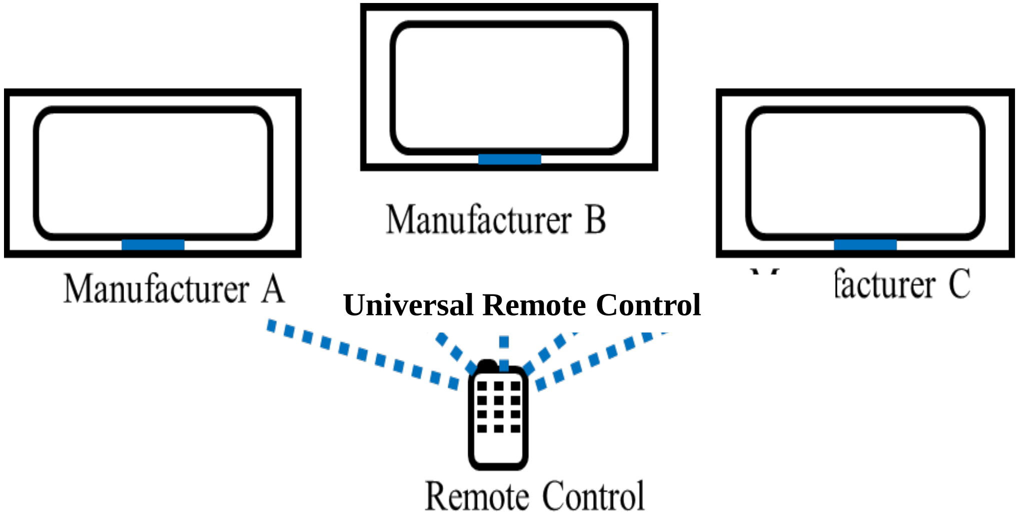
CMPS 251

Read Chapter 10

Lecture 09

Polymorphism



Universal Remote Control

Agenda

- How to use polymorphism
- Polymorphism in methods
- Plymorphism with data member (instance variables)
- Compile-time polymorphism and Run-time polymorphism
- Upcasting
- Downcasting
- Static binding and dynamic binding
- Use of java instanceof operator
- Downcasting using instanceof operator

Polymorphism

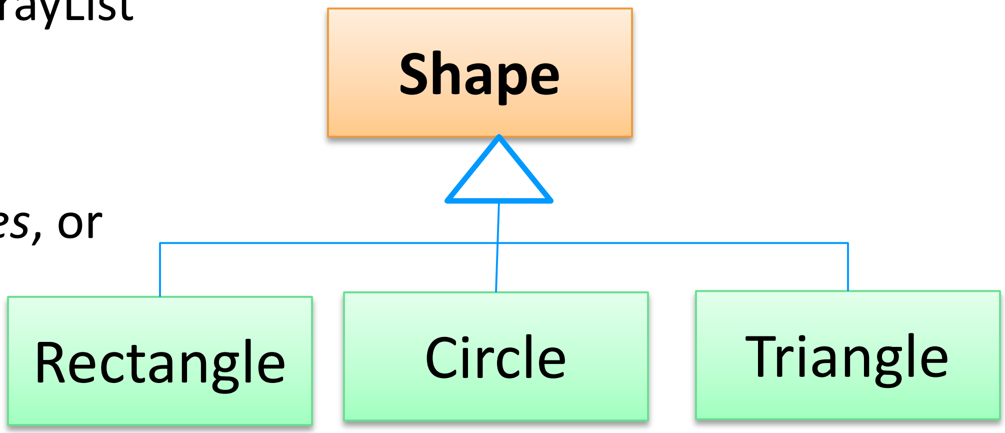
- Poly = many, morph = forms
- A way of coding generically
  - Inheritance lets us inherit attributes and methods from another class.
  - Polymorphism uses those methods to perform different tasks.
  - This allows us to perform a single action in different ways.
  - Ability to use variables of the superclass 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**

