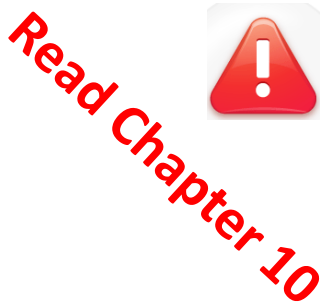
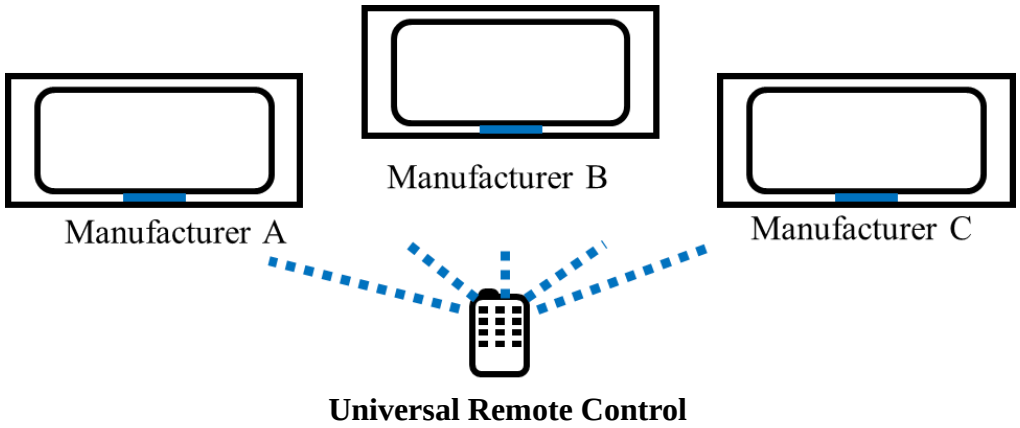
CMPS 251



Polymorphism



Dr. Abdelkarim Erradi

CSE@QU

Outline

- Polymorphism
- Abstract classes
- Interfaces

Polymorphism

Polymorphism

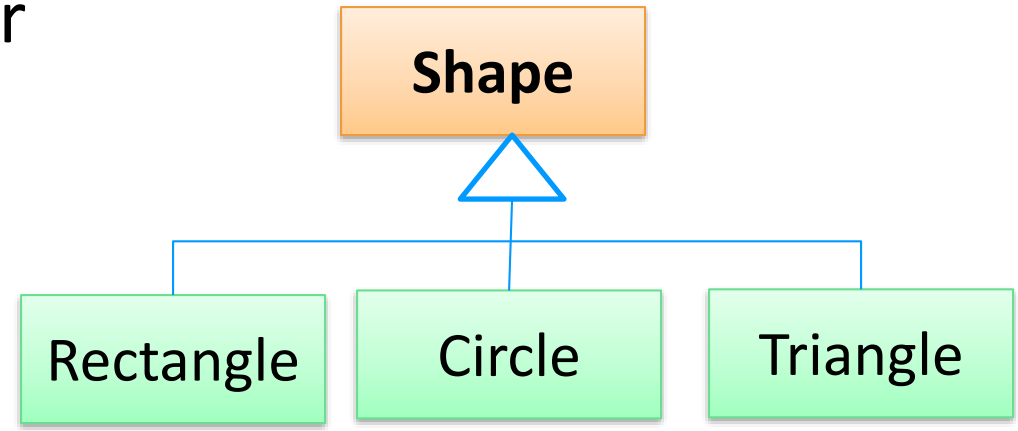
- Poly = many, morph = forms
- A way of coding generically
  - Ability to use a superclass type as ***array type***, *a* ***method parameter*** *or* a **method *return type***.
  - Ability to use variables of the superclass type to call methods on objects of subclass type
    - At execution time, **the correct subclass version of the method is called** based on the type of the referenced object.
    - The method call sent to subclasses ***has "many forms" of results*** **=>** hence the term **polymorphism**
- Polymorphism relies on **dynamic binding** (or late binding) to determine at runtime the exact implementation to call based the receiving object
  - **Dynamic binding** = figuring out which method to call **at runtime**

