PROGRAMMING AND DATA STRUCTURES

INTERFACES

HOURIA OUDGHIRI AND JIALIANG TAN

FALL 2023

OUTLINE

- How Polymorphism is implemented?
- Dynamic Binding
- Interfaces
- Implementing interfaces
- Examples of interfaces

Student Learning Outcomes

At the end of this chapter, you should be able to:

- Explain how polymorphism is implemented using dynamic binding
- Use interfaces to model common behavior between classes and for multiple inheritance

OOP Pillars

Encapsulation

Classes and Objects

Classes
Instance variables
Methods

Inheritance

Creating new Classes

Super Classes
Derived Classes

Polymorphism

Methods Across Classes

Abstract Classes
Dynamic Binding
Interfaces

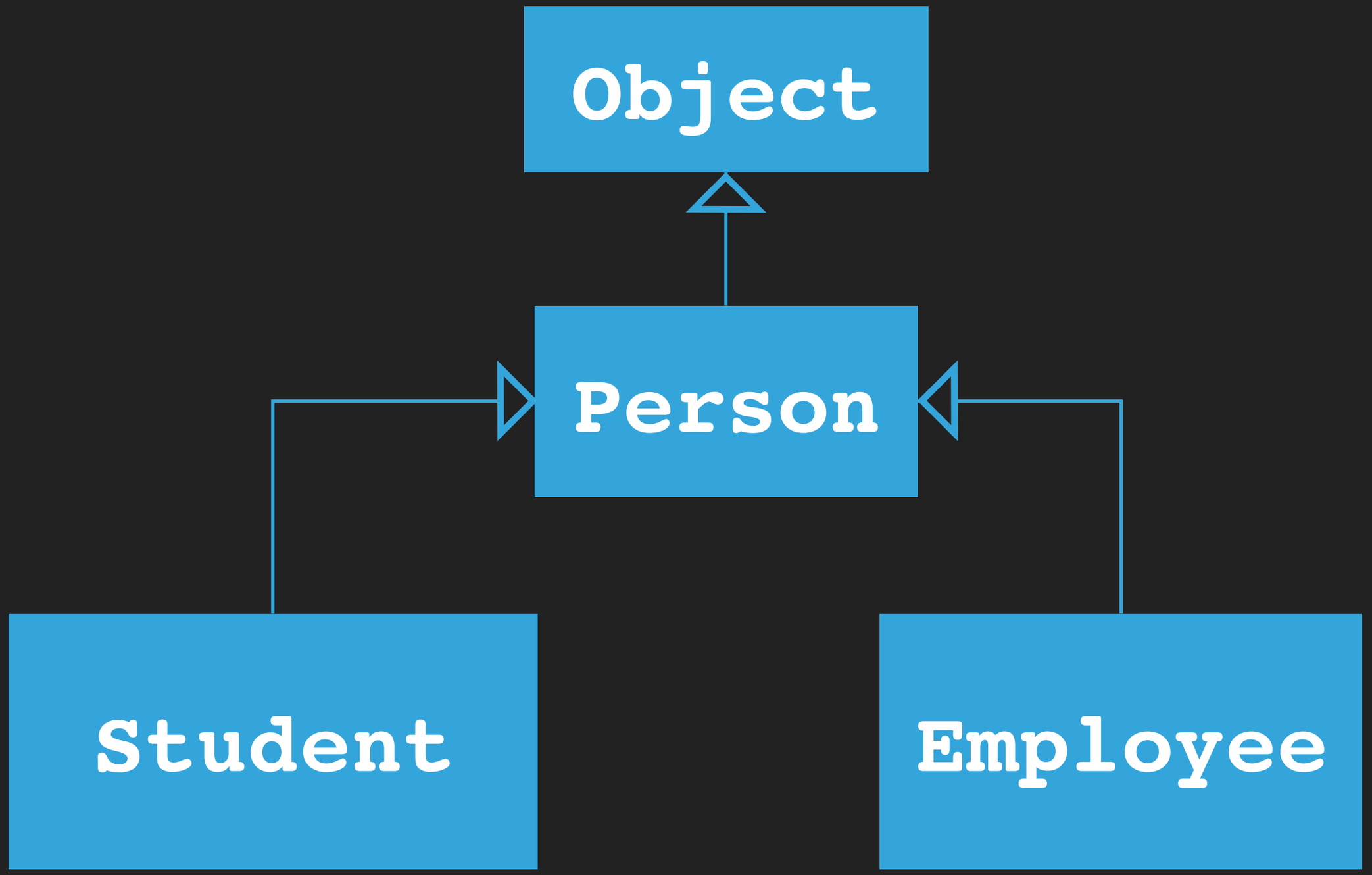
INTERFACES

- Every object of a derived class is an object of the base class
- **Polymorphism**: a variable of a super type can refer to a sub type object

