Learning Consolidation

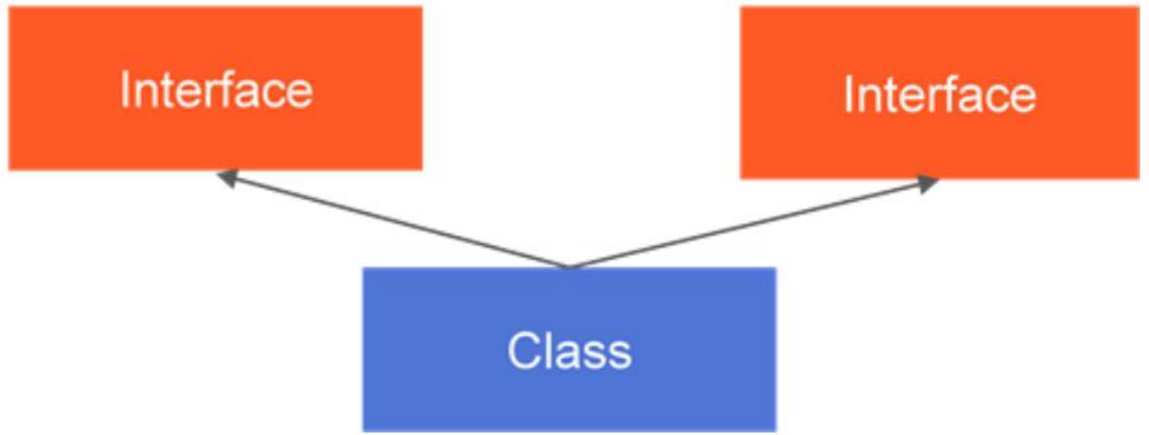
Implement Polymorphism

In this sprint, you have learned to:

- Explore interfaces
- Explain static polymorphism
- Explain dynamic polymorphism
- Define wrapper classes

What Are Interfaces?

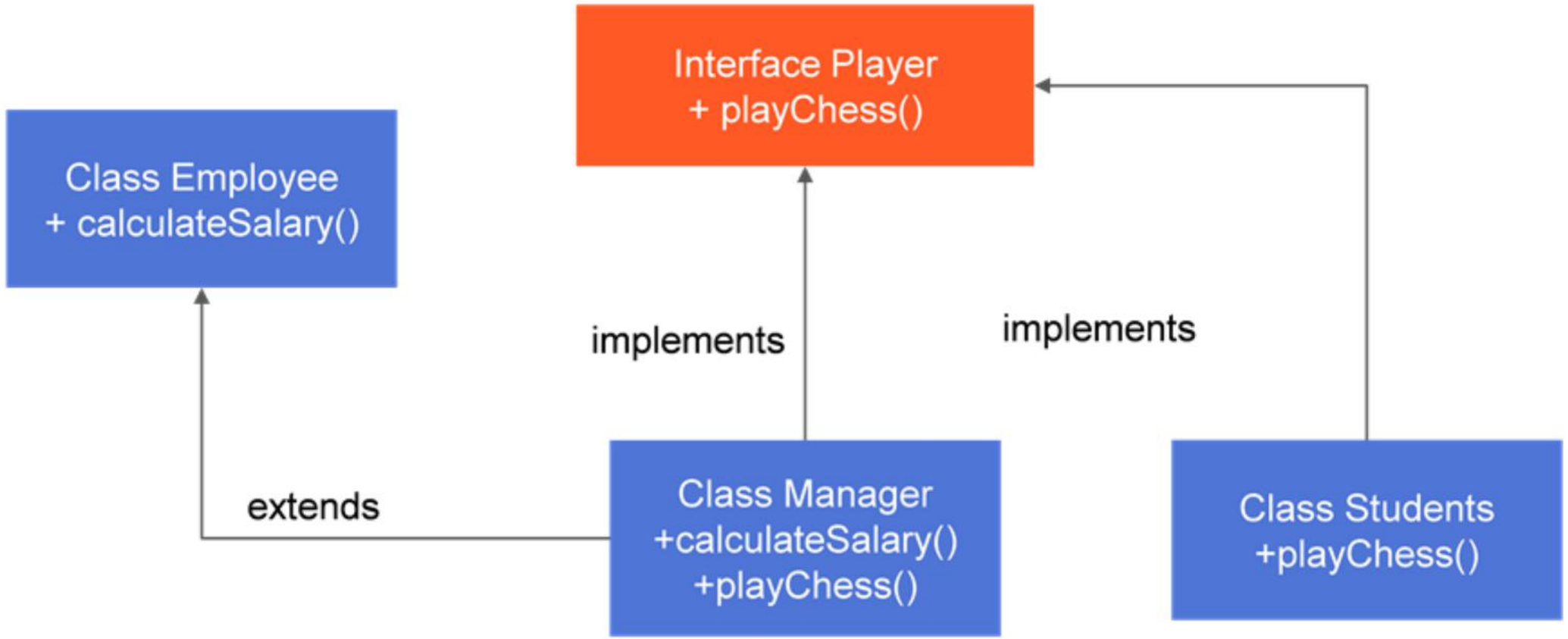
- The interface in Java is a mechanism to achieve abstraction and multiple inheritance.



