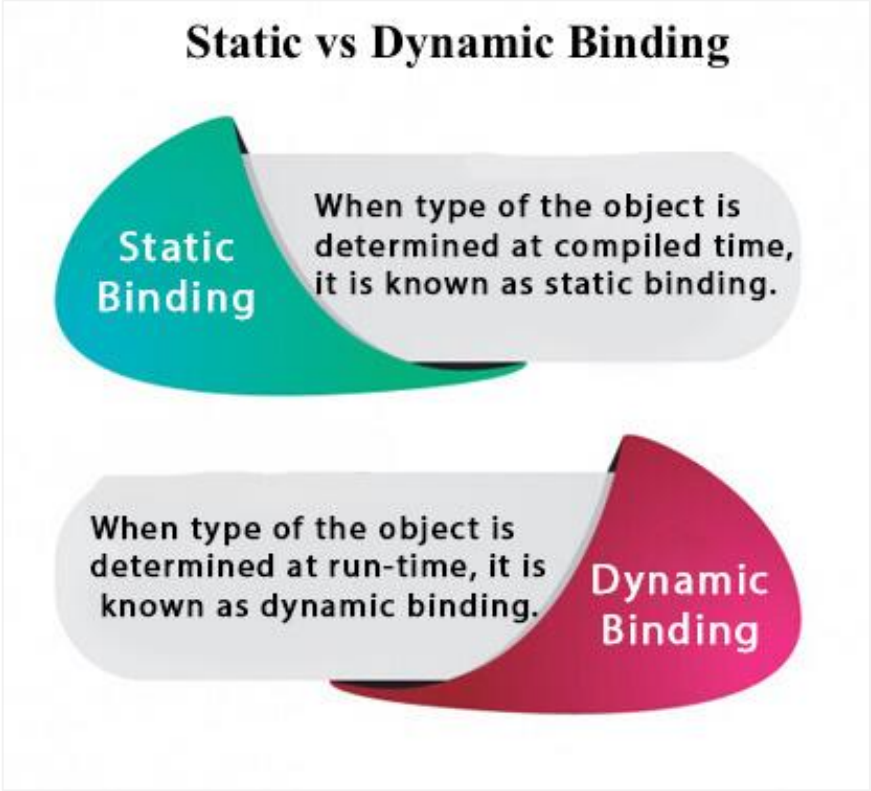
- The binding which can be resolved at compile time by compiler is known as static or early binding.
- The binding of static, private and final methods is compile-time. The reason is that the these method cannot be overridden and the type of the class is determined at the compile time.

Dynamic binding

- When compiler is not able to resolve the call/binding at compile time, such binding is known as Dynamic or late Binding.
- **Method Overriding** is a perfect example of dynamic binding as in overriding both parent and child classes have same method and in this case the **type of the object** determines which method is to be executed.
- The type of object is determined at the run time so this is known as dynamic binding.

Static and Dynamic binding

Static binding	Dynamic binding
Happens at Compile time	Happens at Runtime
Actual object is not used	Actual object is used
Also known as Early binding	Also known as Late binding
Speed is high	Speed is low
Ex: - Method Overloading	**Ex**: - Method Overriding



Static Method Overriding and Binding

- While **static methods in subclasses** technically "hide" the same-named methods from their parent class, it's not considered true hiding in the traditional sense.
- When calling a static method, **the compiler still performs static binding based on the type of the reference variable** (which is always the class name). So, it's not the object's type that determines the called method, but the static type of the reference used.
- **Example:**

```
class ParentClass {
  public static void staticMethod() {
     System.out.println("Parent class static method");
  }
}

class ChildClass extends ParentClass {
  public static void staticMethod() {
     System.out.println("Child class static method");
  }
}
```