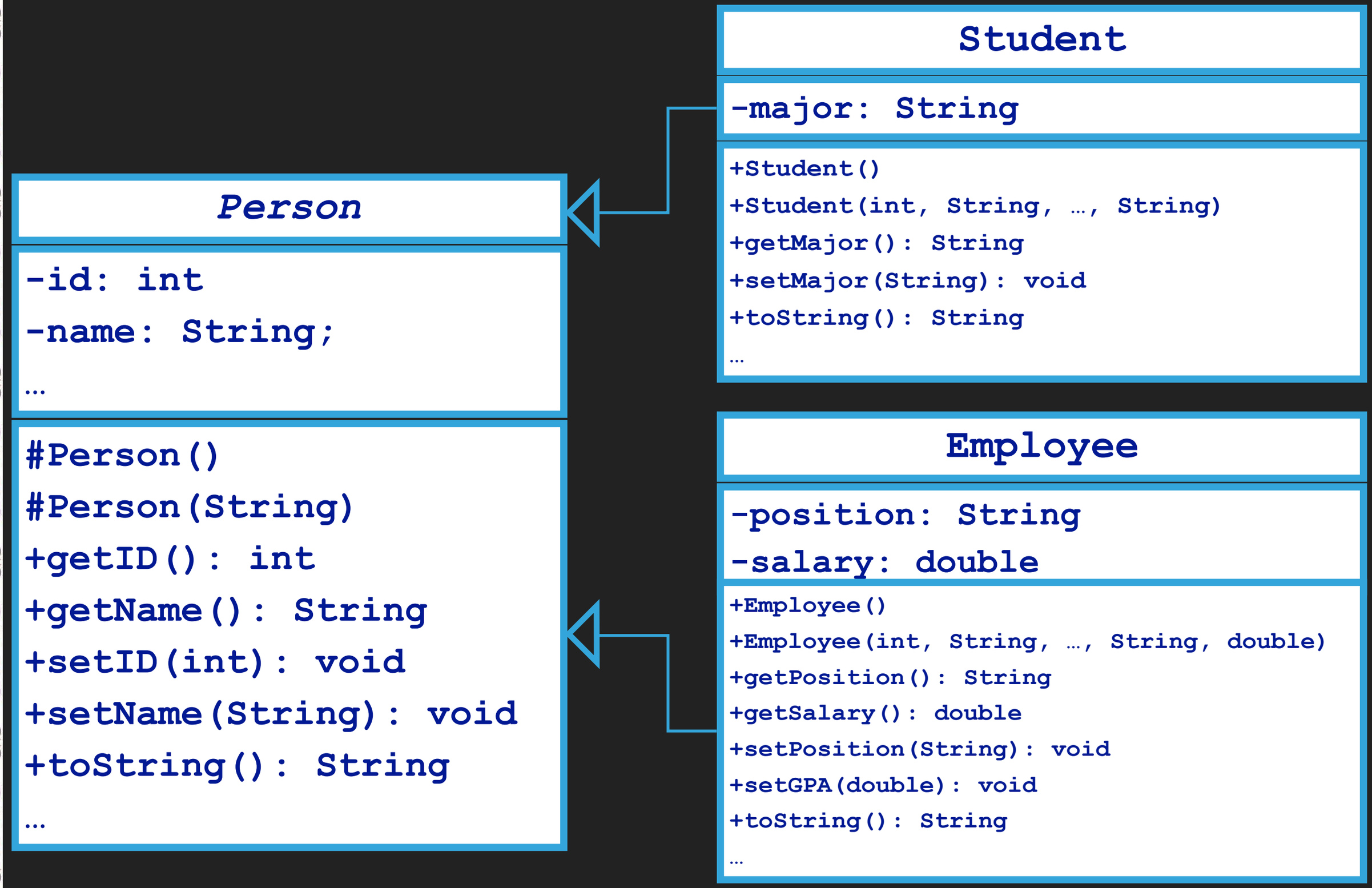
```
Person p = new Student();

String name = p.getName();
```

Polymorphism



INTERFACES


Person
-id: int
-name: String;
…
#Person()
#Person(String)
+getID(): int
+getName(): String
+setID(int): void
+setName(String): void
+toString(): String
…
Student
-major: String
+Student()
+Student(int, String, …, String)
+getMajor(): String
+setMajor(String): void
+toString(): String
…
Employee
-position: String
-salary: double
+Employee()
+Employee(int, String, …, String, double)
+getPosition(): String
+getSalary(): double
+setPosition(String): void
+setGPA(double): void
+toString(): String
…

INTERFACES

```
public class Test{
 public static void main(String[] args){
    Person[] people = new Person[2];
    people[0] = new Student(12345, “”, …, “CSE”);
    people[1] = new Employee(22222,“”, …, “”, 50000);
    for (int i=0; i < people.length; i++)
      System.out.println(people[i].toString());
 }
}
```

- **`people[i]`** may refer to an object of type **`Student`**, or **`Employee`**
- How does the compiler or JVM know?