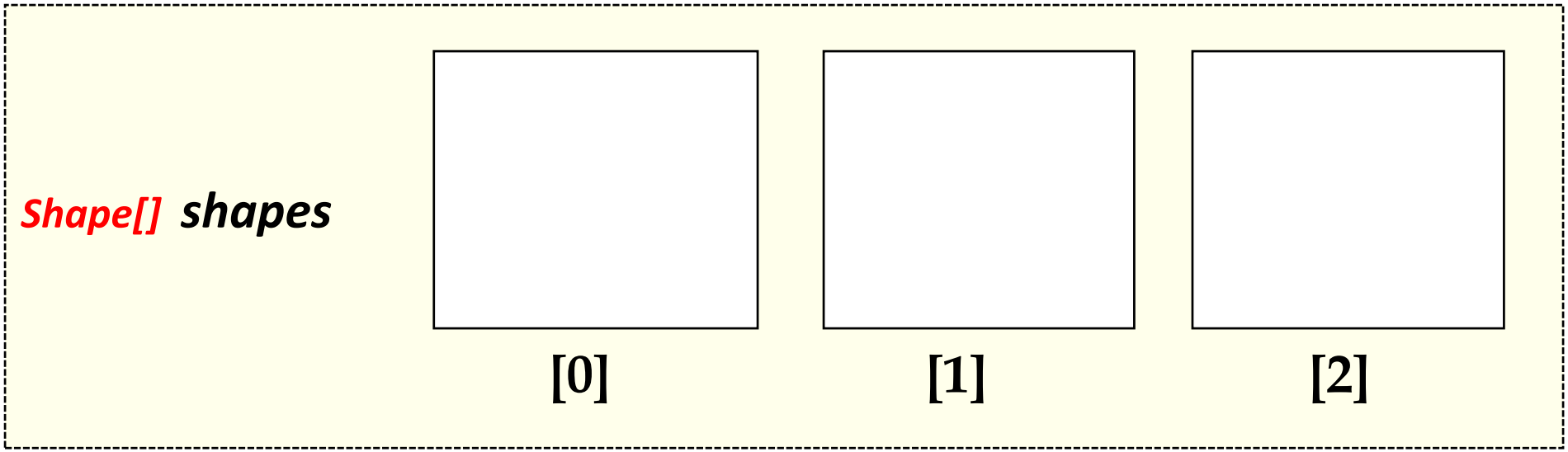
1) Using Polymorphism for Array Type

We can declare an array or ArrayList of type

Shape[] *shapes*

then put in it *rectangles*, *circles*, or *triangles*

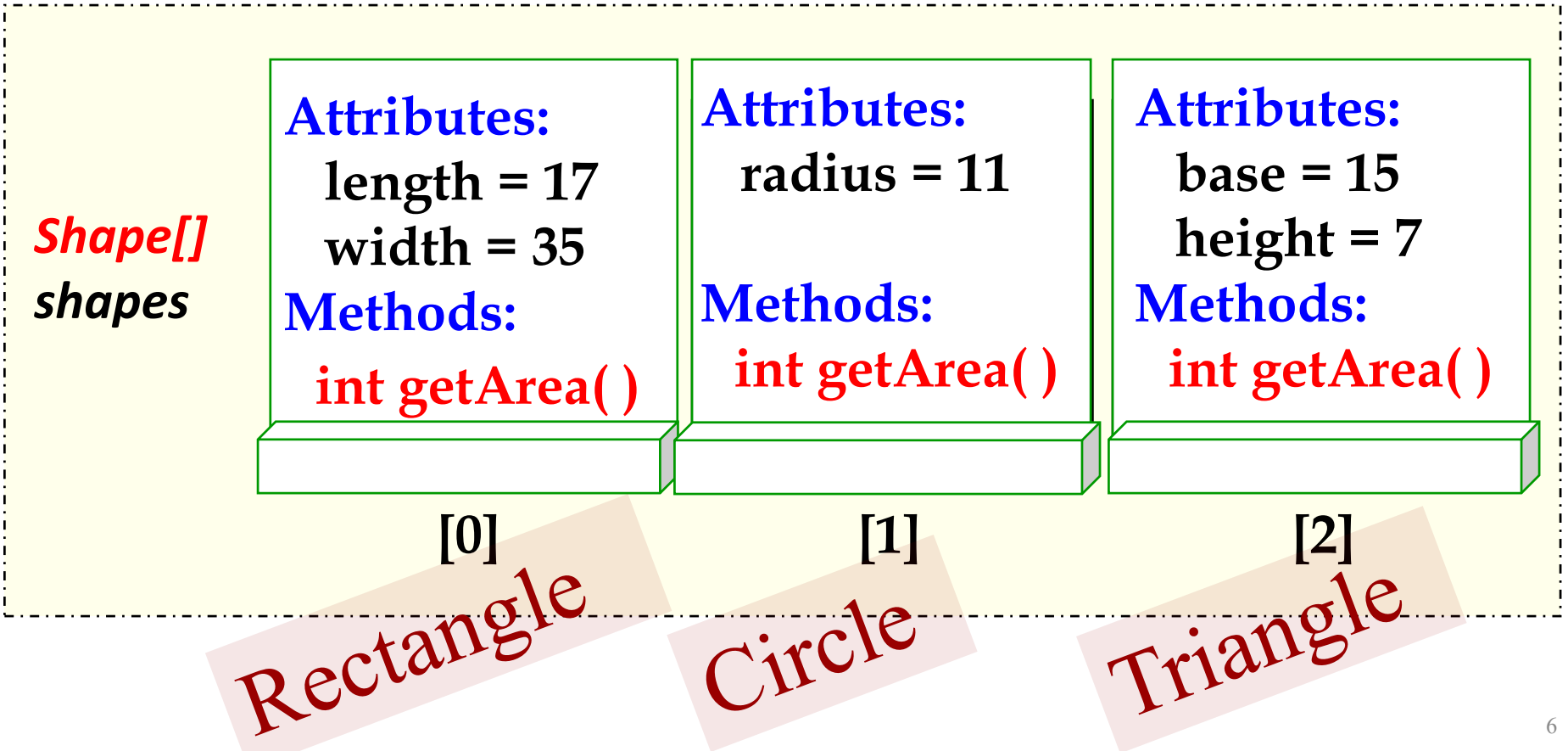




- Declaring the array using the **supertype** keeps things generic: can reference a lot of objects using one generic type

1) Using Polymorphism for Array Type

- To use polymorphism we use the **superclass Shape** as the data type of the array so that we can store in it *rectangles*, *circles*, or *triangles*.



