PROGRAMMING AND DATA STRUCTURES

ABSTRACT CLASSES AND INTERFACES

HOURIA OUDGHIRI

FALL 2021

OUTLINE

- Polymorphism
- Dynamic Binding
- Abstract Classes
- Interfaces

STUDENT LEARNING OUTCOMES

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

- Explain the concept of polymorphism and dynamic binding
- Create abstract classes and extend them to create specific concrete classes
- Use interfaces to model common behavior between classes and multiple inheritance

POLYMORPHISM – INTERFACES

OOP Pillars

Encapsulation	Inheritance	**Polymorphism**
Classes and Objects	Creating new Classes	Methods Across Classes
Classes Instance variables Methods	Super Classes Derived Classes	Dynamic Binding Abstract Classes Interfaces

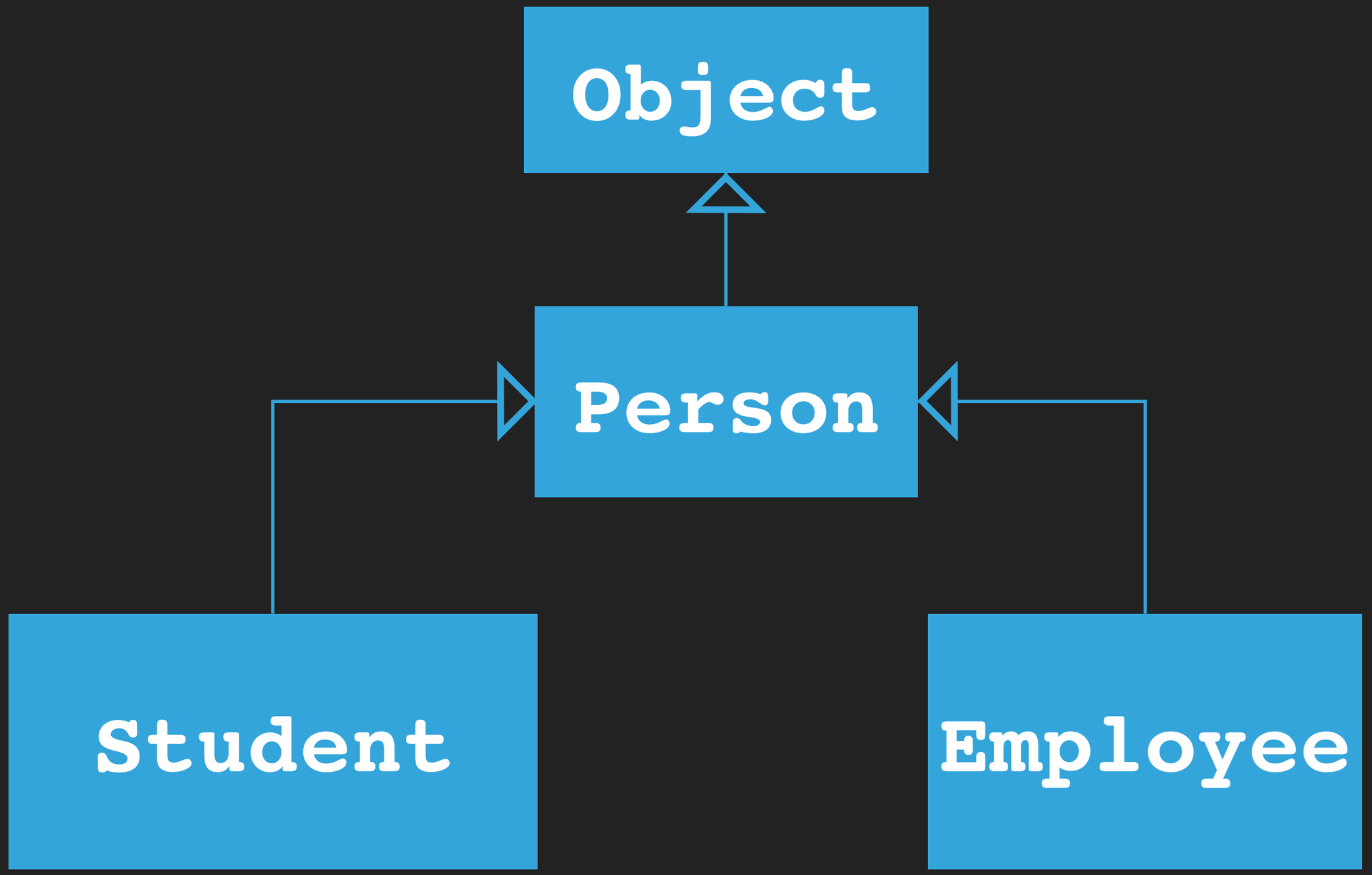
POLYMORPHISM – INTERFACES

- 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

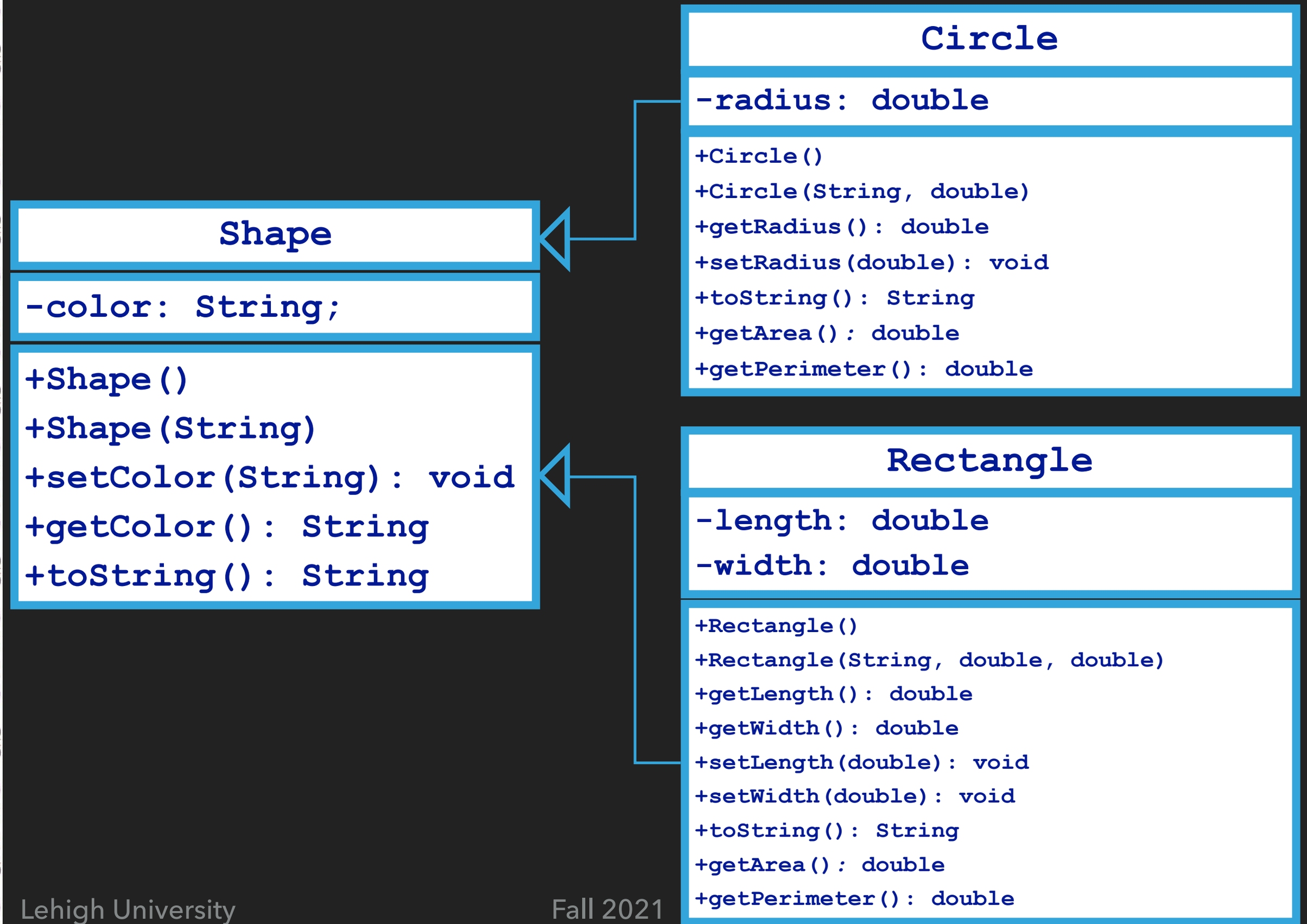
```
Person p = new Student();

name = p.getName();
```

Polymorphism



POLYMORPHISM – INTERFACES



POLYMORPHISM – INTERFACES

```
public class Test{
 public static void main(String[] args){
    Shape[] shapes = new Shape[2];
    shapes[0] = new Circle("red", 5.5);
    shapes[1] = new Rectangle("blue", 2.5, 3.75);
    for (int i=0; i < shapes.length; i++)
       System.out.println(shapes[i].toString());
  }
}
```

- `shapes[i]` may refer to an object of type **`Shape`**, **`Circle`**, or **`Rectangle`**
- How does the compiler know?