Dynamic Binding

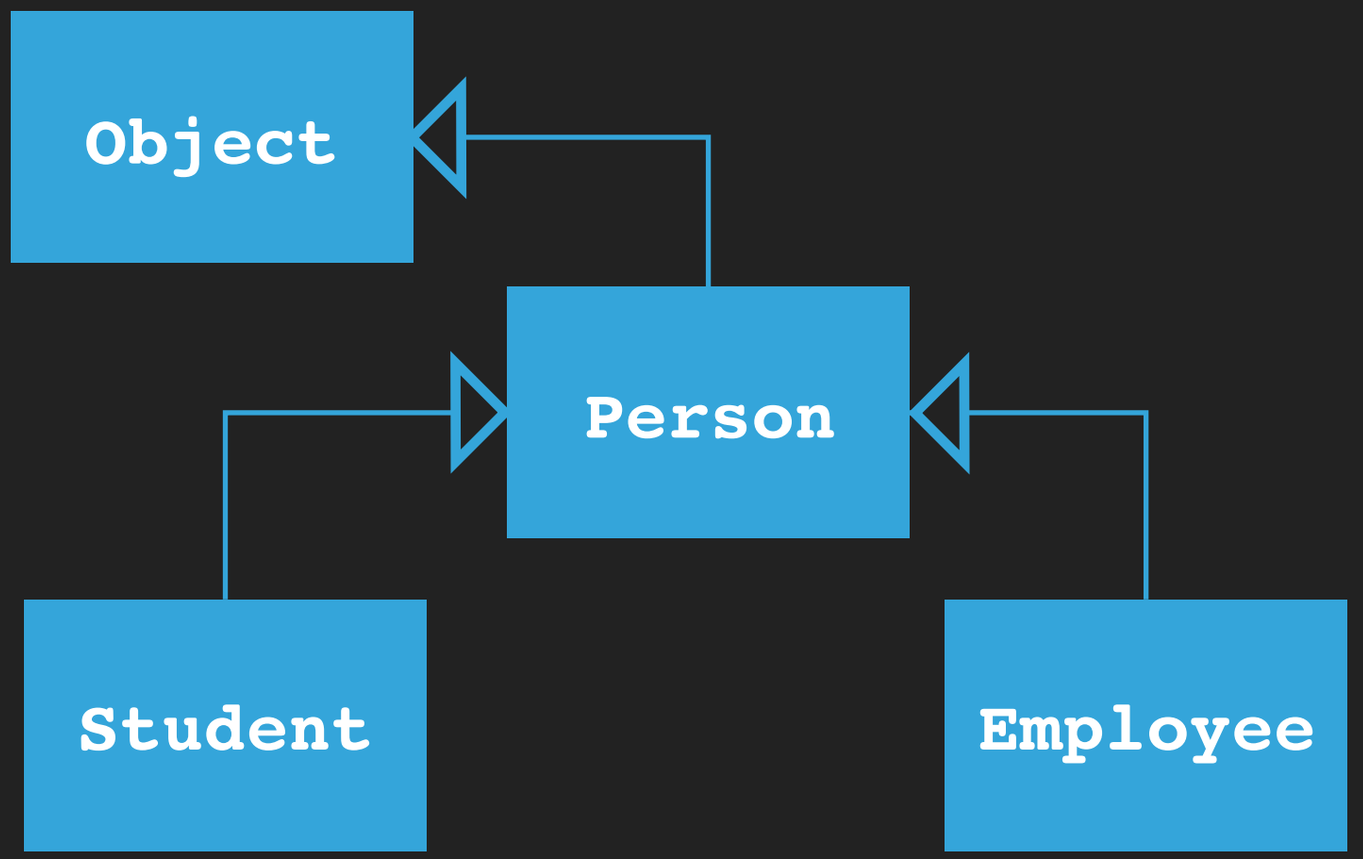
- JVM (Java Virtual Machine) decides which method is invoked at runtime
- Objects have a declared type and an actual type
- Methods are called on the actual type

```
Person p = new Student();
```

