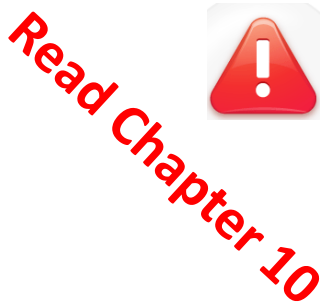
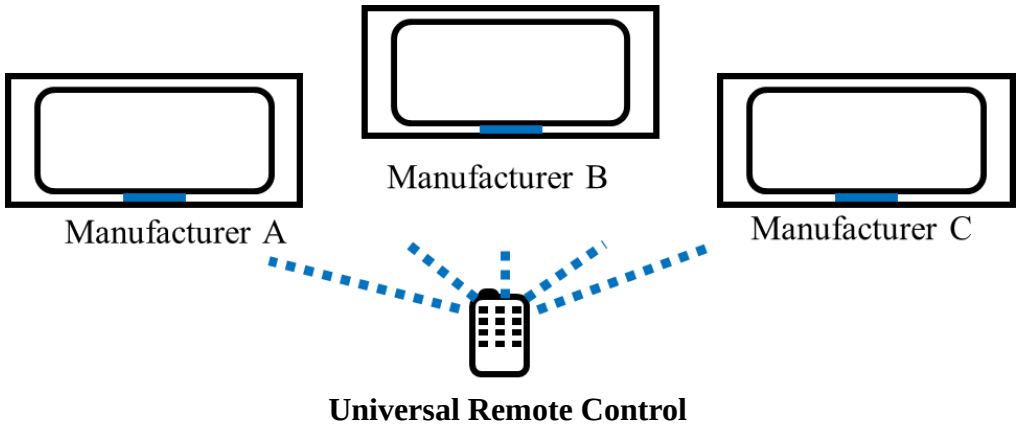
CMPS 251



Polymorphism



Dr. Abdelkarim Erradi

CSE@QU

Outline

- Polymorphism
- Abstract classes
- Interfaces

Polymorphism

Polymorphism

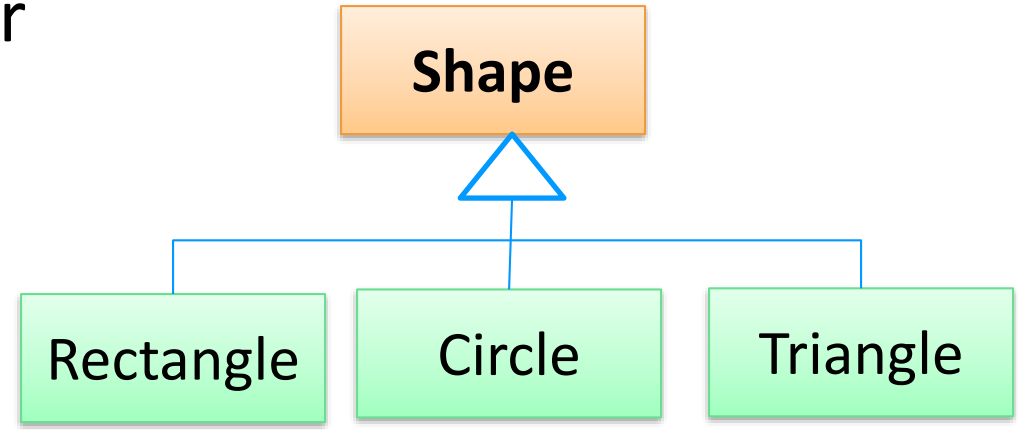
- Ability to use a superclass type as ***array type***, *a* ***method parameter*** *or* a **method *return type***.
- Ability to use variables of the superclass type to call methods on objects of subclass type
  - At execution time, **the correct subclass version of the method is called** based on the 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**