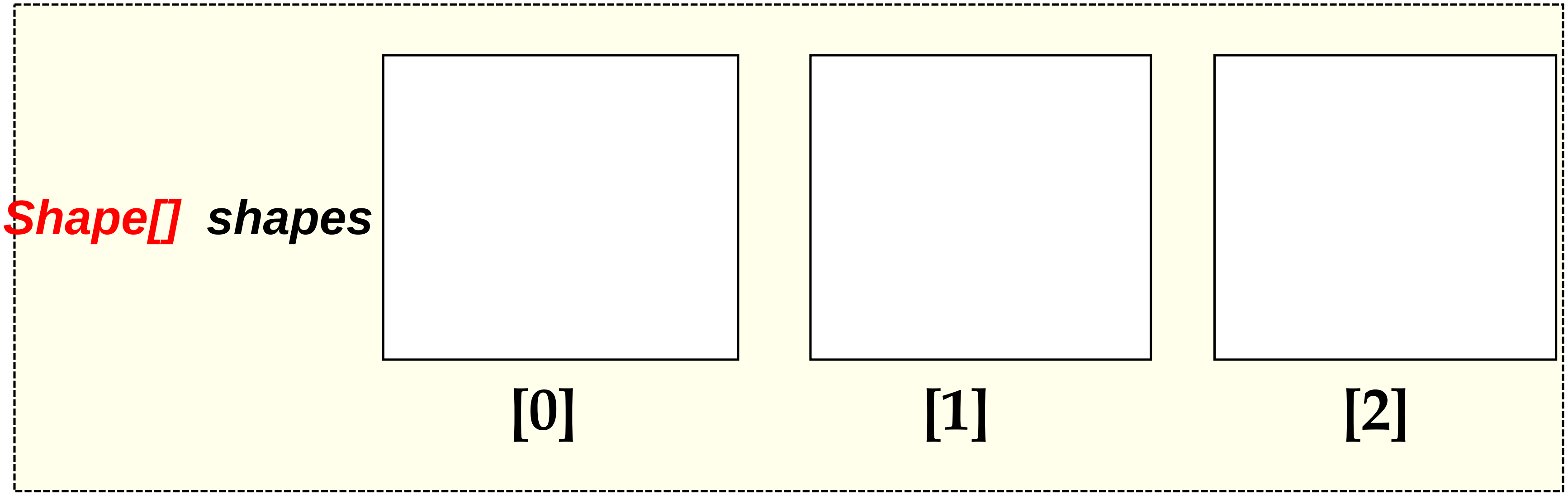
(1) Using Polymorphism for Array Type

We can declare an array or ArrayList of type

Shape[] *shapes*

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

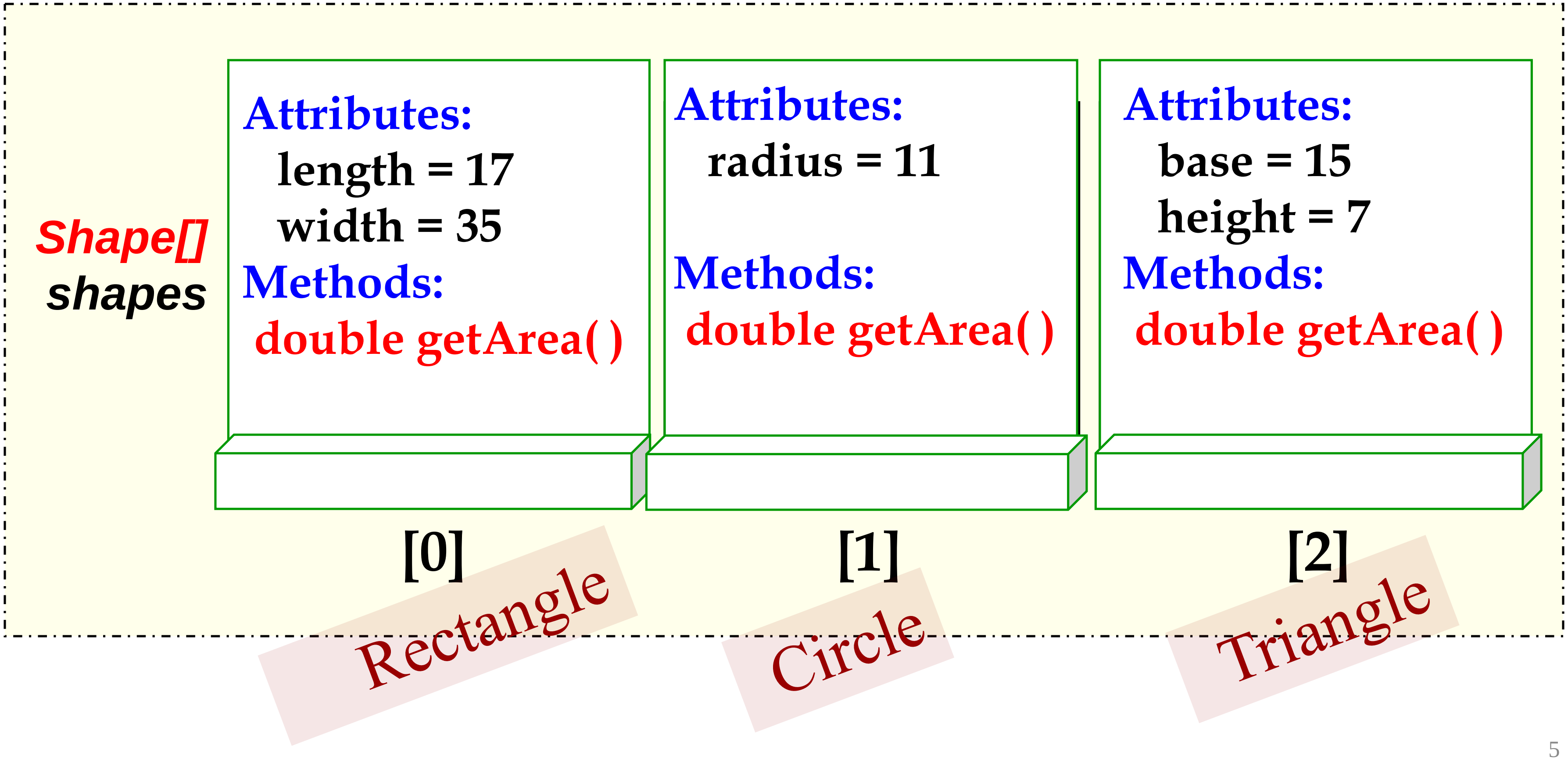




- Declaring the array using the **supertype** keeps things generic: can reference a lot of objects using one generic type