Declared Type

Actual Type

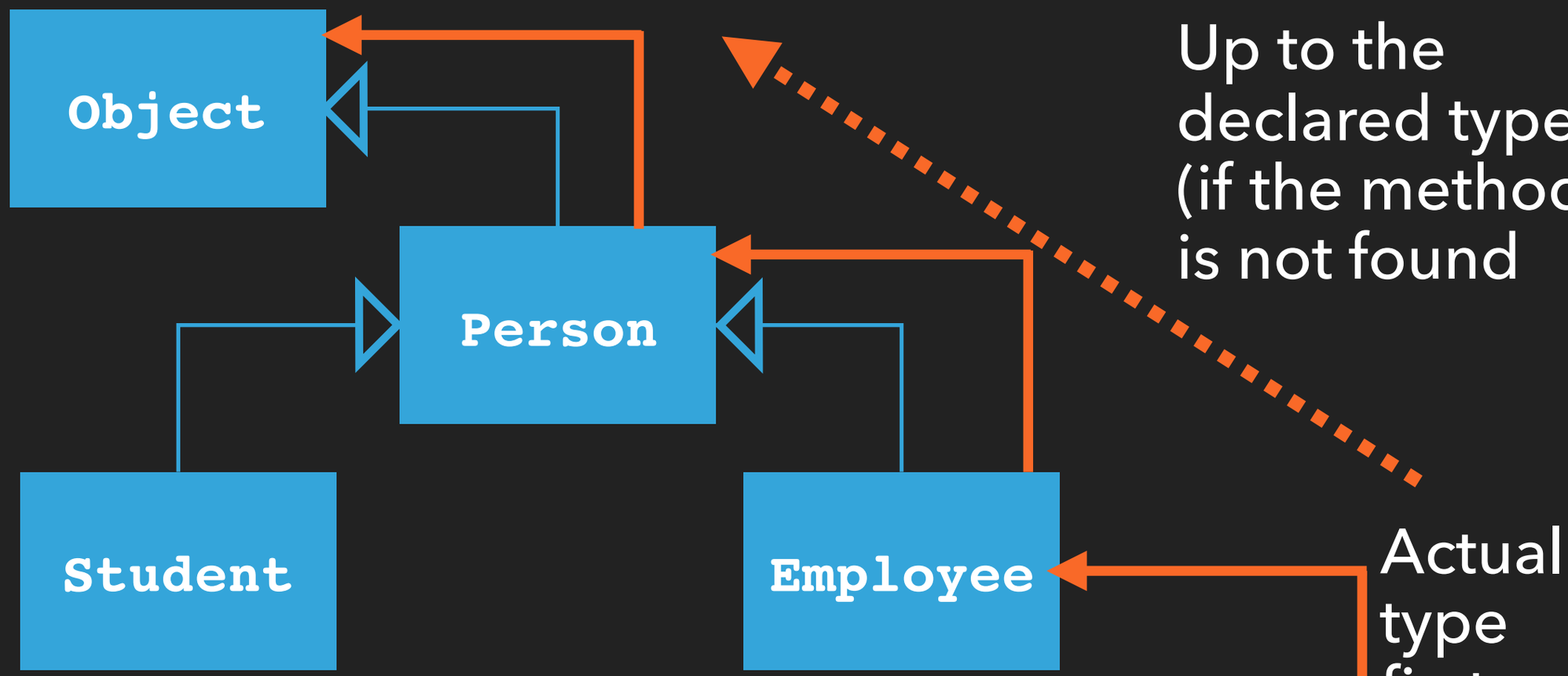
Dynamic Binding




```
Object obj = new Employee();

System.out.print(obj.toString());
```

Dynamic Binding




```
Object obj = new Employee();
System.out.print(obj.toString());
```

Dynamic Binding

```
public class DynamicBindingDemo {
  public static void main(String[] args) {
   print(new GraduateStudent());
   print(new Student());
   print(new Person());
   print(new Object());
  }
  public static void print(Object x) {
    System.out.println(x.toString());
  }
}
class GraduateStudent extends Student {
}
class Student extends Person {
  public String toString() {
    return "Student";
  }
}
class Person extends Object {
  public String toString() {
    return "Person";
  }
}
```

Dynamic Binding

```
public class Test{
    public static void main(String[] args){
      Integer[] list1 = {12, 24, 55, 1};
      Double[] list2 = {12.4, 24.0, 55.2, 1.0};
      printArray(list1);
      printArray(list2);
    }
    public static void printArray(Object[] list)
    {
      for(Object o: list)
         System.out.print(o.toString() + " ");
      System.out.println();
    }
}
```

Object Casting

- An object of the sub class is an object of the super class
- An object of the super class is not an object of the sub class

```
Person p = new Student();//OK

Student s = p;                //ERR
```

Object Casting

- Up casting (implicit)

```
Object obj = new Student();
```

- Down casting (has to be explicit)

```
Student s = obj; // ERROR

Student s = (Student) obj;//down-casting
```

Object Casting

- Down casting

```
Object obj = new Student();

Student s = (Student) obj;
```

- If the actual type of **`obj`** is not **`Student`**, **`ClassCastException`** is thrown
- Avoid the error by checking the actual type of the object (using **`instanceof`** operator)

Object Casting

- Overriding **`equals(Object)`** in class **`Person`**

```
public boolean equals(Object obj){
   if (obj instanceof Person) {
      Person s = (Person) obj;
      return (id == s.id);
   }
   else
      return false;
}
```

INTERFACES

- Abstract class - common behavior for related sub classes
- Interface - common behavior for classes not necessarily related

Abstract Classes

- Abstract classes cannot be instantiated - but can be used as a data type (polymorphism)
- Abstract methods make the class abstract but the class can be abstract without abstract methods
- Abstract methods must be implemented in the sub class. If they are not, the subclass remains abstract
- A sub class can be abstract even if the super class is not (`Object` and `Person`)

Interfaces

- Class-like construct for defining common operations (behavior) for objects from unrelated classes