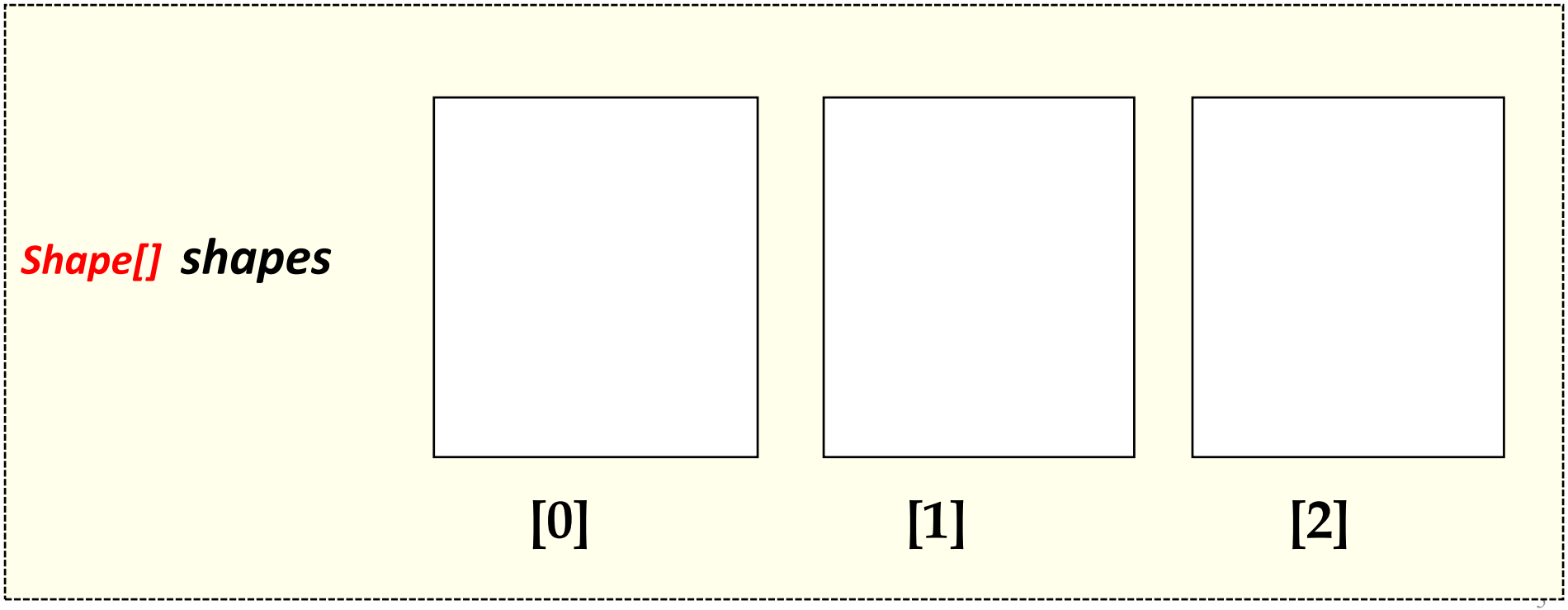
1) Using Polymorphism for Array Type

We can declare an array or ArrayList of type

Shape[] *shapes*