2) Using Polymorphism for Method Parameters

- We can create a method that has **Shape** as parameter type, then use it for objects of type **Rectangle**, **Circle**, and **Triangle**
- Polymorphism allows writing **generic** code that can handle multiple types of objects, in a unified way

```
public static double getPaintCost (Shape shape) {
      int PRICE = 5;
      return PRICE * shape.getArea();
}
```

The actual definition of getArea( ) is known only at runtime, not compile time – this is "dynamic binding"

This is polymorphism! **shape** object passed in could be instance of `Circle`, `Rectangle`, or any class that **extends** Shape

Dynamic Method Binding

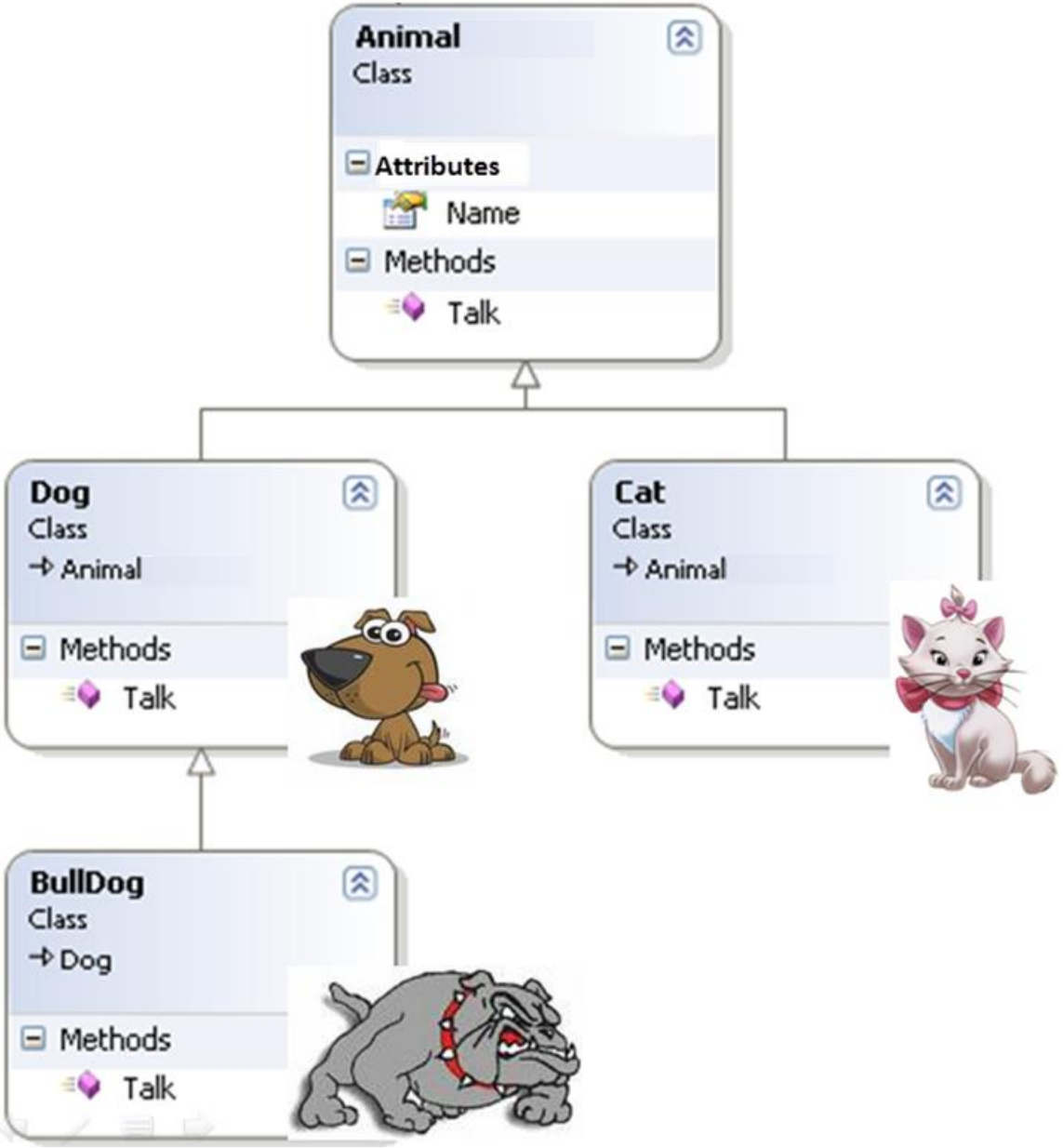
- Dynamic Method Binding
  - Actual method implementation used is not determined until runtime (e.g., the compiler does not know which getArea() method to use until the program runs)
  - Contrast with static binding, in which method gets resolved at compile time
  - For dynamic binding method **calls** using the superclass reference **get routed at runtime to the appropriate implementation** based on the type of the referenced object.
- Example
  - **`Triangle`**, **`Circle`**, and **`Square`** all subclasses of **`Shape`**. Each has an overridden **`getArea()`** method
  - When calling **`getArea()`** using the superclass reference, the program determines at runtime to appropriate implementation of **`getArea()`** method based on the type of the referenced object