(2) 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*.



Shape[]
shapes

Attributes:
length = 17
width = 35
Methods:
double getArea( )

[0]

Rectangle

Attributes:
radius = 11

Methods:
double getArea( )

[1]

Circle

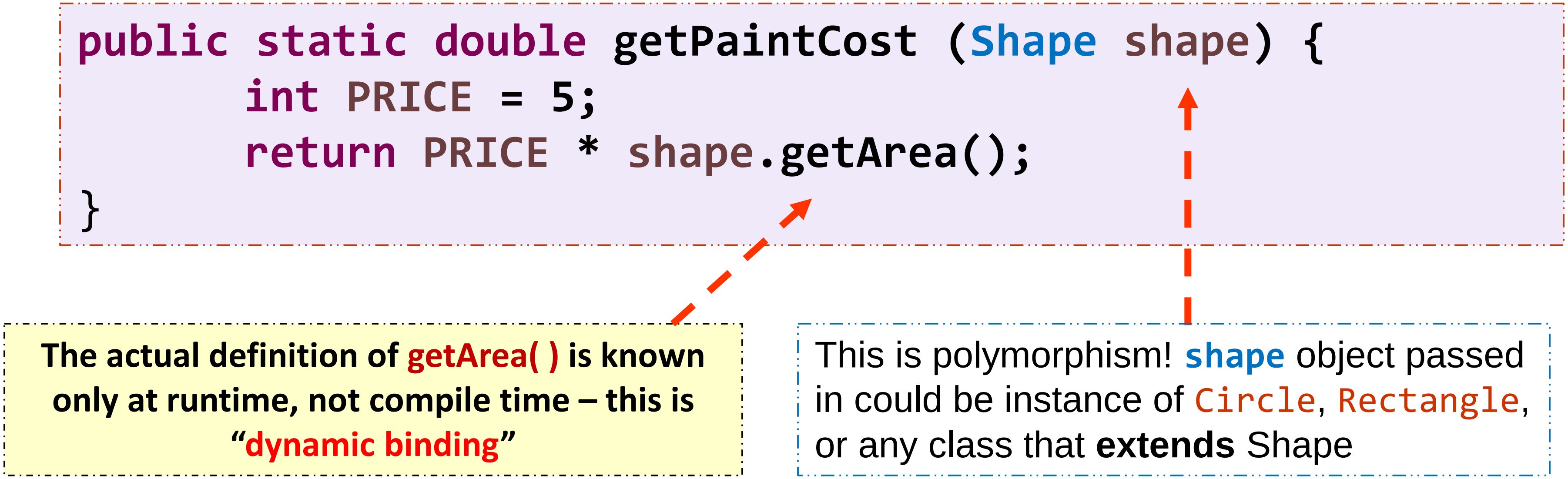
Attributes:
base = 15
height = 7
Methods:
double getArea( )

[2]

Triangle

(3) 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”