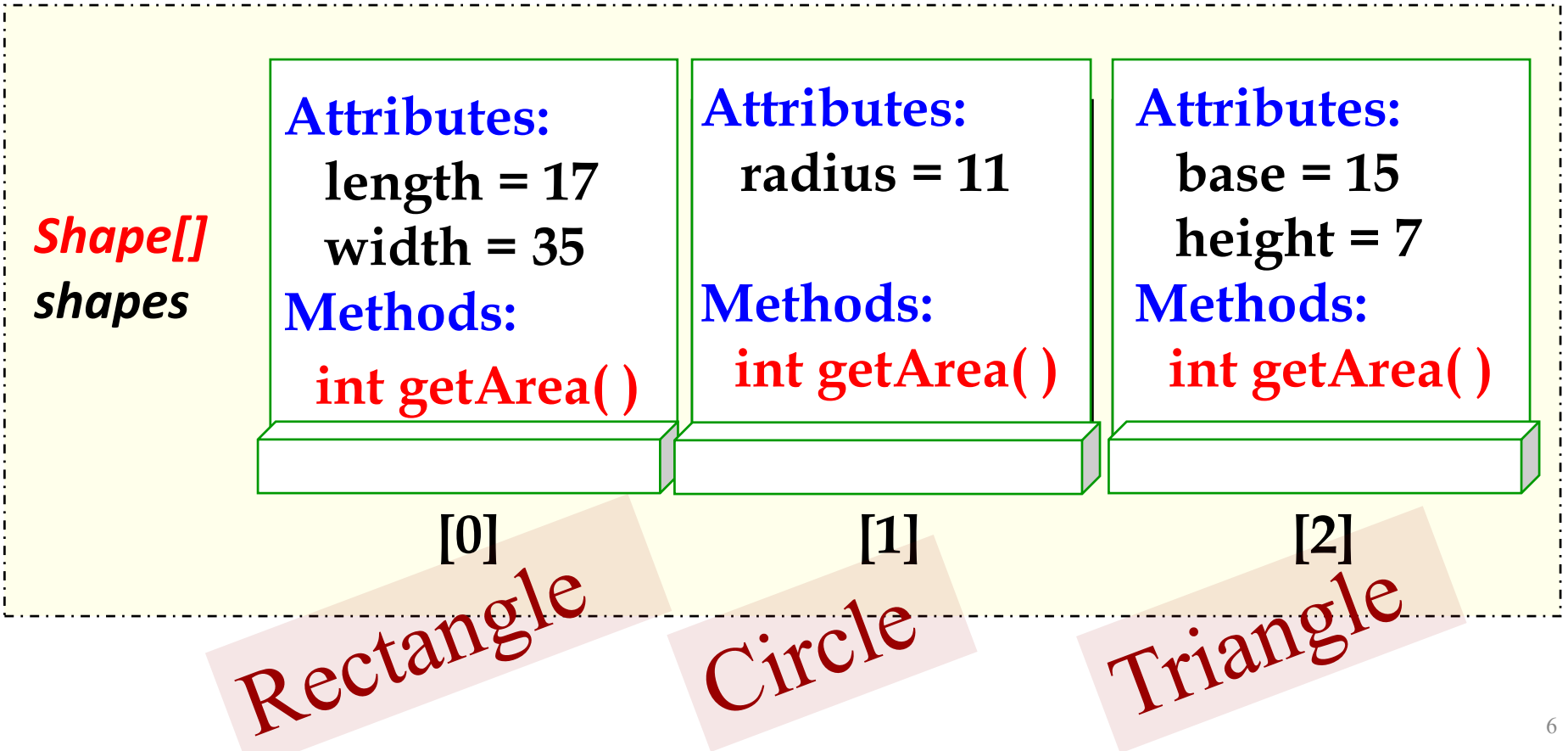
then put in it *rectangles*, *circles*, or *triangles*





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"