Static Method Overriding: Static Binding

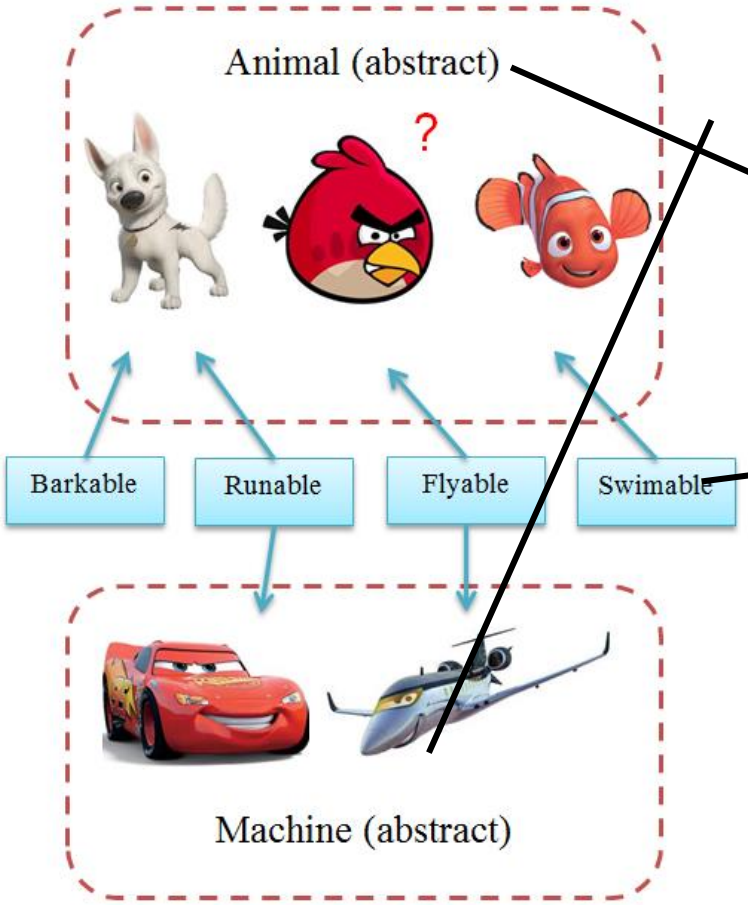
```
public class Main2 {
    public static void main(String[] args) {
        ParentClass.staticMethod(); // Prints "Parent class static method"
        ChildClass.staticMethod(); // Prints "Child class static method"
    }
}
```

Section 2

ABSTRACTION

Abstraction Overview

- **Abstraction** is a process of hiding the implementation details and showing only functionality to the user.



- **Example:**
  - ✓ Sending sms, you just type the text and send the message. You don't know the internal processing about the message delivery

It shows only important things to the user and hides the internal details

Focus on what the object does instead of how it does it.

- **There are two ways to achieve abstraction in java:**
  - ✓ Abstract class (0 to 100%)
  - ✓ Interface (100%)

Purposes of Abstraction

- **Hides implementation details**, focusing on what a class does (functionality) instead of how it does it (internal workings).
- Makes **code easier to understand** and **use**, reducing cognitive load for developers.
- **Changes in internal implementation can be made without affecting external behavior**, ensuring stability and reducing risk of breaking existing code.
- Makes fixing bugs and **adding new features easier** as modifications are localized within the abstracted class.
- *Subclasses can inherit and extend existing functionality*, **saving development time** and **effort**.
- **Abstract classes** and **interfaces** define contracts for behavior without specifying implementation details.

Abstract Class

- A class which is declared as abstract is known as an **abstract class**.
  - ✓ It can have abstract and non-abstract methods.
  - ✓ It needs to be extended and its method implemented.
  - ✓ It cannot be instantiated.
- **Rules for Java Abstract class:**

1. An abstract class must be declared with an abstract keyword.
2. It can have abstract and non-abstract methods.
3. It cannot be instantiated.
4. It can have constructors and static methods also.
5. It can have final methods which will force the subclass not to change the body of the method.

Abstract class and abstract method

- An **abstract class** can not be **instantiated** (you are not allowed to create **object** of Abstract class), but they can be subclassed.

```
Animal animal = new Cat();
```