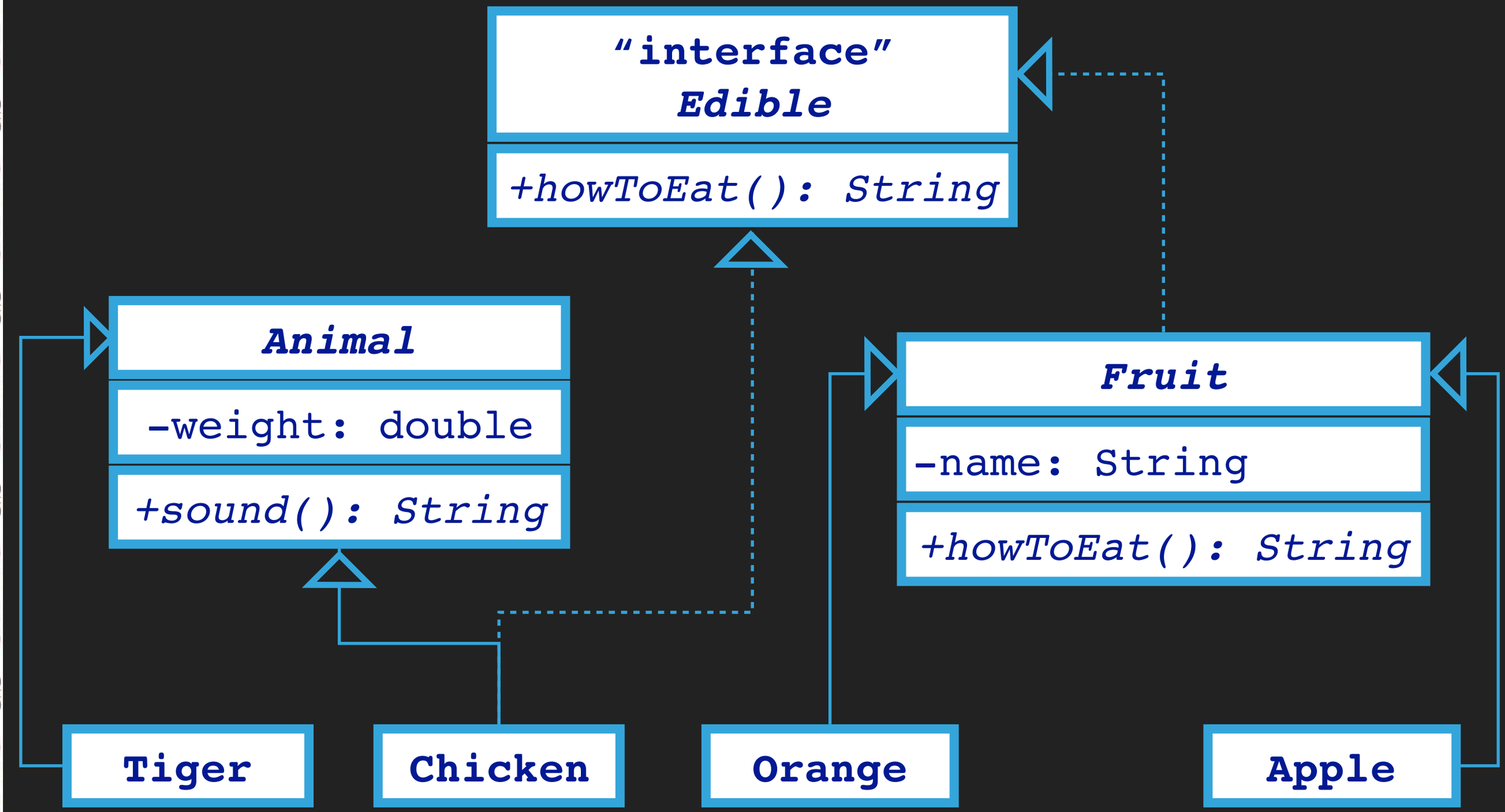
- Similar to abstract classes but model behavior of objects of unrelated classes