Dynamic Binding

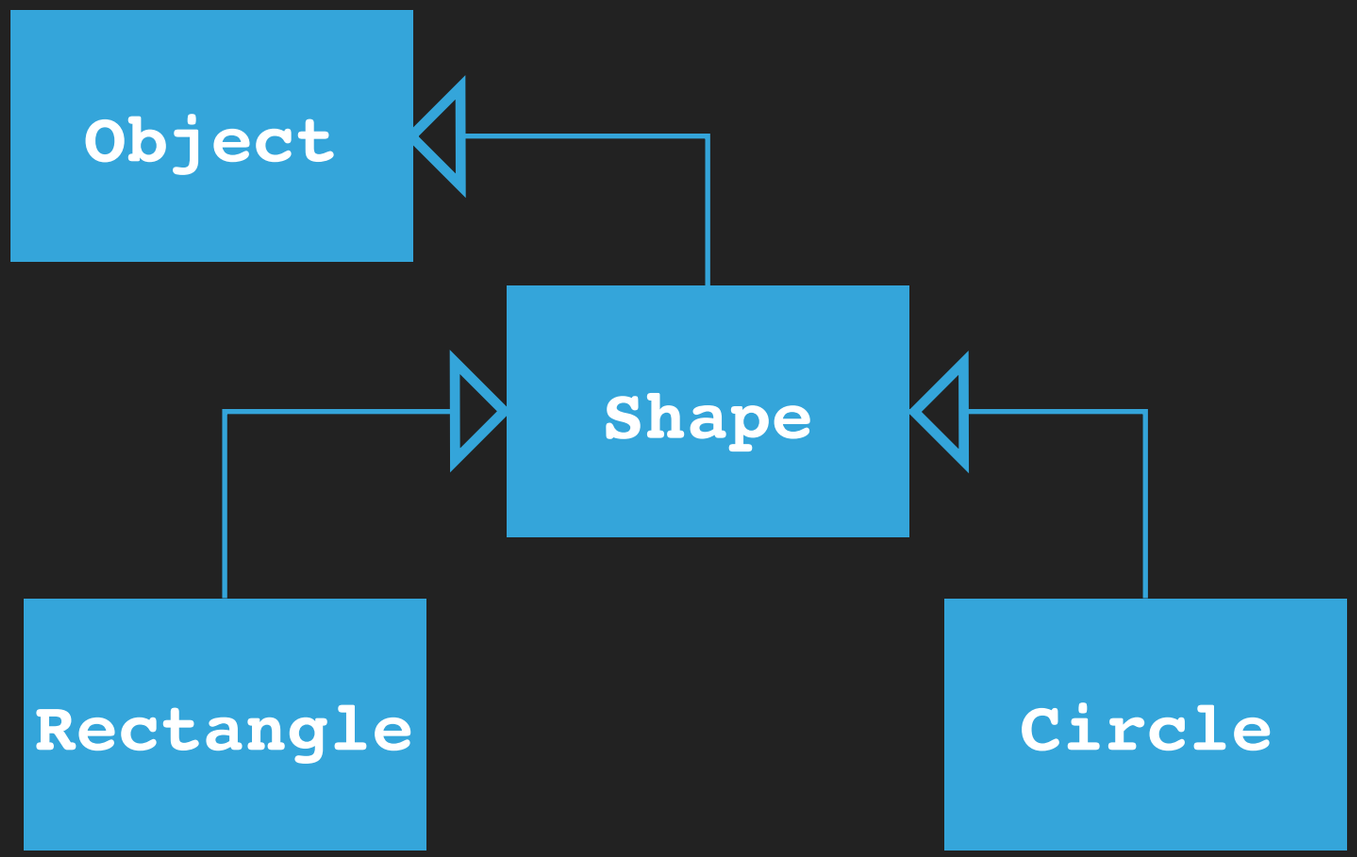
- JVM (Java Virtual Machine) decides which method is invoked at runtime
- Variables have a declared type and an actual type
- Methods are called on the actual type

```
Shape shape = new Circle();
```

Declared Type

Actual Type

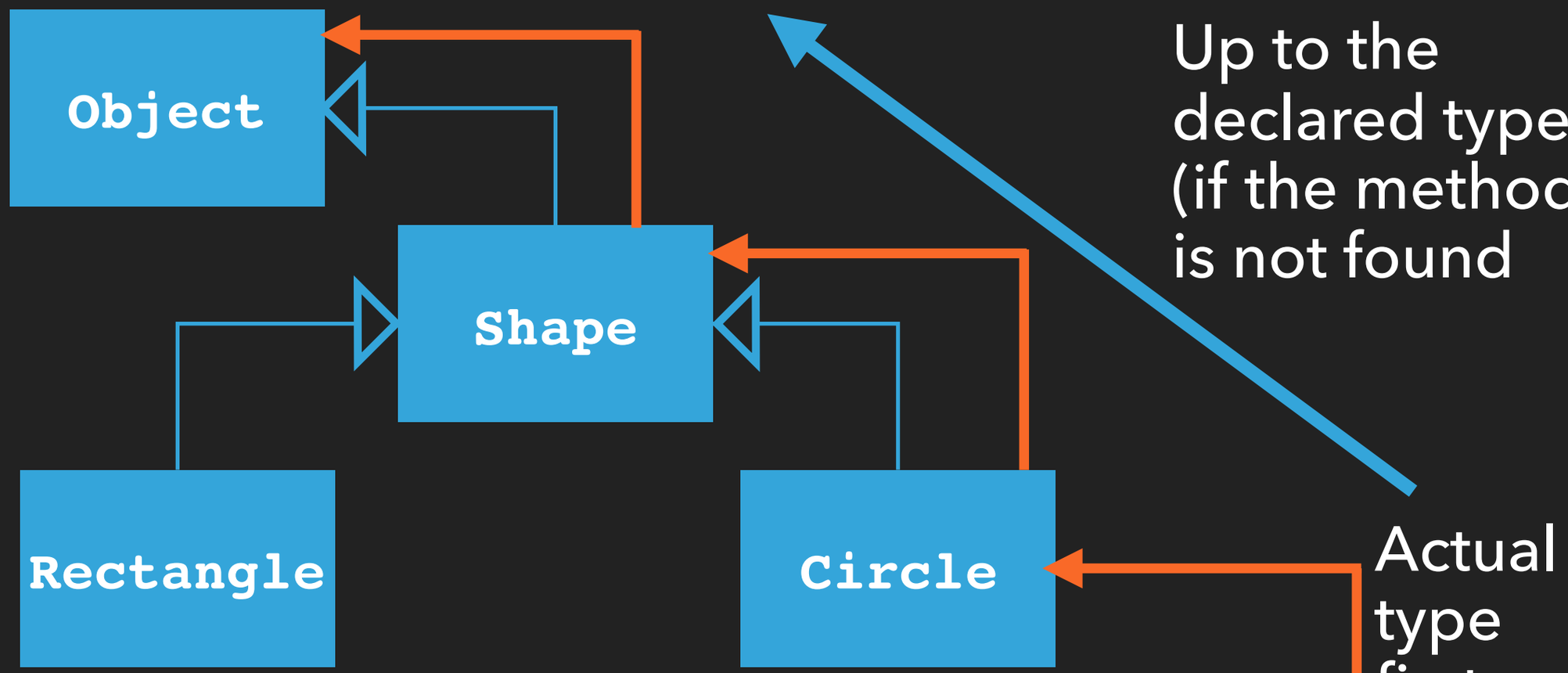
Dynamic Binding




```
Object obj = new Circle();

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

Dynamic Binding