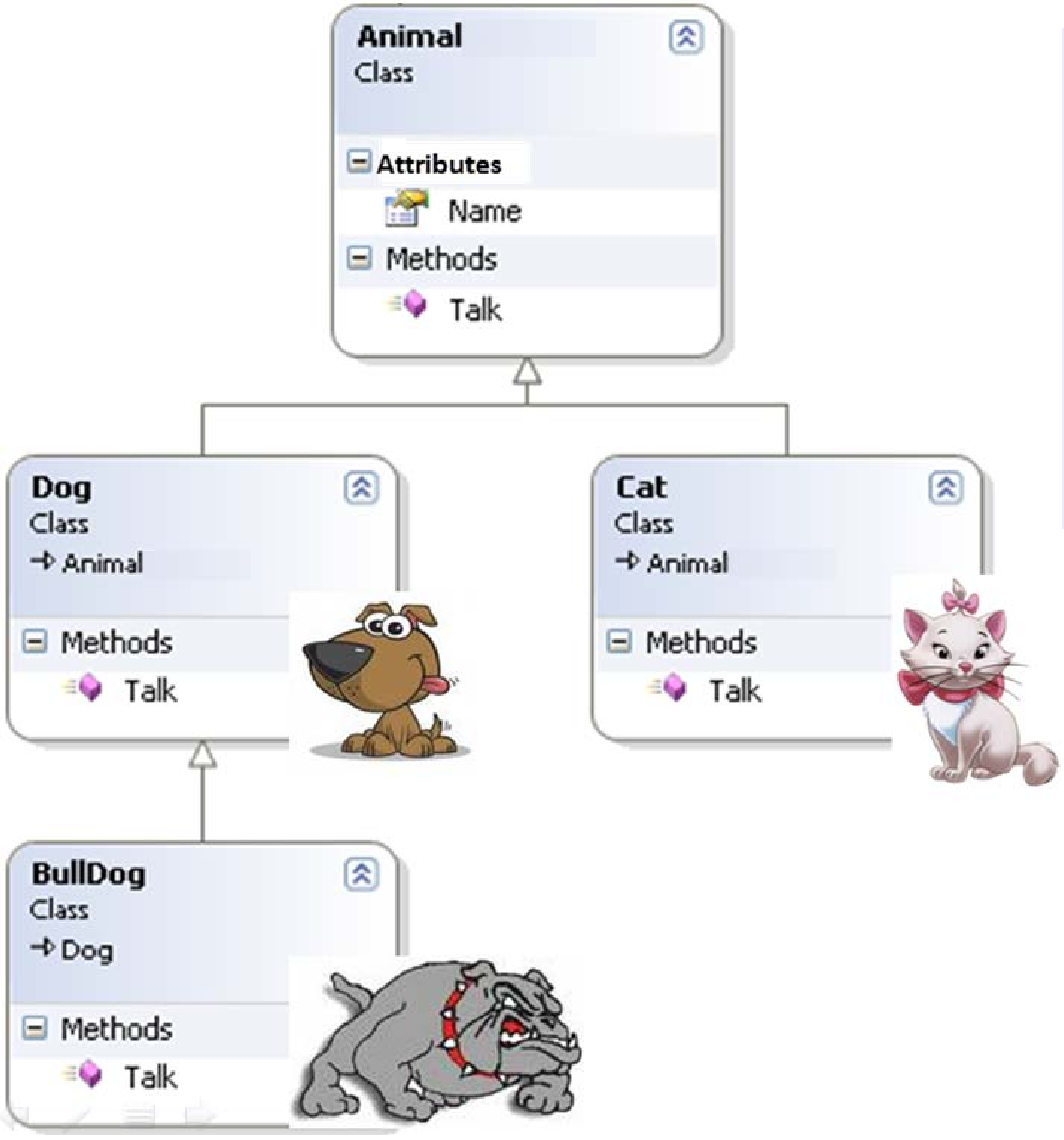
This is polymorphism! **shape** object passed in could be instance of `Circle`, `Rectangle`, or any class that **extends** Shape

(4) Using Polymorphism for Method Return Type

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

```
public Shape createShape(String shapeType) {
    switch (shapeType) {
        case “Rectangle”:
              return new Rectangle(17, 35);
        case “Circle”:
              return new Circle(11);
        case “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**.

Example: Polymorphism in Methods