Dynamic Method Binding

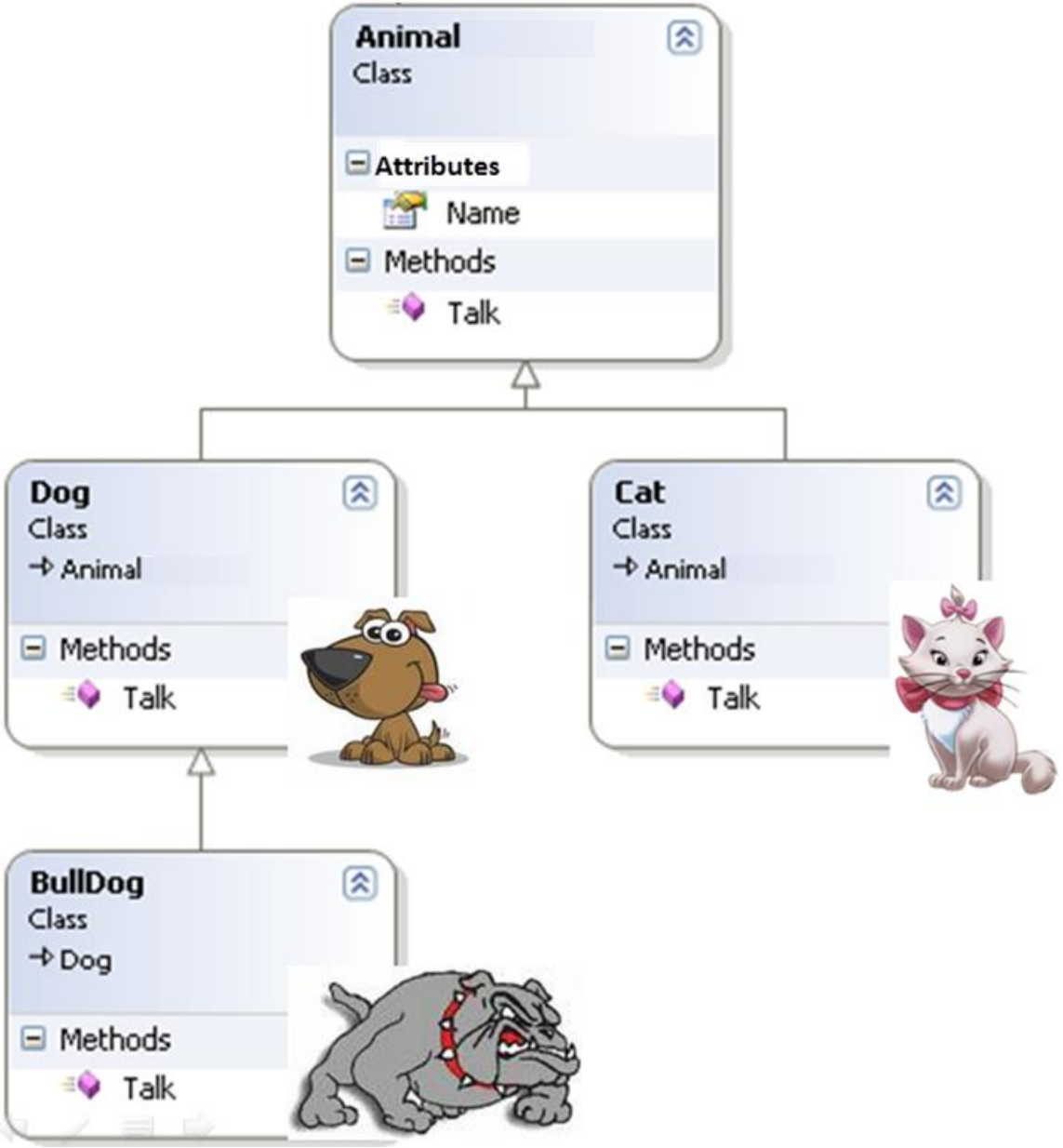
- Dynamic Method Binding
  - At runtime, method **calls** using the superclass reference **get routed to appropriate implementation** based on the type of the referenced object.
- Example
  - **`Triangle`**, **`Circle`**, and **`Square`** all subclasses of **`Shape.`** Each has an overridden **`getArea()`** 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
}
```

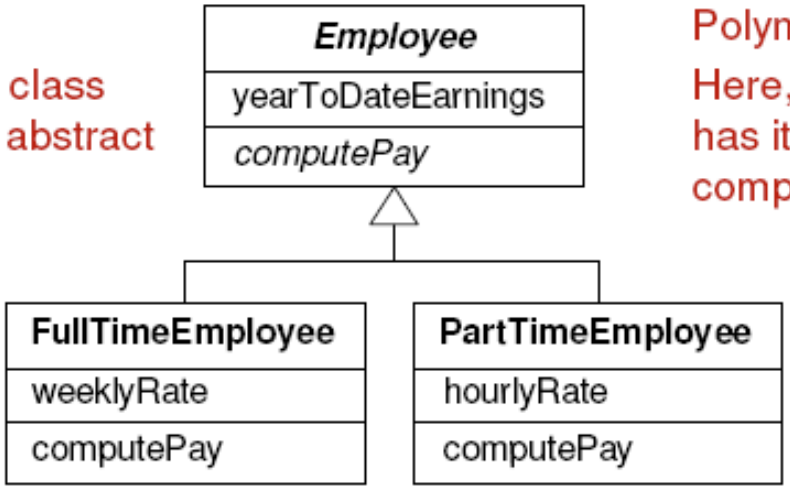
- Motivation