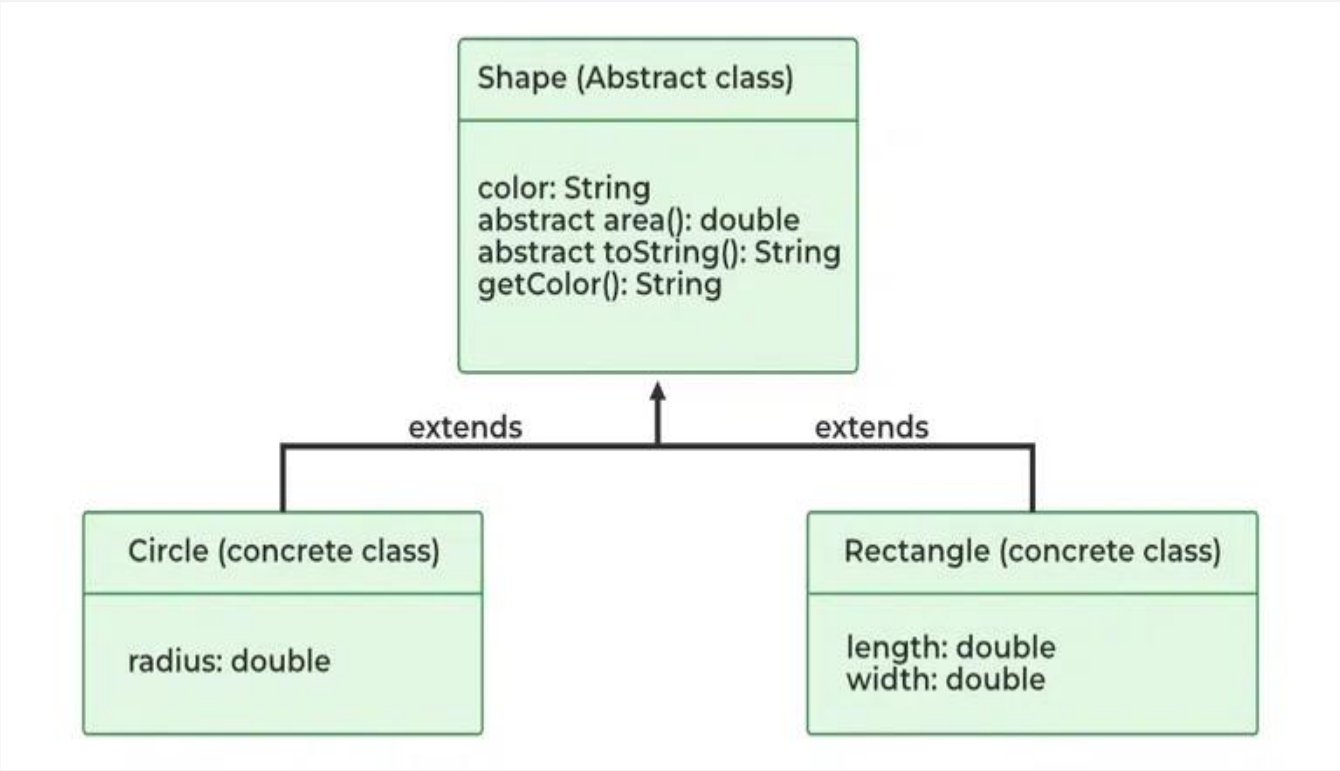
- A subclass usually provides implementations for all of the abstract methods in its parent class.
  - ✓ However, *if it does not*, the subclass must also be declared abstract.
- **Abstract methods:**
  - ✓ An abstract method is a method that is declared without an implementation. For Example:

```
abstract void moveTo(int x, int y);
```

  - ✓ If a class includes abstract methods, the class itself must be declared abstract.

Abstraction Example

- **Example:**



Abstraction Example

- **Example:**

```
public abstract class Shape {
   String color;

   // these are abstract methods
   abstract double area();

   // abstract class can have the constructor
   public Shape(String color) {
    System.out.println("Shape constructor called");
    this.color = color;
   }

   // this is a concrete method
   public String getColor() {
     return color;
   }
}
```

```
class Circle extends Shape {
  double radius;

  public Circle(String color, double radius) {
    // calling Shape constructor
    super(color);
    System.out.println("Circle constructor called");
    this.radius = radius;
  }

  @Override
  double area() {
     return Math.PI * Math.pow(radius, 2);
  }

  @Override
  public String toString() {
     return "Circle color is " + super.getColor() +
                                 "and area is : " + area();
  }
}
```

Abstraction Example

- **Example:**

```
class Rectangle extends Shape {
  double length;
  double width;

  public Rectangle(String color, double length, double width) {
    // calling Shape constructor
    super(color);
    System.out.println("Rectangle constructor called");
    this.length = length;
    this.width = width;
  }

  @Override
  double area() {
    return length * width;
  }

  @Override
  public String toString() {
    return "Rectangle color is " + super.getColor() +
                                   "and area is : " + area();
  }
}
```

```
public class Test {
   public static void main(String[] args) {
     Shape s1 = new Circle("Red", 2.2);
     Shape s2 = new Rectangle("Yellow", 2, 4);
     System.out.println(s1.toString());
     System.out.println(s2.toString());
   }
}
```