```
Object obj = new Circle();

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

```
Shape shape = new Circle();//OK

Circle circle = new Shape();//ERR
```

Object Casting

- Up casting (implicit)

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

- Down casting (has to be explicit)

```
Circle c = obj; // ERROR

Circle c = (Circle) obj;// down-casting
```

Object Casting

- Down casting

```
Circle c = obj; // ERROR

Circle c = (Circle) obj;
```

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

Object Casting

- **`obj1 == obj2`** compares the references to `obj1` and `obj2`
- **`obj1.equals(obj2)`** should compare the attributes of the two objects
- Method **`equals()`** in class **`Object`** compares references only
- Must override **`equals()`** in every class where you need to compare object attributes

Object Casting

- instanceof operator

```
public boolean equals(Object obj){

  if (obj instanceof Circle) {

    Circle c = (Circle) obj;

    return (radius == c.getRadius());
  }
  else
    return false;
}
```

Abstract Classes

- Abstract class - common behavior for related sub classes
- Interface - common behavior for classes not necessarily related
- When the super class is too general that you cannot instantiate it (create objects), the class is made abstract

Abstract Classes

- Class **`Shape`** - abstract class
  creating a shape object does not make any sense (no real attributes)
- Class **`Shape`** - abstract methods **`getArea()`** and **`getPerimeter()`** but with no definition
- Common behavior: every sub class of Shape must have **`getArea()`** and **`getPerimeter()`**