- Examples: Comparable, Edible, Cloneable, Drawable, etc…

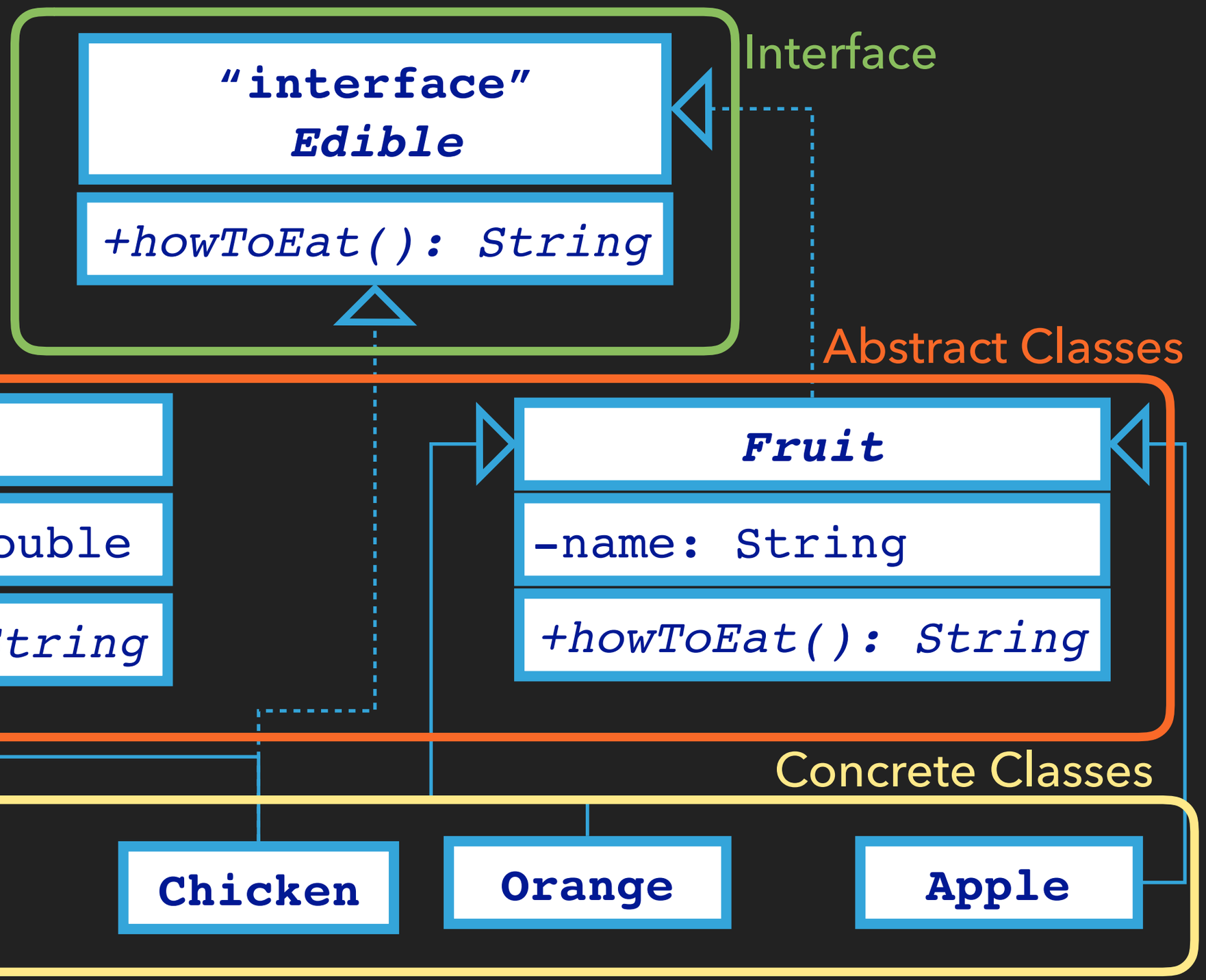
Interfaces

- Defined using the keyword **`interface`** instead of **`class`**
- Contains only static constants, static methods, and abstract methods
- A class **`implements`** an interface (instead of **`extends`**)

Abstract Classes and Interfaces



Abstract Classes vs Interfaces



Interface and Abstract Classes

```
public interface Edible{
    public String howToEat();
}
```

```
public abstract class Animal{
    private double weight;
    public double getWeight(){
        return weight;
    }
    public void setWeight(double w){
        weight = w;
    }
    public abstract String sound();
}
```

```
public class Chicken extends Animal implements Edible{
    public String sound(){
        return "Chicken sound: cock-a-doodle-doo";
    }
    public String howToEat(){
        return "How to eat chicken: Fry it";
    }
}
```

```
public class Tiger extends Animal{
    public String sound(){
        return "Tiger sound: RROOAARR";
    }
}
```

```
public abstract class Fruit implements Edible{
    private String name;
    public String getName(){
        return name;
    }
    public void setName(String n){
        name = n;
    }
}
```

```
public class Orange extends Fruit{
    public String howToEat(){
        return "How to eat an orange: Make orange juice";
    }
}
```

```
public class Apple extends Fruit{
    public String howToEat(){
        return "How to eat an apple: Make apple pie";
    }
}
```

Test Class

```
public class Test{
    public static void main(String[] args){
        Object[] objects = {new Tiger(),
                            new Chicken(),
                            new Apple(),
                            new Orange()};
        for(Object o: objects){
            if(o instanceof Edible){
                System.out.println(((Edible)o).howToEat());
            }
            if(o instanceof Animal){
                System.out.println(((Animal)o).sound());
            }
        }
    }
}
```

Interface `Comparable<E>`

- **`java.lang.Comparable`**: Interface to define the behavior of compareTo between objects of any class type **`E`**

- **`Comparable`** has only one abstract method **`compareTo()`**

```
public interface Comparable<E> {
        int compareTo(E obj);
}
```

Interface Comparable

```
public interface Comparable<E> {
        int compareTo(E obj);
}
```

obj1.compareTo(obj2)

- returns **0** if **obj1 == obj2**
- returns **> 0** if **obj1** comes after **obj2**
- returns **< 0** if **obj1** comes before **obj2**

Interface Comparable

```
public abstract class Person implements Comparable<Person>{
    // Data members
      …
    // Class Person constructors and methods
    /**
     * Methods compareTo
     * @param p Person object being compared to this object
     * @return integer indicates the order of the two objects this and p
     *          0 if the id of this and p are identical
     *          >0 if the id of this is greater than the id of p
     *          <0 if the id of this is less than the id of p
     */
    public int compareTo(Person p){
        return this.id - p.id;
    }
}
```

Interface `Comparable`

- **`java.util.Arrays.sort()`** accepts objects from any class that implements **`Comparable`**

```
public class Test{
    public static void main(String[] args){
        Person[] people = new Person[3];
        // code to create the objects
        . . .
        System.out.println("Original List");
        printArray(people);

        System.out.println("List sorted by name");
        sortArray(people, true); // sorting by name
        printArray(people);

        System.out.println("List sorted by id");
        java.util.Arrays.sort(people); // sorting by id
        printArray(people);
    }
```