Output:

```
Shape constructor called
Circle constructor called
Shape constructor called
Rectangle constructor called
Circle color is Redand area is : 15.205308443374602
Rectangle color is Yellowand area is : 8.0
```

Interface

- An **interface in Java** is a blueprint of a class. It has static constants and abstract methods.
- The interface in Java is *a mechanism to achieve abstraction*.
  - ✓ There can be only abstract methods in the Java interface, not method body.
  - ✓ It is used to achieve abstraction and multiple inheritance in Java.
- **Why use Java interface?**

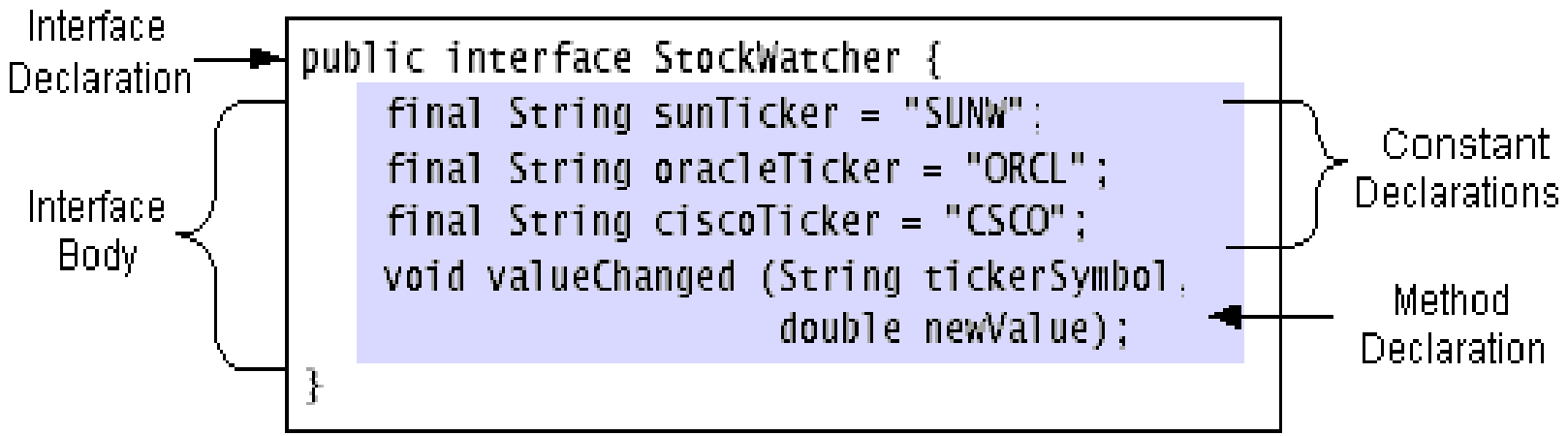
1. It is used to achieve abstraction.
2. By interface, we can support the functionality of multiple inheritance.
3. It can be used to achieve loose coupling.

Interface

- **Syntax:**

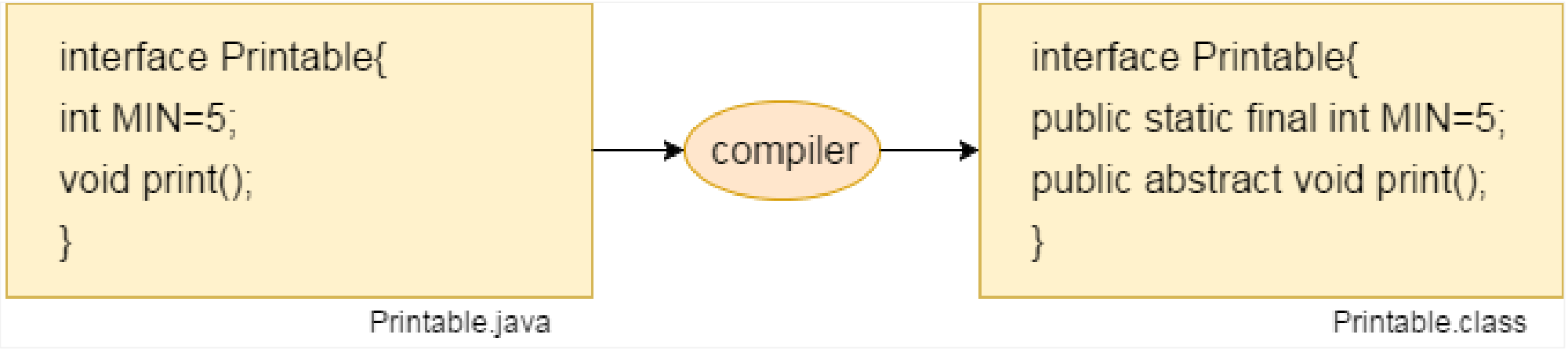
```
[public] interface <InterfaceName>[extends SuperInterface] {
        // Interface Body
   }
```