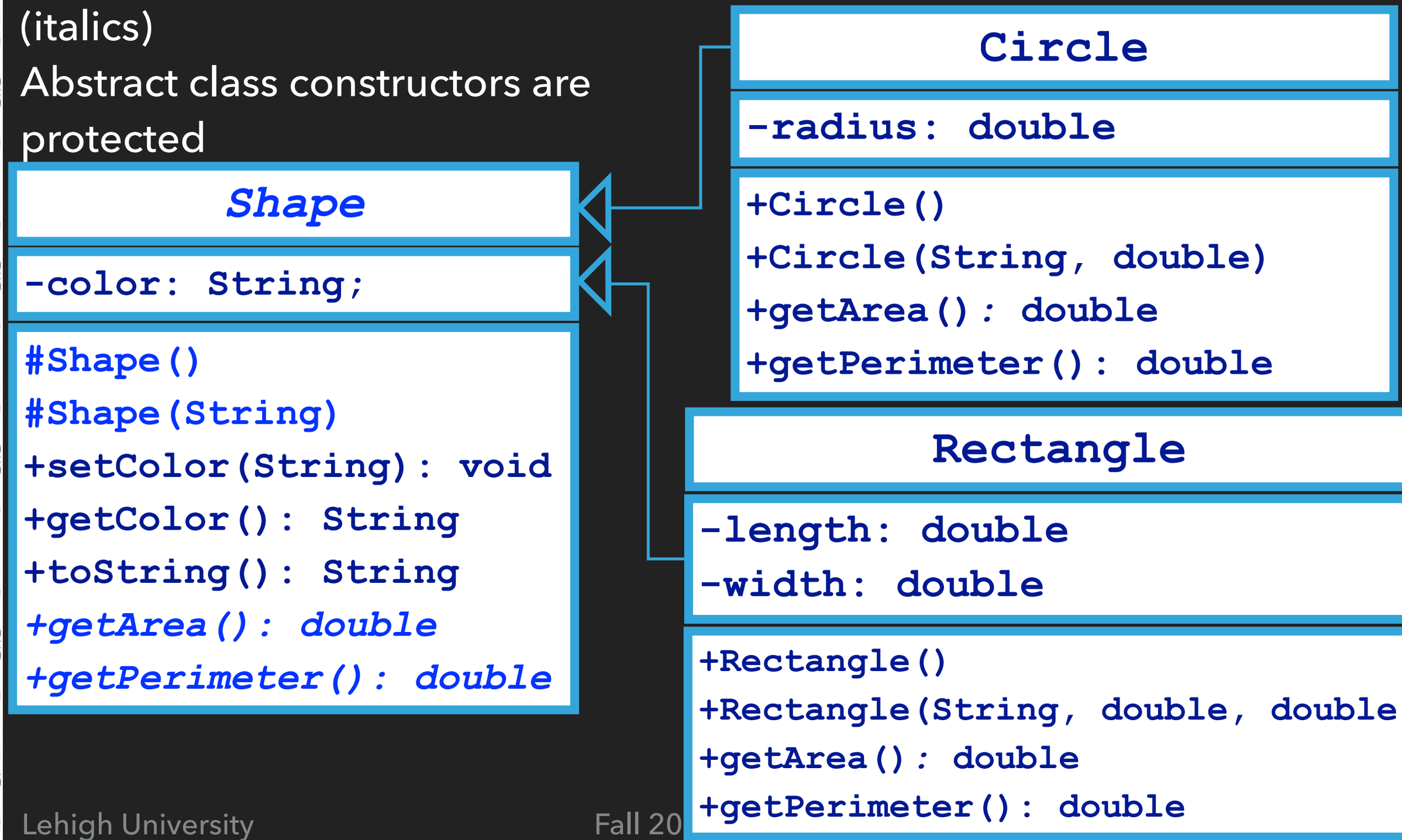
Abstract Classes

- Concrete Class - can be instantiated
- Abstract Class - cannot be instantiated - has abstract methods to be implemented in concrete sub classes
- **Common behavior**: abstract methods in the abstract class

Abstract Classes

Abstract class and abstract methods (italics)

Abstract class constructors are protected



Abstract Classes

```
public abstract class Shape {
    private String color;
   // constructors
   protected Shape(){…}
   protected Shape(String){…}
   // methods
   public void setColor(String){…}
   public String getColor(){…}
   public String toString(){…}
   public abstract double getArea();
   public abstract double getPerimeter();
}
```

Abstract Classes

```
public class Rectangle extends Shape {
  private double length, width;
  // constructors
  public Rectangle() {super(); …}
  public Rectangle(String c, double l,
  double w) {super(c); …}
  // methods
  . . .
  public double getArea() {
    return length * width;
  }
  public double getPerimeter(){
    return 2 * (length + width);
  }
}
```

```
public class Circle extends Shape{
  private double radius;
  // constructors
  public Circle(){super(); …}
  public Circle(String c, double r){
   super(c); …}
  // methods
  . . .
  public double getArea() {
    return radius * radius * Math.PI;
  }
  public double getPerimeter(){
    return 2 * radius * Math.PI;
  }
}
```

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 Shape)

Interfaces

- Interface - class-like construct for defining common operations (behavior) for objects from unrelated classes
- Similar to abstract classes but contain behavior of objects of unrelated classes
- Examples: Comparable, Edible, Cloneable, Drawable, etc…