3) Using Polymorphism for Method Return Type

- We can write general code, leaving the type of object to be decided at runtime

```
public Shape createShape(ShapeTypeEnum shapeType)
{
   switch (shapeType) {
      case ShapeTypeEnum.Rectangle:
           return new Rectangle(17, 35);
      case ShapeTypeEnum.Circle:
           return new Circle(11);
      case ShapeTypeEnum.Triangle :
           return new Triangle(15, 7);
   }
}
```

Polymorphism Example 2



Note that all animals have **Talk** method but the **implementation is different**:

- Cat says Meowww!
- Dog says: Arf! Arf!
- BullDog : Aaaarf! Aaaarf!

Polymorphism Example 2 (cont.)

- `Example:`
  - `Animal` array containing references to objects of the various `Animal` subclasses (Cat, Dog, etc.)
  - We can loop through the array of animals and call the method ***talk***
  - Each specific type of `Animal` does ***talk*** in a its own unique way.
  - The method call sent to a variety of objects ***has "many forms" of results*** => hence the term **polymorphism**.

Benefits

- Enables "***program in the general***" rather than "***program in the specific***"
  - This can simplify programming by writing general code that can handle multiple types of objects, **in a unified way**
- Makes it possible to **call methods with different implementations using one interface**
- **Easier to extend the program** by adding subclasses without modifying the **general portions** of the program that use the superclass type to call methods on objects of subclass type

e.g., Add a Lion class that extends Animal and provides its own talk method implementation. The generic code that manipulates the List<Animal> can invoke the Lion *talk* method

instanceof operator

- The **`instanceof`** operator is used to determine if an object is of a particular class.

  `if (shape1 instanceof Circle)`

  Returns **true** if the object to which **shape1** points "is a" **Circle**

- Every object in Java knows its own class by using the **`getClass`** method inherited from the Object class
  - The `getClass` method returns an object of type `Class`
  - To get the object's class name you can use `shape1.getClass().getName()`

Downcasting

- Attempting to invoke a subclass-only method directly on a superclass reference is a compilation error.
- A technique known as **downcasting** enables a program to invoke subclass-only methods

```
if (member instanceof Instructor)
   System.out.println(
      ((Instructor) member).getOffice());

if (member instanceof Student)
   System.out.println(
      ((Student) member).getGpa());
```

Abstract classes

Abstract Classes

- Idea
  - Use an abstract class when you want to define a **template** to guarantee that all **subclasses** in a hierarchy will have certain common methods
  - Abstract classes can contain implemented methods and **abstract methods** that are NOT implemented
- Syntax

```
public abstract class SomeClass {
    public abstract SomeType method1(…);  // No body
    public SomeType method2(…) { … } // Not abstract
}
```

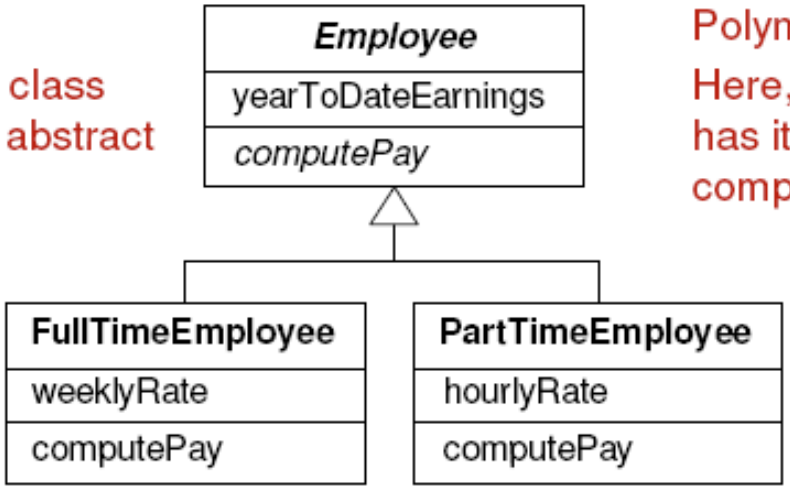
- Motivation
  - Guarantees that all subclasses will have certain methods => **enforce a common design**.
  - Lets you make collections of mixed type objects that can processed polymorphically

Abstract Classes

- An abstract class has one or more abstract methods that subclasses MUST override
  - Abstract methods do not provide implementations because they **cannot be implemented in a general way**
  - Constructors and `static` methods cannot be declared `abstract`
- An abstract class cannot be instantiated