Interface `Cloneable`

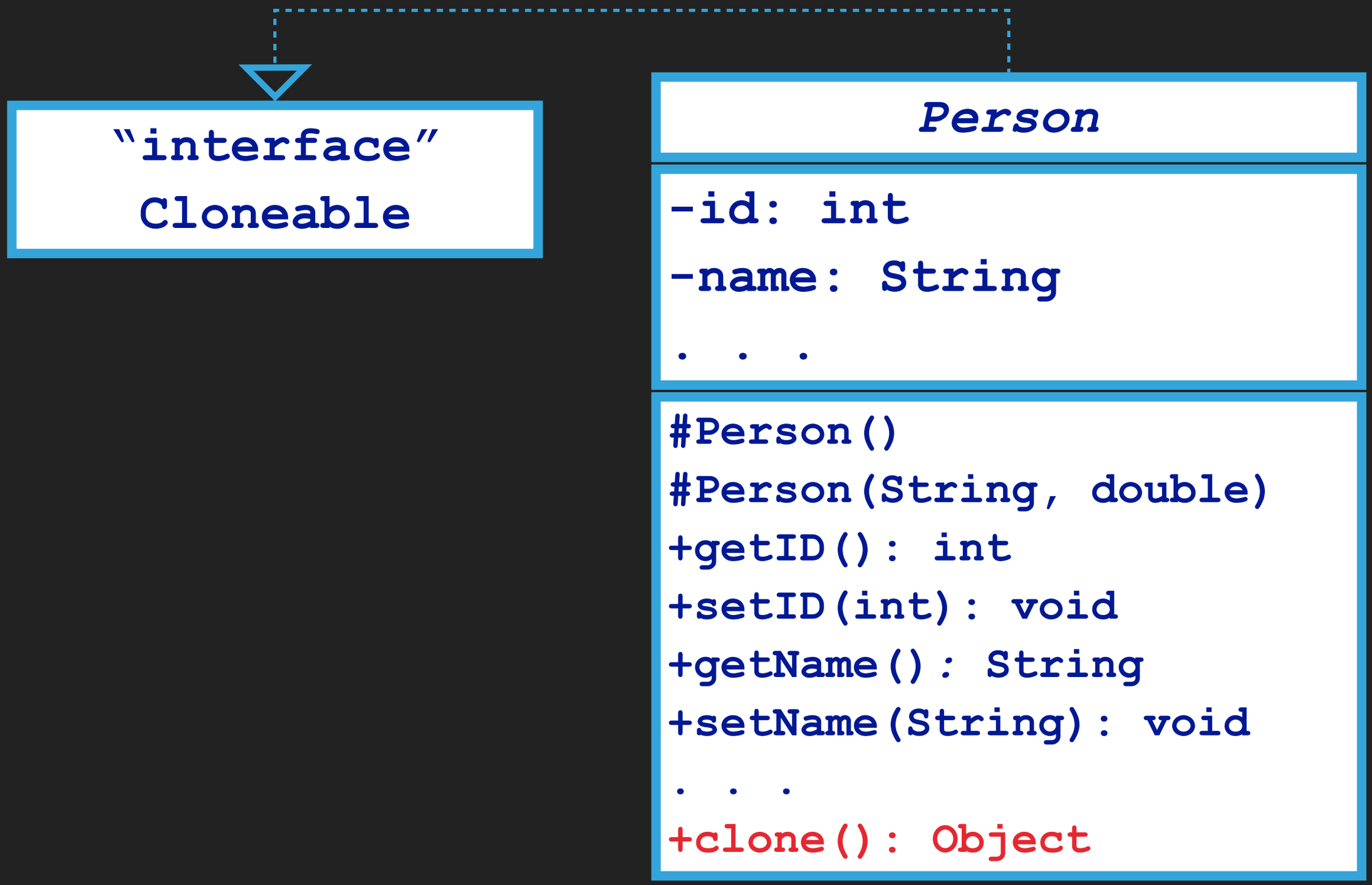
- **`java.lang.Cloneable`**: Empty Interface to define the behavior of the method clone() for objects of any class
- The interface is empty (marker interface) and is only used to mark a class as having the cloneable behavior

```
public interface Cloneable {
}
```

Interface `Cloneable`

- Implementing the interface **`Cloneable`** consists in overriding the method **`clone()`** from class **`Object`** (**`Object clone()`**)
- Many classes in the Java API implement the interfaces **`Comparable`** and **`Cloneable`**