Interfaces

- Defined using the keyword **`interface`** instead of **`class`**

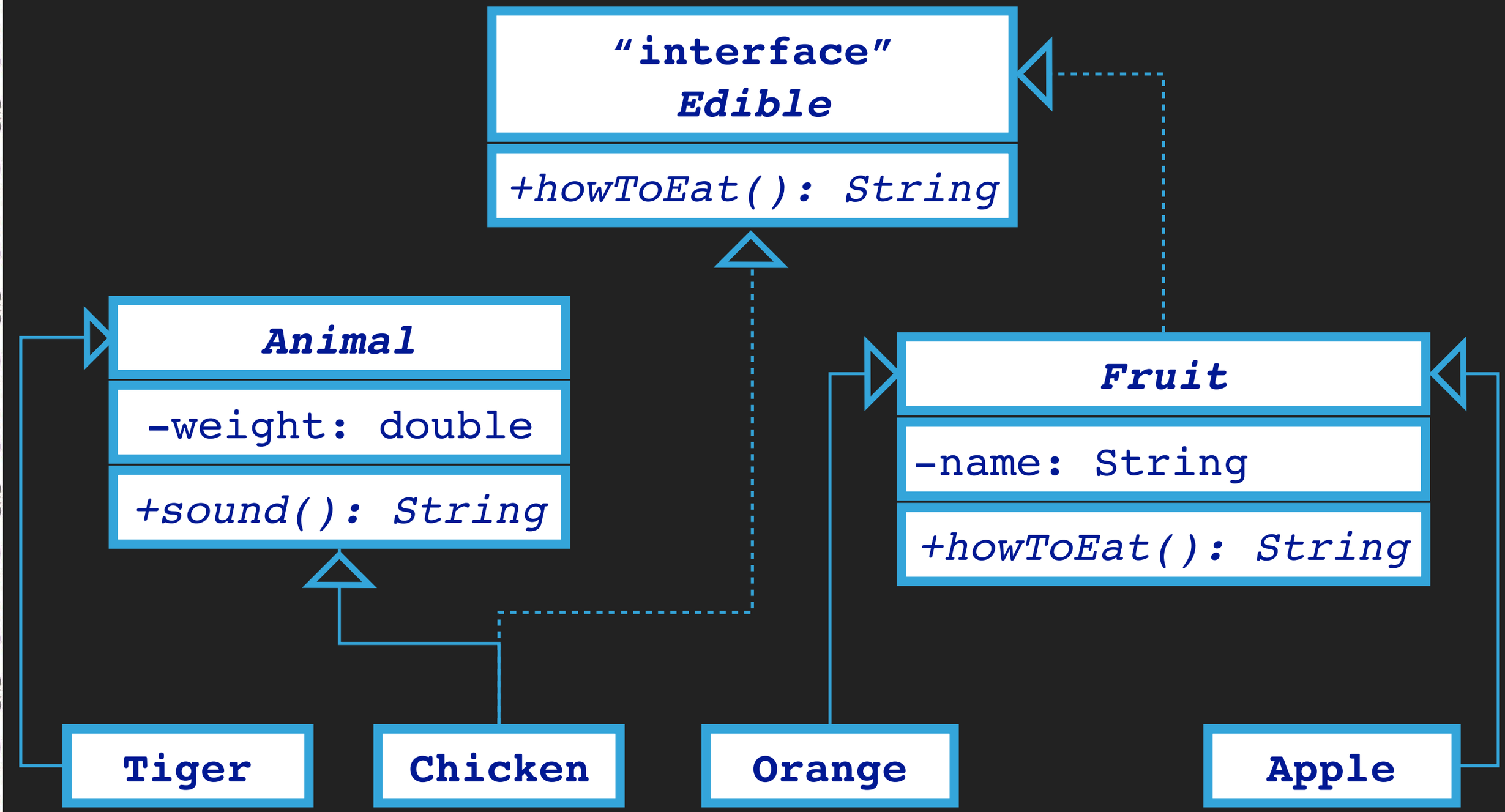
- Contains only static constants, static methods, and abstract methods

- A class **`implements`** an interface (instead of `extends`)

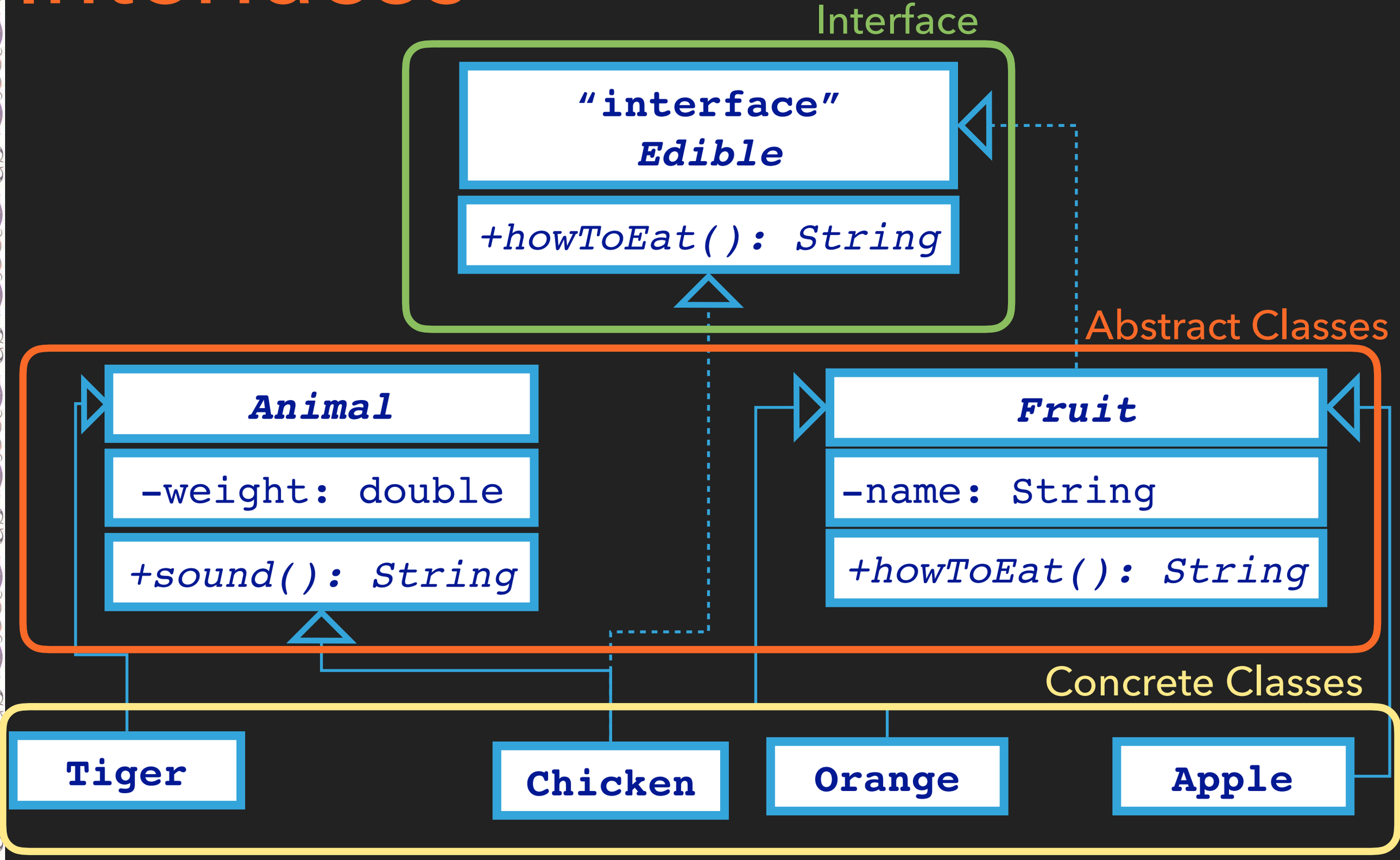
Interfaces

```
public interface InterfaceName {
   constant static declarations;
   static methods;
   abstract method signatures;
}
public interface Edible {
  // Describe how to eat any object
  public abstract String howToEat();
}
 public class Fruit implements Edible
```