  - Guarantees that all subclasses will have certain methods => **enforce a common design**.
  - Lets you make collections of mixed type objects that can processed polymorphically

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

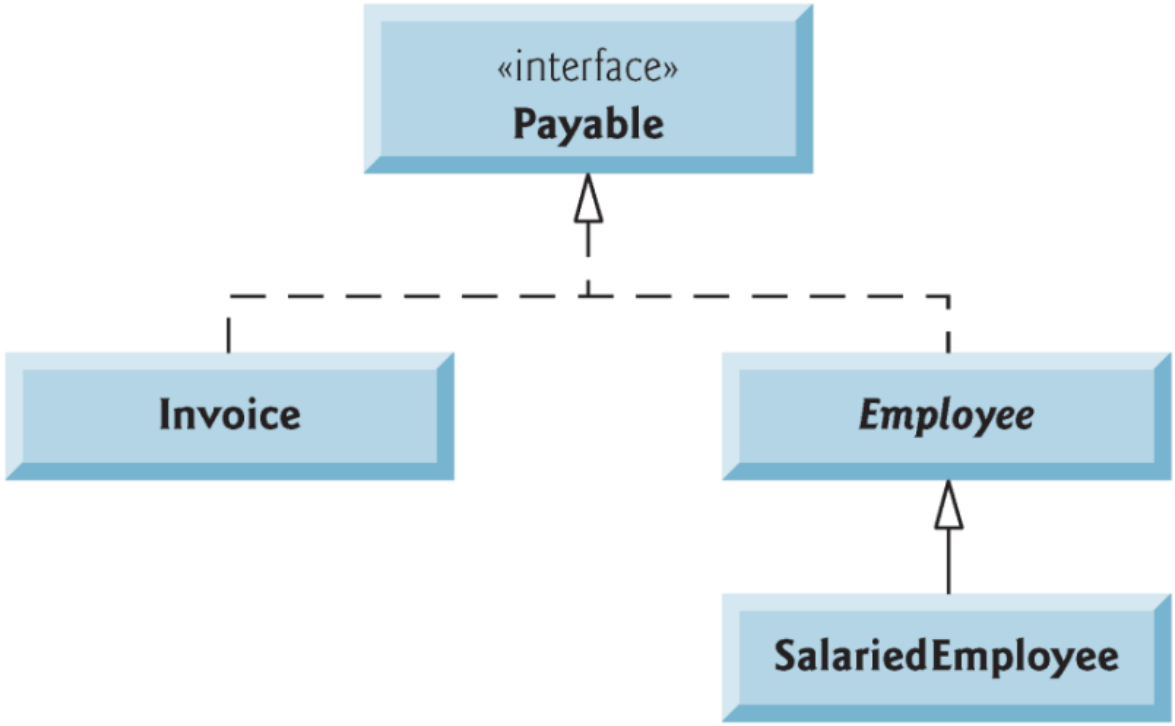
- Idea
  - **Interfaces** are used to define a set of common methods that must be implemented by **classes not related by inheritance**
  - The interface specifies ***what*** operations a class must perform but does not specify ***how*** they are performed
- Syntax