Abstraction:

Employee is an abstract class and *computePay()* is an abstract operation (italicized)



Polymorphism:

Here, each type of Employee has its own version of computePay()

Abstract Class Example

Shape.java

```
public abstract class Shape {

    public abstract double getArea();

    public String getName() {
        return "Shape";
    }
}
```

Rectangle.java

```
public class Rectangle extends Shape{
    private double width;
    private double height;

    public Rectangle(int w, int h) {
        this.width = w;
        this.height = h;
    }

    @Override
    public double getArea() {
        double area = width * height;
        return area;
    }

    @Override
    public String getName() {
        return "Rectangle";
    }
}
```

Abstract Class Example

Shape.java

```
public abstract class Shape {

    public abstract double getArea();

    public String getName() {
        return "Shape";
    }
}
```

Circle.java

```
public class Circle extends Shape {
    private double r;

    public Circle(double r) {
        this.r = r;
    }

    @Override
    public double getArea() {
        return Math.PI * r * r;
    }

    @Override
    public String getName() {
        return "Circle";
    }
}
```

Example illustrating using Abstract Classes + Polymorphism

- You have Circle and Rectangle classes, each with getArea methods
- Goal: Get sum of areas of an array of Circles and Rectangles

=> Declare an array using an abstract class ***Shape***

```
Shape[] shapes = { new Circle(…), new Rectangle(…) … };
double areaSum =
      shapes.stream().mapToDouble(s -> s.getArea()).sum();
System.out.printf("Sum of area of all shapes %.2f ", areaSum);
```

Class Modifiers

- **Public** - publicly accessible
  - without this modifier, a class is only accessible within its own package
- **abstract** – cannot be instantiated
  - its abstract methods must be implemented by its subclass; otherwise that subclass must be declared `abstract` also
- **final** class cannot be extended (e.g., String class)
- **final** method in a superclass cannot be overridden in a subclass

Interfaces

Interfaces

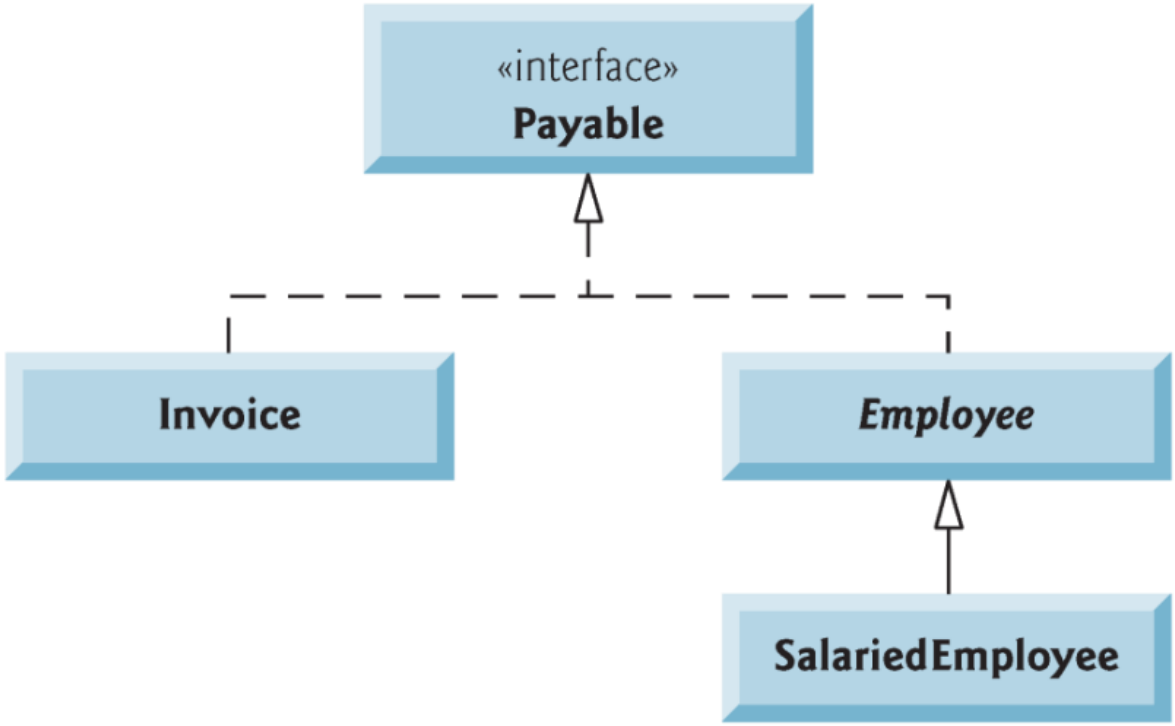
- Idea
  - **Interfaces** are used to define a set of common methods that must be implemented by **classes not related by inheritance**
  - The interface specifies ***what*** operations a class must perform but does not specify ***how*** they are performed
- Syntax

```
public interface SomeInterface {
   public SomeType method1(…);  // No body
   public SomeType method2(…);  // No body
}
public class SomeClass implements SomeInterface {
  // Real definitions of method1 and method 2
}
```