Interfaces


"interface"
Edible
+howToEat(): String
Animal
-weight: double
+sound(): String
Fruit
-name: String
+howToEat(): String
Tiger
Chicken
Orange
Apple

Interfaces


Interface
"interface"
Edible
+howToEat(): String
Abstract Classes
Animal
-weight: double
+sound(): String
Fruit
-name: String
+howToEat(): String
Concrete Classes
Tiger
Chicken
Orange
Apple

Interfaces

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

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

POLYMORPHISM – INTERFACES

Interfaces

```
public interface Edible {
   // Describe how to eat
   String howToEat();
}
```

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

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

Interfaces

```
public interface Edible {
   // Describe how to eat
   String howToEat();
}
```

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

Interfaces

```
class Apple extends Fruit {
   public String howToEat() {
      return "Apple: Make apple pie”;
   }
}
```

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

Interfaces

```
public class TestInterface {
  public static void main(String[] args){
    Object[] objects = { new Tiger(), new Chicken(),
                         new Apple(), new Orange()};

    for(int i=0; i < objects.length; i++){
      System.out.println(objects[i].toString());
      if (objects[i] instanceof Edible){
       System.out.println(((Edible)objects[i]).howToEat());
      }

      if (objects[i] instanceof Animal) {
        System.out.println(((Animal)objects[i]).sound());
      }
    }
  }
}
```

Interfaces

- Abstract methods in an interface may have a default definition
- When an interface is implemented, the default definition may be used or overridden
- Example:

```
public default String howToEat() {
   return “Eat it the way you want”;
}
```

- In classes **`Chicken`**, **`Fruit`**, **`Apple`**, **`Orange`**, the default definition can be used as is or can be overridden

Interfaces - Comparable

- **`java.lang.Comparable`**: Interface to define the comparable feature between objects of any class

- The interface has only one abstract method **`compareTo()`**

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

Interfaces - Comparable

- **`int compareTo()`**
  - returns **`0`** : the two arguments are equal
  - returns **`>  0`** : the first argument comes after the second argument
  - returns **`< 0`** : the first argument comes before the second argument

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

Interfaces - Comparable

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

```
public class Circle extends Shape
          implements Comparable<Circle> {
    … //Circle constructors and methods
    // compareTo() implementation
    public int compareTo(Circle cc){
        if (radius == cc.radius) return 0;
        else if (radius > cc.radius) return 1;
        else return -1;
    }
}
```

Interfaces - Comparable

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

```
public class SortCircles {
  public static void main(String[] args) {
    Circle[] circles = {new Circle(3.4),
                        new Circle(55.4),
                        new Circle(7.67),
                        new Circle(11.54)};
    java.util.Arrays.sort(circles);
    for (Circle circle: circles)
      System.out.println(circle.toString());
  }
}
```

Interfaces - Cloneable

- Empty Interface to define the ability to be cloned for objects of any class (marker interface)

- The interface is empty and is only used to mark a class as having the cloneable feature

```
public interface Cloneable {
}
```