```
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

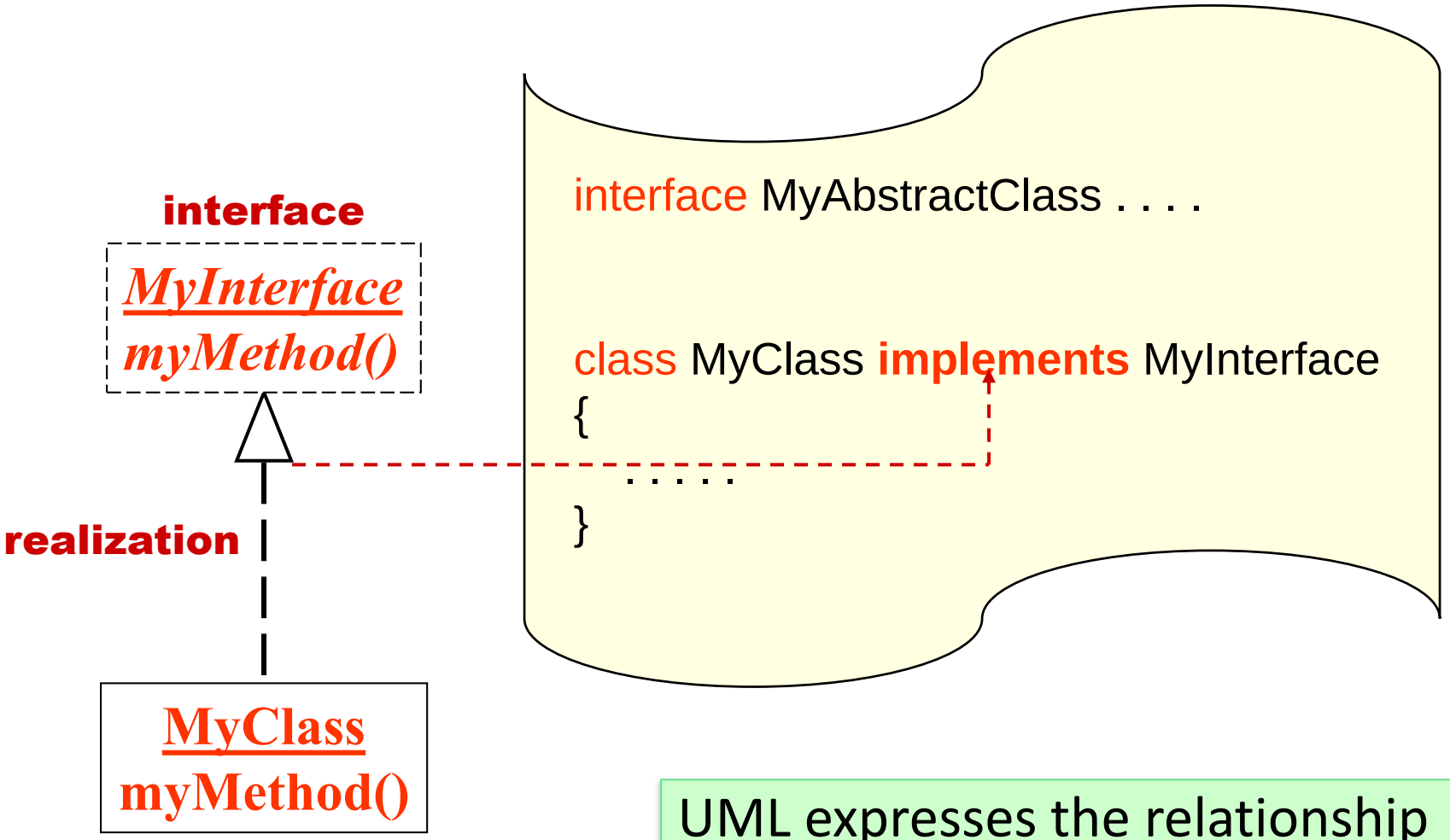
```
public class Employee implements Payable{
    ...
    @Override
    public double getPaymentAmount() {
        return this.salary;
    }
    ...
}
```