- Motivation
  - Interfaces enables requiring that **unrelated classes implement a set of common methods**
  - **Benefit from polymorphism:** objects of unrelated classes that implement a certain interface can be **processed polymorphically**

Interface Example

- A finance system has Employees and Invoices
- `Employee` and `Invoice` are not related by inheritance
- But to the company, they are both ***Payable***



Interface Example

Payable.java

```
public interface Payable {
    double getPaymentAmount();
}
```

Employee.java

```
public class Employee implements Payable{
    ...
    @Override
    public double getPaymentAmount() {
        return this.salary;
    }
    ...
}
```

Invoice.java

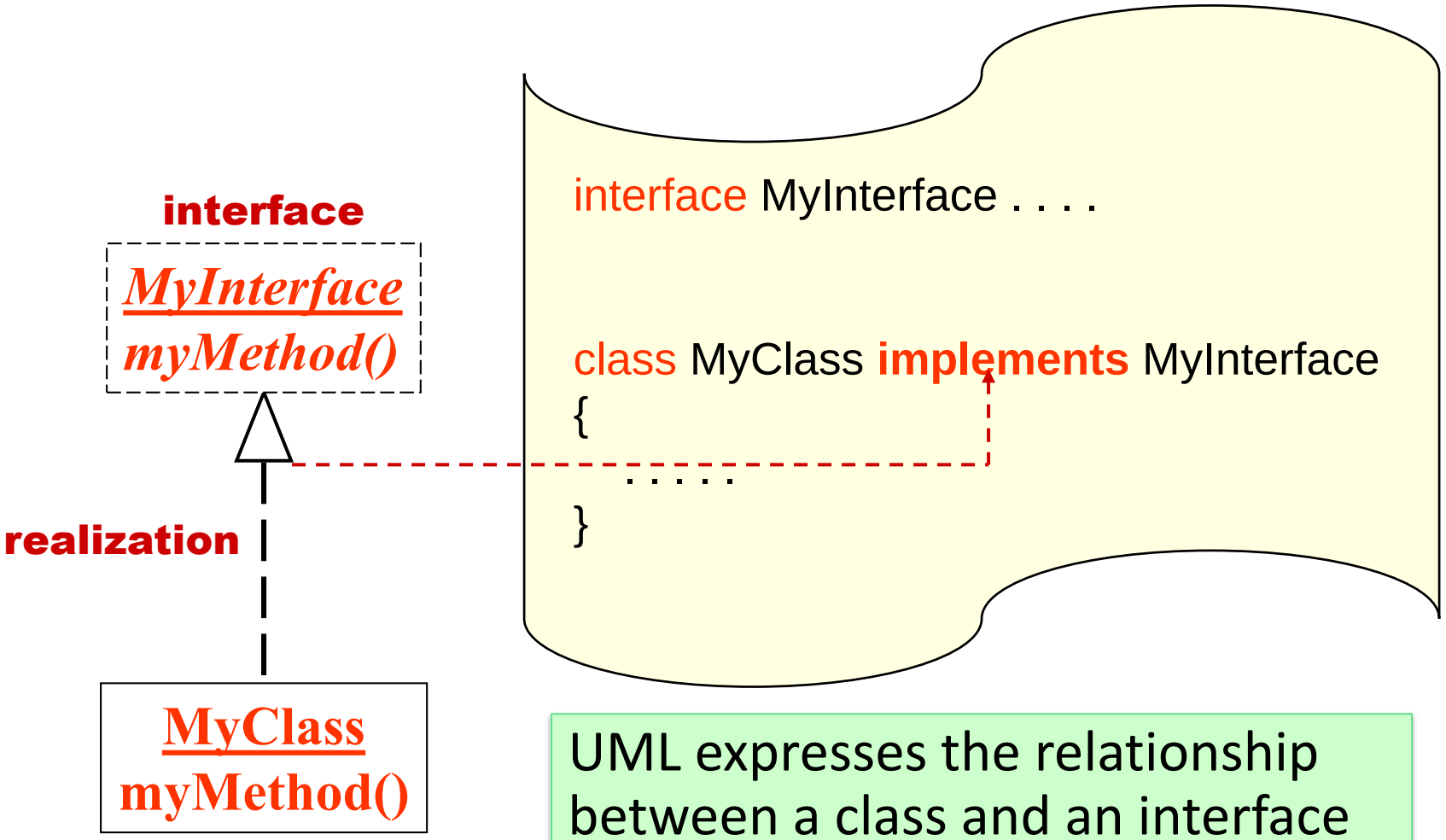
```
public class Invoice implements Payable {
    ...
    @Override
    public double getPaymentAmount() {
        return this.totalBill;
    }
    ...
}
```