- Interfaces in Java have only constant (static and final) variables and public abstract methods that do not have any implementation.
- Java interfaces also represents an **is-a** relationship.
- Interfaces impose a set of rules for the classes that implement them through the abstract methods.
- The class that implements that interface must provide implementation for all its abstract methods.

Unrelated Classes - Implement a Common Interface

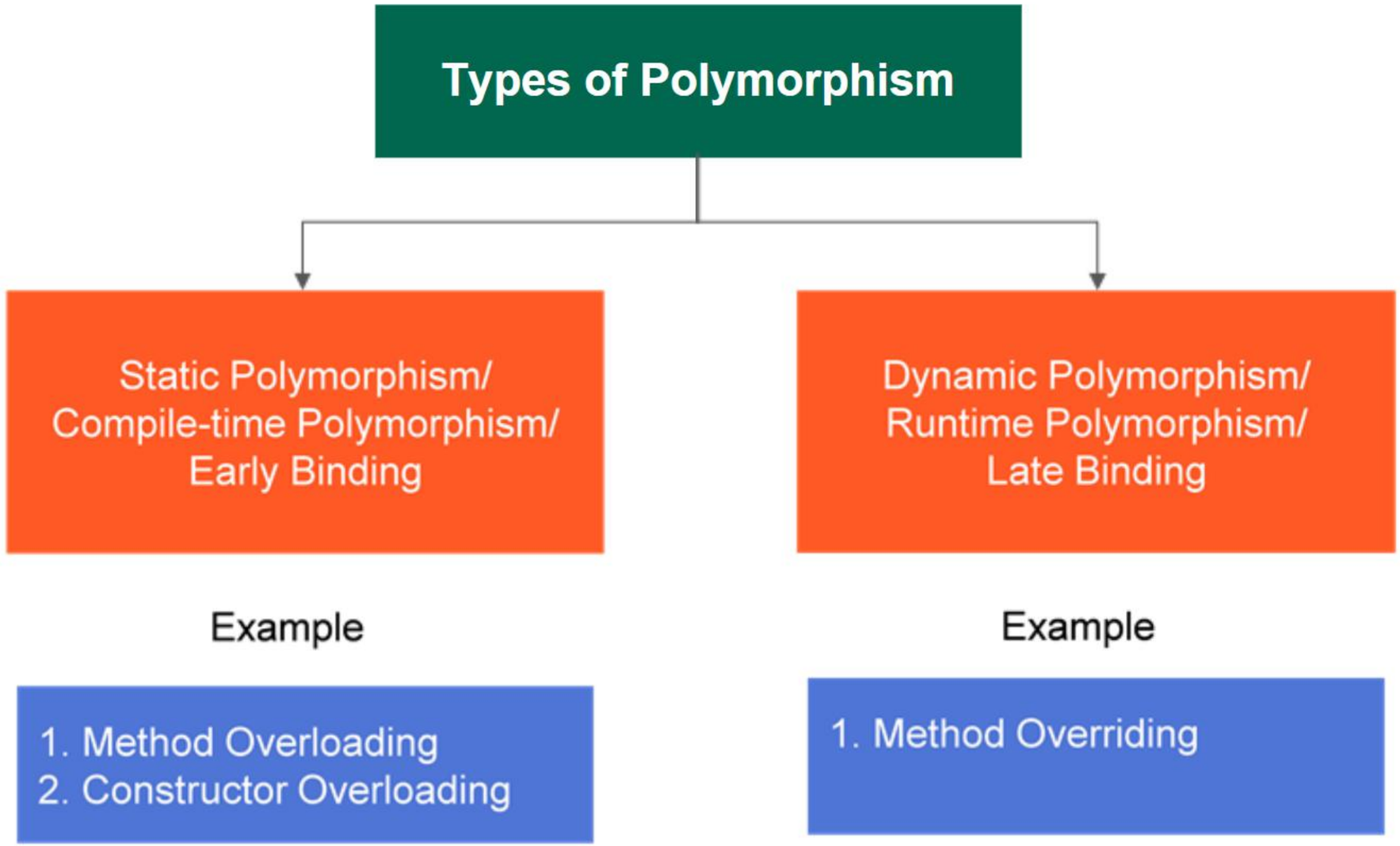
- Interfaces are meant to share behavior across independent or different objects.
- The `Manager` and `Student` are two different objects and can implement the `Player` interface.