Interface `Cloneable`



Interfaces - `Cloneable`

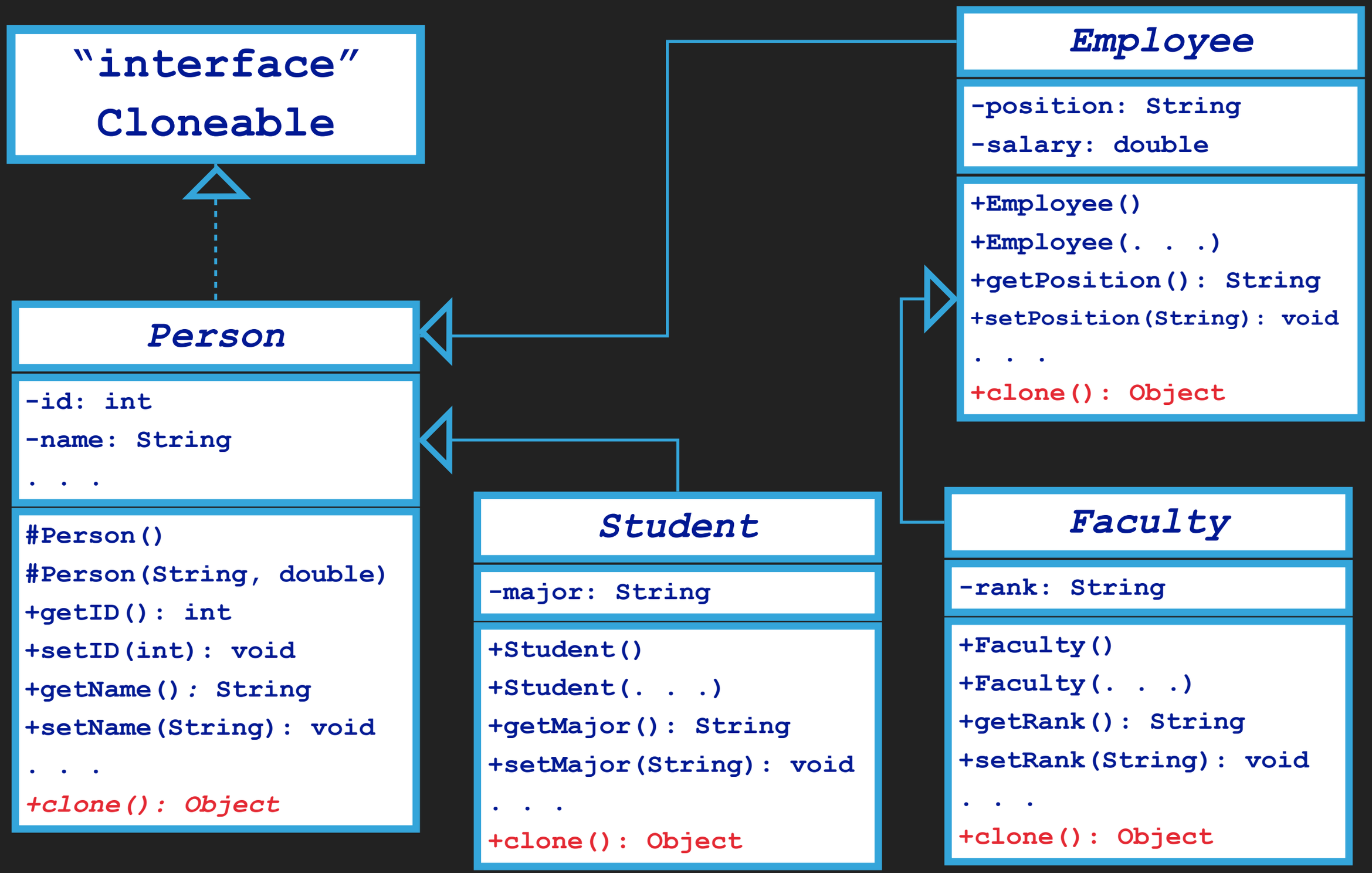
- `Clone()` may be implemented in two different ways
  - `Clone()` (**Shallow copy**)

    Return a reference to `this` object

  - `Clone()` (**Deep copy**)

    Return a new object with the attributes of `this` object

Interface `Cloneable`



Interfaces - Cloneable

- Clone() (Shallow copy)

```
// clone implemented using shallow copy
    public Object clone(){
        return this;
    }
```

- Clone() (Deep copy)

```
// clone implemented using shallow copy
public Object clone(){
  return new Student(this.getID(), this.getName(),
                     this.getAddress(), this.getPhone(),
                     this.getEmail(), this.major);
}
```

Interfaces - `Cloneable`

- Using the shallow copy `clone()`

```
Person p = (Person) (people[0].clone());
p.setName("Claire Vendome");
System.out.println(people[0].getName()); // Claire Vendome
System.out.println(p.getName()); // Claire Vendome
```

- Using the deep copy `clone()`

```
Person p = (Person) (people[0].clone());
p.setName("Claire Vendome");
System.out.println(people[0].getName()); // Claire Vendome
System.out.println(p.getName()); // Paul Leister
```

Implementing Multiple Interfaces

- Alternative to multiple inheritance
- Java does not allow multiple inheritance - extends only one class
- Java allows the implementation of multiple interfaces - implements a list of interfaces

Implementing Multiple Interfaces

```
public class Person implements Comparable<Person>,
                               Cloneable {
    …
    public int compareTo(Person p){
        return id - p.id;
    }
    public abstract Object clone();
}
```

Summary

	Data members	Constructors	Methods
Interface	Only static constants (static final)	No instantiation No constructor	Only abstract, default, and static
Abstract Class	No restrictions	No instantiation (Constructors invoked by sub classes only)	No restrictions
Concrete Class	No restrictions	Can be instantiated	No abstract methods

Summary

- Polymorphism - Dynamic Binding
- Abstract classes - common behavior between related classes - abstract methods
- Interfaces - common behavior between unrelated classes - Comparable, Cloneable, Edible, Scalable, Drawable, …
- Interfaces are used as an alternative to multiple inheritance