Think of this *Interface!!!* implemented by ALL Living Creators (Animals and Plants) regardless of their inheritance hierarchy!

```
public interface LivingCreator {
    //"وَمَا مِنْ دَابَّةٍ فِي الْأَرْضِ إلا عَلَى اللَّهِ رِزْقُهَا"
    //القارت (الانسان)  العاشب (البقرة)  اللاحم (القط)
    void eat();

    //Crawl, swim, run, fly
    //"وَاللَّهُ خَلَقَ كُلَّ دَابَّةٍ مِنْ مَاءٍ ۖ فَمِنْهُمْ مَنْ يَمْشِي عَلَىٰ بَطْنِهِ وَمِنْهُمْ مَنْ يَمْشِي عَلَىٰ رِجْلَيْنِ وَمِنْهُمْ مَنْ يَمْشِي عَلَىٰ أَرْبَعٍ ۚ يَخْلُقُ اللَّهُ مَا يَشَاءُ ۚ"
    void move();

    //Increase in size of individual cells or in the number of cells
    //"هُوَ الَّذِي خَلَقَكُمْ مِنْ تُرَابٍ ثُمَّ مِنْ نُطْفَةٍ ثُمَّ مِنْ عَلَقَةٍ ثُمَّ يُخْرِجُكُمْ طِفْلًا ثُمَّ لِتَبْلُغُوا أَشُدَّكُمْ ثُمَّ لِتَكُونُوا شُيُوخًا"
    void grow();

    //Reproduce either from egg, pollen, sperm, etc.
    //"يَا أَيُّهَا النَّاسُ اتَّقُوا رَبَّكُمُ الَّذِي خَلَقَكُمْ مِنْ نَفْسٍ وَاحِدَةٍ وَخَلَقَ مِنْهَا زَوْجَهَا وَبَثَّ مِنْهُمَا رِجَالًا كَثِيرًا وَنِسَاءً"
    void reproduce();

    //"كُلُّ نَفْسٍ ذَائِقَةُ الْمَوْتِ"
    //Animals and Plants die in different ways
    void die();
}
```

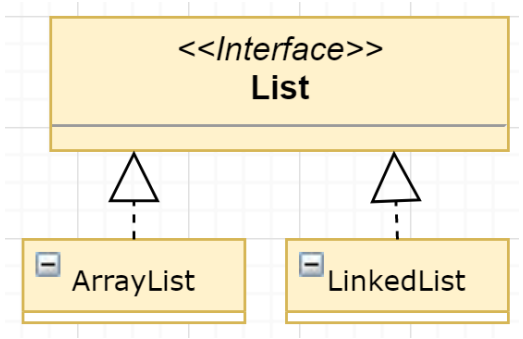
Interfaces

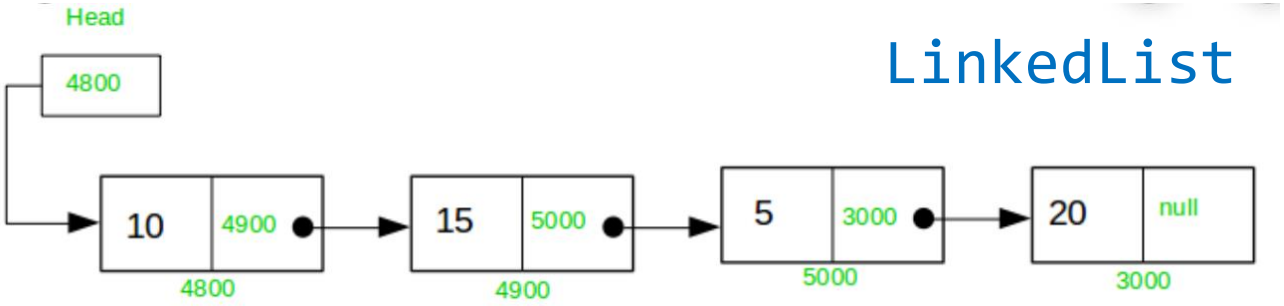
UML Notation …… Typical Java Implementation



UML expresses the relationship between a class and an interface through a **realization**.

Java Example - List Interface





- **List** = A collection that stores its elements in a sequence, and allows access to each element by its position in the sequence.
- **List** in an interface that can be assigned either an ArrayList or a LinkedList
- **ArrayList** stores its elements in an array
  - When adding an item to an ArrayList, if the underlying array is full then a new ArrayList object is created with extra 50% of current array size, and the elements are moved to this new ArrayList
- **LinkedList** is best to use when there are lots of insertions and deletions in the middle of the list

Interfaces: Declare Common Methods
Spot the Similarities!