- **Example:**



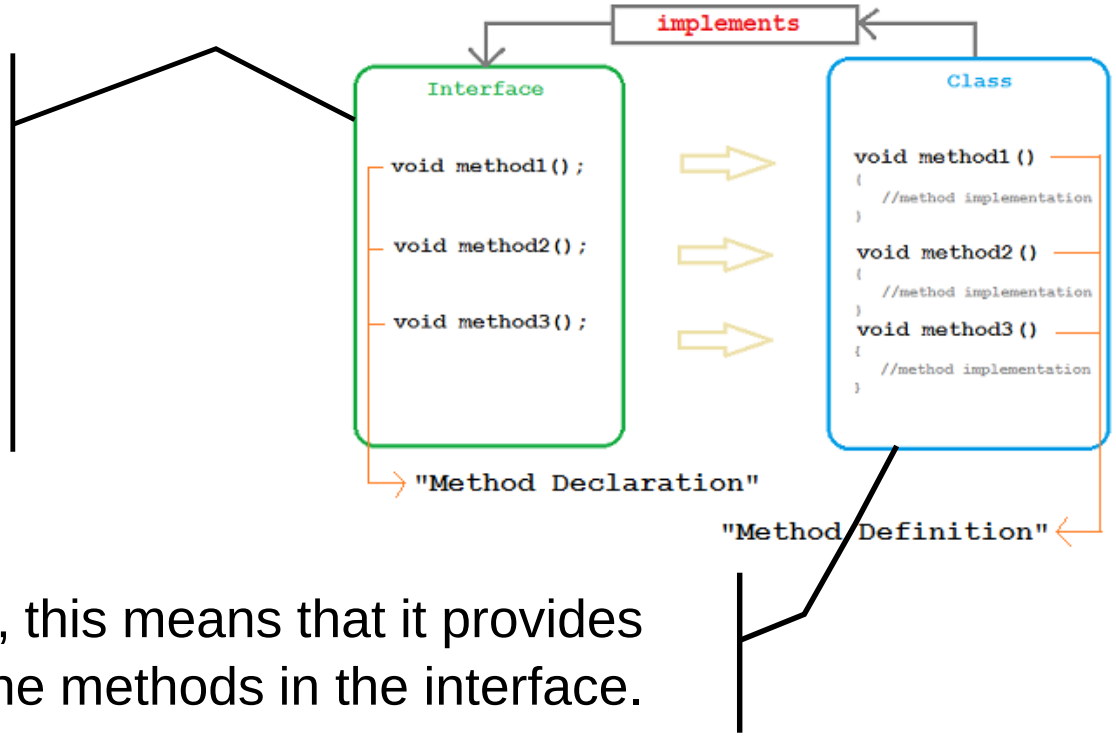
Interface

- The Java compiler adds public and abstract keywords *before the interface method*.
- Moreover, it adds public, static and final keywords *before data members*.
- **Consider the following figure:**



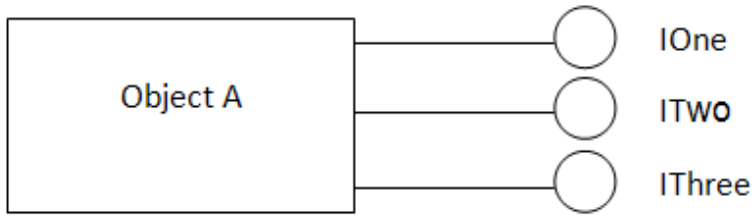
Interfaces

- ✓ An interface is a definition of method prototypes and possibly some constants (static final fields).
- ✓ An interface does not include the implementation of any methods.