Interfaces - Cloneable

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

Interfaces - Cloneable

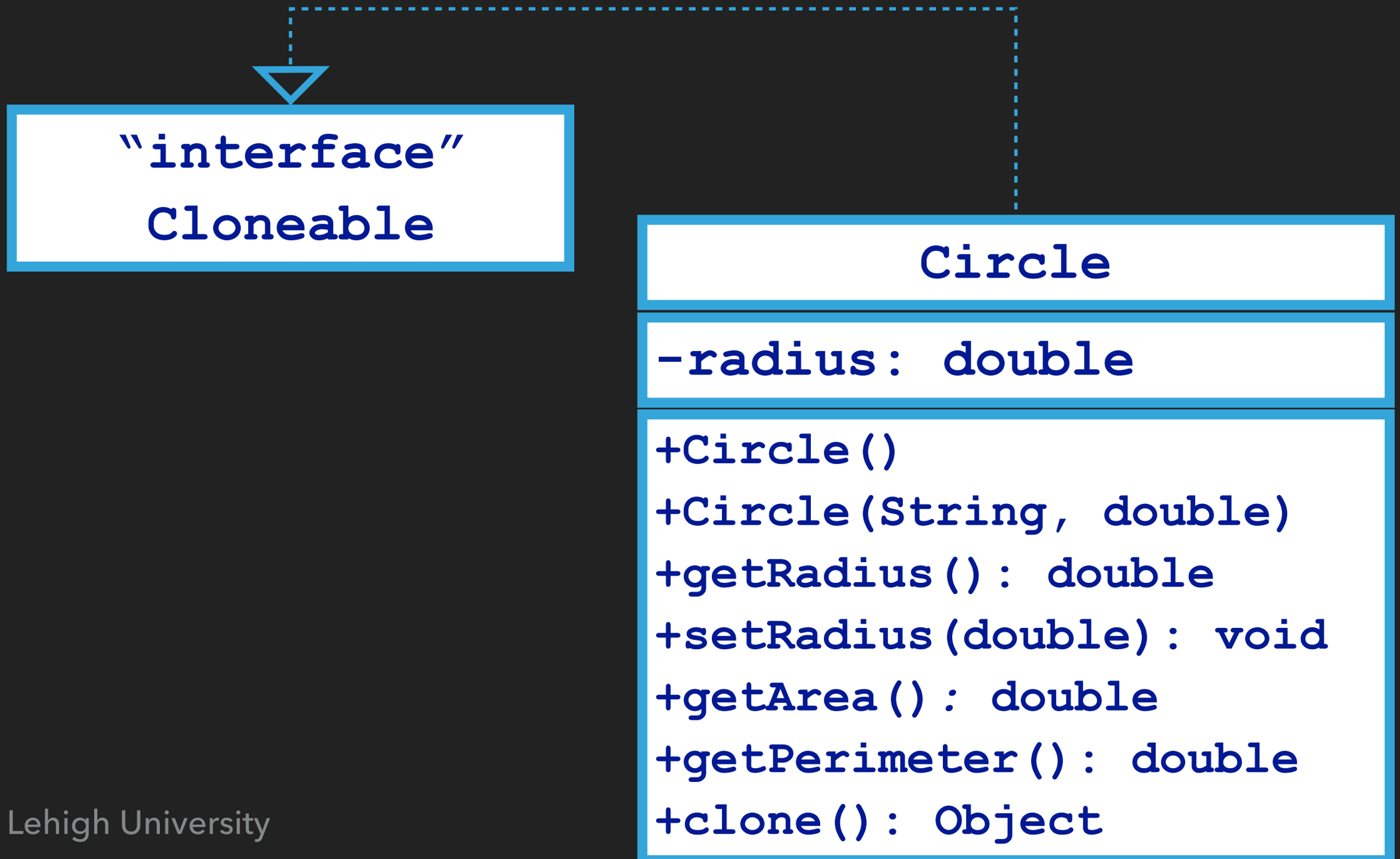
- Classes that implement the interface **`Cloneable`**:

  `Calendar`, **`Date`**

```
Calendar calendar =
        new GregorianCalendar(2020, 26, 2);
Calendar calendar1 = calendar;//shallow copy
Calendar calendar2 = //deep copy
                (Calendar) calendar.clone();
boolean equal1 = (calendar == calendar1);
boolean equal2 = (calendar == calendar2);
boolean equal3 = calendar.equals(calendar2);
System.out.println(equal1);
System.out.println(equal2);
System.out.println(equal3);
```

Interfaces - Cloneable

- Shallow copy vs. Deep copy



Interfaces - Cloneable

- Shallow copy

```
Object clone(){
    return this;
}
```

- Deep copy

```
Object clone(){
    return new Circle(this.getColor(),
                      this.getRadius());
}
```

Interfaces - Cloneable

- Using the shallow copy **`clone()`** method

```
Circle circle1 = new Circle("Blue", 5.5);
Circle circle2 = (Circle) circle1.clone();//downcast
circle2.setRadius(10.0);
System.out.println(circle1.toString());
System.out.println(circle2.toString());
```

- Using the deep copy **`clone()`** method

```
Circle circle1 = new Circle("Blue", 5.5);
Circle circle2 = (Circle) circle1.clone();
circle2.setRadius(10.0);
System.out.println(circle1.toString());
System.out.println(circle2.toString());
```

Implementing Multiple Interfaces

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

Implementing Multiple Interfaces

```
public interface Cloneable { }
```

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

```
public class Circle extends Shape
                implements Comparable<Circle>, Cloneable {
    …
    public int compareTo(Circle cc){
        if (radius == cc.radius) return 0;
        else if (radius > cc.radius) return 1;
        else return -1;
    }
    public Object clone(){
      return new Circle(radius));
    }
}
```

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, ...
- Interfaces are used as an alternative to multiple inheritance