Invoice.java

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

Interfaces

UML Notation …… Typical Java Implementation



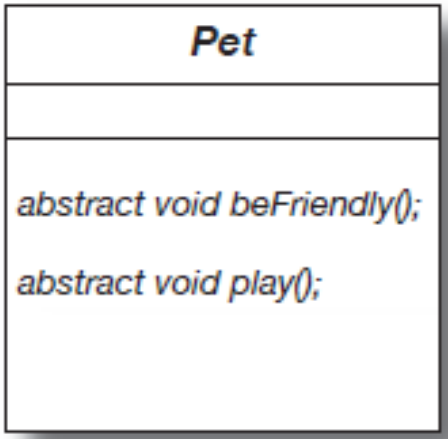
UML expresses the relationship between a class and an interface through a **realization**.

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 رزْقُهَا"
    //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();
}
```

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



A Java interface is like a 100% pure abstract class.

All methods in an interface are abstract, so any class that IS-A Pet MUST implement (i.e. override) the methods of Pet

To DEFINE an interface:

```
public interface Pet {...}
```

Use the keyword "interface" instead of "class"

To IMPLEMENT an interface:

```
public class Cat extends Animal implements Pet {...}
```

Use the keyword "implements" followed by the interface name. Note that when you implement an interface you still get to extend a class

Making and Implementing the Pet interface

interface methods are implicitly public and abstract, so typing in 'public' and 'abstract' is optional (in fact, it's not considered 'good style' to type the words in, but we did here just to reinforce it

You say 'interface' instead of 'class' here

```
public interface Pet {

    public abstract void beFriendly();

    public abstract void play();

}
```

All interface methods are abstract, so they MUST end in semicolons. Remember, they have no body!

Cat IS-A Animal and Cat IS-A Pet

You say 'implements' followed by the name of the interface.

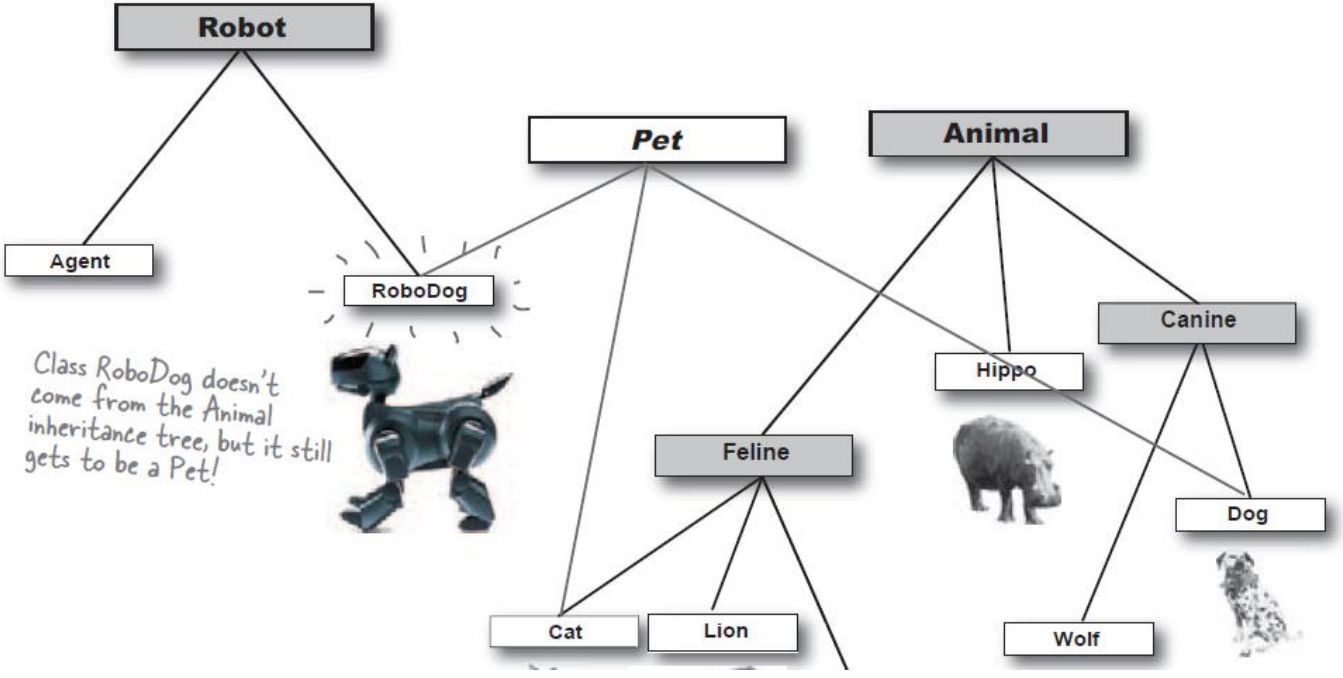
```
public class Cat extends Animal implements Pet {