Cars and Bikes have in common

Cars

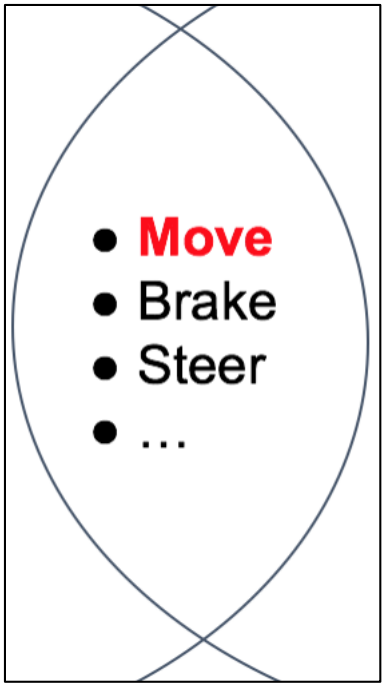
- Play radio
- Turn off/on headlights
- Turn off/on turn signal
- Lock/unlock doors
- …

- **Move**
- **Brake**
- **Steer**
- **…**

Bikes

- Drop kickstand
- Change gears
- …

Interface declares common methods



- **Interfaces** declare common methods of different classes: “lowest common denominator”
- E.g., both Car and Bike have **move** method but implemented in different ways:
  - cars drive
  - bikes pedal
- Interfaces are contracts that classes agree to **implement** (i.e., define all methods declared the in interface)

Declaring an Interface

```
public interface Transporter {

    public void move();

}
```

- Interfaces declare methods - not define them
  - Interfaces are only contracts, not classes that can be instantiated
  - ensure consistency and guarantee that classes has certain methods
- All classes that sign contract (implement this interface) must define actual implementation of declared interface methods

Creating Interfaces

- An interface declaration begins with the keyword **interface** and contains only **constants** and **abstract methods**
  - All interface members must be public
  - All methods declared in an interface are **implicitly public abstract methods**
  - All attributes are implicitly public, static and final
- A class **implementing** the interface must declare each method in the interface with specified signature

Implementing an Interface

```
public class Car implements
Transporter {

    public Car() {
        // constructor
    }

    public void drive() {
        //code for driving car
    }

    @Override
    public void move() {
        this.drive();
    }

}
```

- **Car** implements **Transporter** interface
  - declare that Car "acts-as" Transporter
- Promises compiler that Car will define all methods in Transporter interface i.e., move()
- Method ***signature*** (name and number/type of parameters) must match how it's declared in interface
- Otherwise…

"Error: Car does not override method move() in Transporter"

Implementing an Interface

```
public class Car implements Transporter {

    public Car() {
        //code elided
    }
    public void drive() {
        //code elided
    }

    @Override
    public void move() {
        this.drive();
    }
    //more methods ...
}
```

```
public class Bike implements Transporter {

    public Bike() {
        //code elided
    }
    public void pedal() {
        //code elided
    }

    @Override
    public void move() {
        this.pedal();
    }

    //more methods ...
}
```

@Override is an annotation – a signal to the compiler to enforce that the interface actually has the method declared

Implementing Multiple Interfaces

- Classes can implement **multiple** interfaces
  - "I signed my rent agreement, so I'm a renter, but I also signed my employment contract, so I'm an employee. I'm the same person."
  - The Car can implement both the Transporter and the Colorable interface
- Class implementing interfaces must define every single method from each interface

```
public interface Colorable {

    public void setColor(Color c);
    public Color getColor();

}

public class Car implements Transporter, Colorable {

    public Car(){ //body ... }

    public void drive(){ //body ... }
    public void move(){ //body ... }
    public void setColor(Color c){ //body ... }
    public Color getColor(){ //body ...  }
}
```

Polymorphism Using interfaces

- A way of coding generically
  - way of referencing many related objects as one generic type
    - cars and bikes can both `move()` → refer to them as **`Transporter`** objects
    - phones and Teslas can both `getCharged()` → refer to them as *`Chargeable`* objects, i.e., objects that implement **`Chargeable`** interface
    - Emplyees and invoices can both getPaymentAccount() → refer to them as *`Payable`* objects

```
for ( Payable payable : payables ) {
        payable.getPaymentAmount();
}
```

default Interface Methods

- Interfaces also may contain `public default methods` with concrete default implementations used when an implementing class does not override the methods.
- To declare a default method, place the keyword **default** before the method's return type and provide a concrete method implementation.
- Any class that implements the original interface will not break when a default method is added.
  - The class simply receives the new default method.
- Interfaces can also have static methods.

Abstract Class vs. Interface

- Abstract classes and interfaces cannot be instantiated
- Abstract classes and interfaces may have abstract methods that must be implemented by the subclasses
- Classes that implement an interface **can be from different inheritance hierarchies**
  - An interface is often used when unrelated classes need to provide **common methods** or use common constants
  - When a class implements an interface, it establishes an ***IS-A*** relationship with the interface type. Therefore, interface references can be used to invoke polymorphic methods just as an abstract superclass reference can.
- Concrete subclasses that extend an abstract superclass are **all related to one other by inheriting from a shared superclass**
- Interfaces cannot define instance attributes and constructors
  - Interfaces can have abstract methods, methods with a default implementation, static methods and static constants.
- Classes can extend only ONE abstract class but they may implement more than one interface

Summary

- Inheritance = “factor out” the common attributes and methods and place them in a single superclass

  => Removing code redundancy will result in a smaller, more flexible program that is easier to maintain.

- Interfaces are contracts, can’t be instantiated
  - force classes that implement them to define specified methods
- Polymorphism allows for generic code by using superclass/interface type variables to manipulate objects of subclass type
  - make the client code more **generic** and ease extensibility