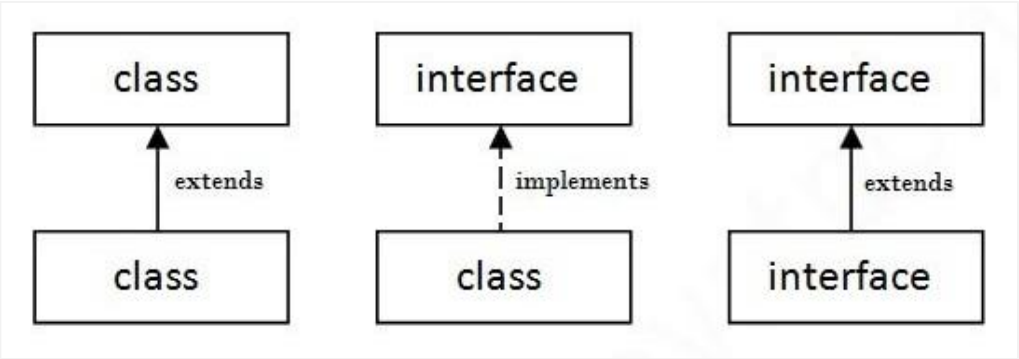


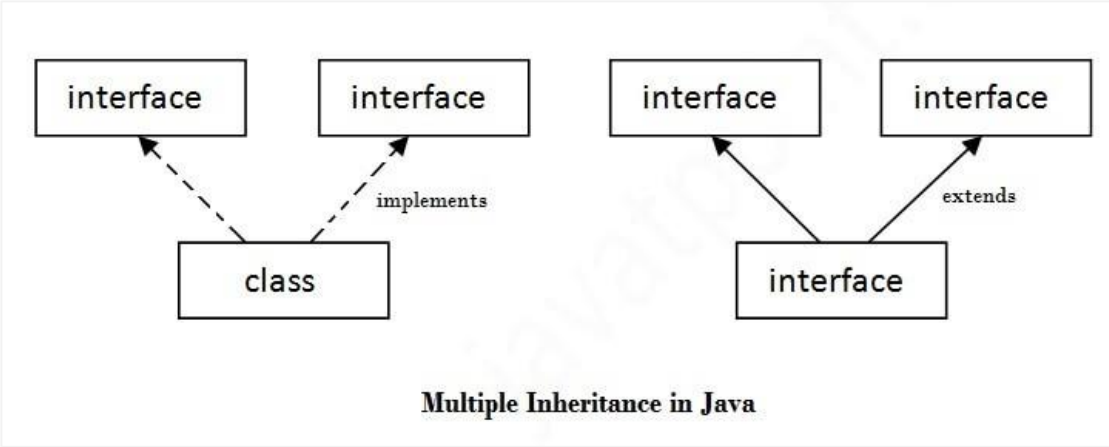
- ✓ A **class can `implement` an interface**, this means that it provides implementations for all the methods in the interface.

- Java classes can implement any number of interfaces (**multiple interface inheritance**).



Relationship Between Class and Interface





Class	Interface
✓ In class, you **can** instantiate variables and create an object.	✓ In an interface, you **can't** instantiate variables and create an object.
✓ A class can contain concrete (with implementation) methods	✓ The interface cannot contain concrete (with implementation) methods
✓ The access specifiers used with classes are private, protected, and public.	✓ In Interface only one specifier is used- public.

Java Interface Example 1: Bank

- Let's see another example of java interface which provides the implementation of Bank interface.

```
interface Bank {
  float rateOfInterest();
}

class SBI implements Bank {
  public float rateOfInterest() {
    return 9.15f;
  }
}

class PNB implements Bank {
  public float rateOfInterest() {
    return 9.7f;
  }
}
```

```
public class TestInterface {

  public static void main(String[] args) {

    Bank b = new SBI();
    System.out.println("ROI: " + b.rateOfInterest());
  }
}
```

Output:

```
ROI: 9.15
```

Interfaces Example 2

- Example:

```
public interface Forward {
   void drive();
}

public interface Stop {
   void park();
}

public interface Speed {
   void turbo();
}

public class GearBox {
   public void move() {
}
```

```
class Automatic extends GearBox
                        implements Forward, Stop, Speed {
    public void drive() {
        System.out.println("drive()");
    }
    public void park() {
        System.out.println("park()");
    }
    public void turbo() {
        System.out.println("turbo()");
    }
    public void move() {
        System.out.println("move()");
    }
}
```