    public void beFriendly() {...}

    public void play() {..}


    public void roam() {...}

    public void eat() {...}

}
```

You SAID you are a Pet, so you MUST implement the Pet methods. It's your contract. Notice the curly braces instead of semicolons.

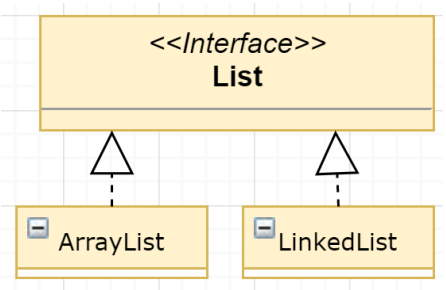
These are just normal overriding methods.

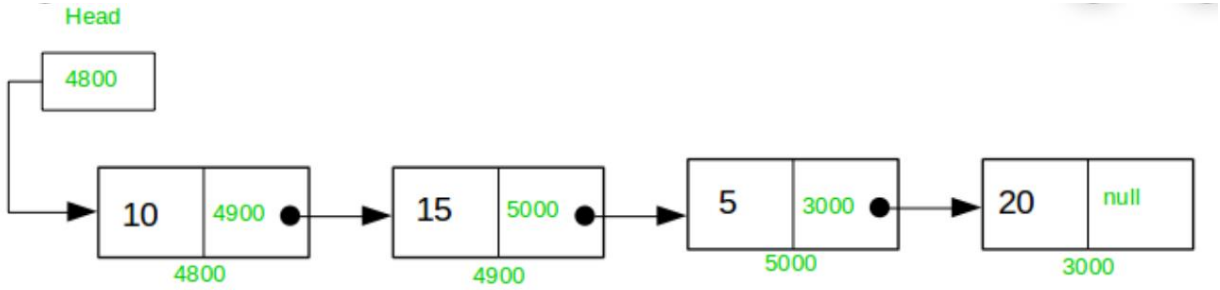
Why interface instead of abstract class?

- Classes can directly extend only one class (abstract or otherwise)
- Classes can **implement many interfaces**
- **Classes from different inheritance hierarchies can implement the same interface**



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
- 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
- A LinkedList is best to use when there are lots of insertions and deletions in the middle of the list

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

- Using inheritance, we can “factor out” the common attributes and methods and place them in a single superclass.

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

- Polymorphism takes inheritance one step further by using superclass/interface type variables to manipulate objects of subclass type
  - This make the client code more **generic** and ease extensibility