Introduction to Polymorphism

- Polymorphism is a combination of two words: “poly” means “many” and “morph” means “forms.”
- Polymorphism means “the state of existing in different forms.”
- In OOP, polymorphism defines the ability of an object, method, or variable to take on different forms based on the characteristics required.
- Polymorphism enables an object to change its behavior based on the method being called.
- Similar to inheritance, polymorphism is also a core OOP concept; however, it primarily aims to improve code readability and provide a common interface to interact with objects.

Types of Polymorphism



Static Polymorphism - Method Overloading

- A class having multiple methods with the same name but different parameters, these methods are called overloaded methods, and this feature is called method overloading.
- The `add()` method is overloaded. These methods have the same name but accept different arguments.
- Here, the same method, `add()`, will perform different operations based on the parameter.
- The `add()` method that is called is determined by the compiler. Hence, it is also known as compile-time **polymorphism.**
- Method overloading makes the code more readable and easier to maintain.
- Method overloading is achieved by either:
  - Changing the number of arguments, or
  - Changing the data types of arguments.

```java
public class AddDemo {

    public void add(int num1, int num2) {}
    public void add(int num1, int num2, int num3) {}
    public void add(float num1, float num2) {}
    public void add(double num1, double num2) {}
    public static void main(String[] args) {
        AddDemo addDemo = new AddDemo();
        addDemo.add(10, 12);
        addDemo.add(12, 23, 34);
        addDemo.add(12.5f, 4.5f);
        addDemo.add(34.89, 23.899999);
    }
}
```

Dynamic Polymorphism – Method Overriding

- Dynamic polymorphism is implemented in Java by method overriding.
- When the same method is defined in a superclass and in a subclass, then the method of the subclass overrides the method of the superclass. This is known as method overriding.
- Method overriding enables a subclass to provide its own implementation of a method that already has an implementation defined in its superclass.
- To override a method, present in the superclass, the subclass method should have:
  - The same name.
  - The same parameters.
  - The same return type as the method in the superclass.

```
public abstract class Employee {
    String name;
    int empId;
    public void calculateSalary(){}
    public void calculateBonus(){}
}
class Manager extends Employee{
    @Override
    public void calculateSalary() {
        super.calculateSalary();
        //specific logic
    }

    @Override
    public void calculateBonus() {
        super.calculateBonus();
        //specific logic
    }
```

Method Overriding and Dynamic Method Dispatch

The `employee` variable is a reference for the `Employee` class.

```
public static void main(String[] args) {
    Employee employee = new Manager();
    employee.calculateSalary();
    employee.calculateBonus();

}
```

The `employee` variable is an object for the `Manager` class.

- Here, the `employee` variable is a reference of the `Employee` class but an object of the `Manager` class.
  - Here, the `employee` variable will check the `calculateSalary`() and `calculateBonus`() methods in the `Employee` class during compile time. This is called early binding.
  - But at runtime, it will become an object of the `Manager` class. Hence, at run time, it will call the `calculateSalary`() and `calculateBonus`() methods of the `Manager` class. This is also called late binding.
- Since method invocation is determined by the JVM at execution time, it is known as **runtime polymorphism or dynamic polymorphism.**
- The process whereby the call to the overridden method is resolved at runtime is called the dynamic method dispatch.

Wrapper Class

- Wrapper classes make the primitive data types act as objects because most of the time, we need to use the primitive types as objects for a program.
- A wrapper class wraps the primitive data type into an object.

Primitive data types	Wrapper classes
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean

Wrapper Class (contd.)

- Primitive type:
  - `int ageOfEmployee = 35;`

- Wrapper class:
  - `Integer salaryOfEmployee = new Integer(35000);`

Wrapper Class (contd.)

- Wrapping is done by the compiler.
- If a primitive is used where an object is expected, then the compiler, in its wrapper class, boxes the primitive.
- Similarly, if a number object is used where a primitive is expected, then the object is unboxed by the compiler.

Example of Boxing and Unboxing

```
public class Main {
        public static void main(String[] args)
        {
                int num1 = 10;
                Integer numObj1 = new Integer(20);
                Integer numObj2 = new Integer(num1); //Boxing
                Integer numObj3 = num1; //Autoboxing

                System.out.println("numObj1= " + numObj1 + "\n" +
"numObj2= " + numObj2 + "\n" + "numObj3= " + numObj3 + "\n");

                Integer numObj4 = new Integer(36);
                int num2 = numObj4.intValue(); //Unboxing
                int num3 = numObj4; //AutoUnboxing

        System.out.println("num2= " + num2 + "\n" + "num3= " + num3);
    }
}
```