Interfaces Example 2

- **Example:**

```
public class Car {
    public static void cruise(Forward x) {
       x.drive();
    }
    public static void park(Stop x) {
       x.park();
    }
    public static void race(Speed x) {
       x.turbo();
    }
    public static void move(GearBox x) {
       x.move();
    }
    public static void main(String[] args) {
       Automatic auto = new Automatic();
       cruise(auto); // Interface Forward
       park(auto); // Interface Stop
       race(auto); // Interface Speed
       move(auto); // class GearBox
    }
}
```

Multiple inheritance in Java by interface

- Multiple inheritance is not supported in the case of class because of ambiguity.
- However, it is supported in case of an interface because there is no ambiguity.
- It is because its implementation is provided by the implementation class.
- **Example**:

```
interface Printable {
  void print();
}

interface Showable {
  void print();
}

class A4 implements Printable, Showable {
  public void print() {
    System.out.println("Printing and showing document");
  }
}

public class TestInterface2 {
public static void main(String[] args) {
    A4 a4 = new A4();
    a4.print();
  }
}
```

- **Output**:

```
Printing and showing document
```

Java 8 Default Method in Interface

- Since Java 8, we can have method body in interface. But we need to make it default method.
- **Example**:

```
interface Drawable {
  void draw();

  default void msg() {
    System.out.println("default method");
  }
}
class Rectangle implements Drawable {
  public void draw() {
    System.out.println("drawing rectangle");
  }
}

public class TestInterfaceDefault {
     public static void main(String[] args) {
         Drawable d = new Rectangle();
         d.draw();
         d.msg();
     }
}
```

Output:

```
drawing rectangle
default method
```

You can override the default implementation with their own version.

Java 8 Default Method in Interface

- **Examples:**
  - ✓ `Comparable` interface with a default `compareTo()` method: *Provides a basic comparison logic that implementing classes can use or override for specific needs*.
  - ✓ `List` interface with a default `sort()` method: *Offers a standard sorting algorithm that list implementations can leverage or replace with custom sorting logic.*

```
@SuppressWarnings({"unchecked", "rawtypes"})
default void sort(Comparator<? super E> c) {
   Object[] a = this.toArray();
   Arrays.sort(a, (Comparator) c);
   ListIterator<E> i = this.listIterator();
   for (Object e : a) {
     i.next();
     i.set((E) e);
   }
}
```

Java 8 Static Method in Interface

- Since Java 8, we can have static method in interface.
- **Example**:

```
interface Drawable {
  void draw();

  static int cube(int x) {
    return x * x * x;
  }
}

class Rectangle implements Drawable {
  public void draw() {
    System.out.println("drawing rectangle");
  }
}

public class TestInterfaceStatic {
     public static void main(String[] args) {
          Drawable d = new Rectangle();
          d.draw();
          System.out.println(Drawable.cube(3));
     }
}
```

Output:

```
drawing rectangle
27
```

Abstract class and Interface

- **Abstract class** and **interface** both are used to achieve abstraction where we can declare the abstract methods. Abstract class and interface both can't be instantiated.
- But there are many differences between abstract class and interface that are given below.

No.	Abstract class	Interface
1	Abstract class can **have abstract and non-abstract** methods.	Interface can have **only abstract** methods.
2	Abstract class **doesn't support multiple inheritance**.	Interface **supports multiple inheritance**.
3	Abstract class **can have final, non-final, static and non-static variables**.	Interface has **only static and final variables**.
4	Abstract class **can have static methods, main method and constructor**.	Interface **can't have static methods, main method or constructor**.
5	Abstract class **can provide the implementation of interface**.	Interface **can't provide the implementation of abstract class**.
6	The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.

7	**Example:**
	public abstract class Shape{
	public abstract void draw();
	}
		Example:
		public interface Drawable {
		void draw();
		}

Summary

- Polymorphism, which means "many forms,"
  - ✓ is the *ability to treat* an *object of any subclass* of a base class as if it *were an object of the base class*.
- Abstract class is a class that may contain abstract methods and implemented methods.
  - ✓ An *abstract method* is one *without a body* that is declared with the reserved word abstract
- An interface is a collection of constants and method declarations.
  - ✓ When a class implements an interface, it must declare and provide a method body for each method in the interface

References

- *https://docs.oracle.com/javase/tutorial/java/landl/abstract.html*
- *https://www.mygreatlearning.com/blog/polymorphism-in-java/*
- *https://www.javatpoint.com/runtime-polymorphism-in-java*
- *https://www.geeksforgeeks.org/polymorphism-in-java/*

Questions

THANK YOU!