```
class Animal {
  public void animalSound() {
    System.out.println("The animal makes a sound");
  } }
Class Cat extends Animal {
  public void animalSound() {
    System.out.println("The cat says: meow meow");
  } }
class Dog extends Animal {
  public void animalSound() {
    System.out.println("The dog says: bow wow");
  } }
class Main {
  public static void main(String[] args) {
    Animal myAnimal = new Animal();  // Create a Animal object
    Animal myCat = new Cat();  // Create a Cat object
    Animal myDog = new Dog();  // Create a Dog object
    myAnimal.animalSound();
    myCat.animalSound();
    myDog.animalSound();
  }  }
```

- Superclass Animal has a method called animalSound().
- Subclasses of Animals could be Cats, Dogs.
- They also have their own implementation of an animal sound

Types of Polymorphism

- There are two types of polymorphism in Java:
  - compile-time polymorphism, and
  - runtime polymorphism.
- We can perform polymorphism in java by
  - method overloading, and
  - method overriding.
- If you overload a static method in Java, it is the example of compile time polymorphism.
- Here, we focus on runtime polymorphism in java.
- **Runtime polymorphism** or **Dynamic Method Dispatch** is a process in which a call to an overridden method is resolved at runtime rather than compile-time.
- In this process, an overridden method is called through the reference variable of a superclass.
- The determination of the method to be called is based on the object being referred to by the reference variable.
- Let's first understand the upcasting before Runtime Polymorphism.

Upcasting

- If the reference variable of Parent class refers to the object of Child class, it is known as upcasting.
- Consider this example:

```
class A{}
class B extends A{}
A a = new B(); //upcasting
```

- This reference variable **a** can access all the methods and variables of class **A** but only overridden methods in child class **B**.

- Since method invocation is determined by the JVM not compiler, it is known as runtime polymorphism.

Upcasting Example

```
class Bike{
  void run(){System.out.println("running");}
}
class Splendor extends Bike{
 void run(){System.out.println("running safely with 60km");}

  public static void main(String args[]){
    Bike b = new Splendor();//upcasting
    b.run();
  }
}

//running safely with 60km
```

- In this example, we are creating two classes Bike and Splendor.
- Splendor class extends Bike class and overrides its run() method.
- We are calling the run method by the reference variable **b** of Parent class.
- Since it refers to the subclass object and subclass method overrides the Parent class method, the subclass method is invoked at runtime.

Example of Runtime Polymorphism

```
class Bank{
int getDepositTime(){return 0;}
}
class SBI extends Bank{
int getDepositTime(){return 2;}
}
class ICICI extends Bank{
int getDepositTime(){return 3;}
}
class AXIS extends Bank{
int getDepositTime(){return 1;}
}
class TestPolymorphism{
public static void main(String args[]){
Bank b;
b = new SBI();  // Bank b = new SBI();
System.out.println("SBI deposit time: "+b.getDepositTime());
b=new ICICI();
System.out.println("ICICI deposit time: "+b.getDepositTime());
b=new AXIS();
System.out.println("AXIS deposit time: "+b.getDepositTime());
}
}
```

Java Runtime Polymorphism with Data Member

- A method is overridden, not the data members, so runtime polymorphism can't be achieved by data members.
- In the example given below, both the classes have a data member speedlimit.
- We are accessing the data member by the reference variable of Parent class Bike which refers to the subclass object of Honda.
- Since we are accessing the data member speedlimit which is not overridden, hence it will access the data member speedlimit of the Parent class Bike always.
- Runtime polymorphism can't be achieved by data members.

```
class Bike{
 int speedlimit=90;
}
class Honda extends Bike{
 int speedlimit=150;

 public static void main(String args[]){
  Bike obj=new Honda();  //upcasting
System.out.println(obj.speedlimit); // 90
} }
```

Java Runtime Polymorphism with Multilevel Inheritance

```
class Animal{
void eat(){
System.out.println(”An animal is eating");}
}
class Dog extends Animal{
void eat(){
System.out.println(”A dog is eating fruits");}
}
class BabyDog extends Dog{
void eat(){System.out.println(”A baby dog is drinking milk");
}
public static void main(String args[]){
Animal a1,a2,a3;
a1=new Animal();
a2=new Dog(); //upcasting
a3=new BabyDog(); //upcasting

a1.eat();
a2.eat();
a3.eat();
} }
```

Output:

```
    An animal eating
    A dog is eating fruits
    A baby dog is drinking Milk
```

Example

```
class Animal {
void eat(){
System.out.println(”An animal is eating...");
}  }
class Dog extends Animal{
void eat() {
System.out.println(”A dog is eating...");
}  }
class BabyDog extends Dog{
public static void main(String args[]){
Animal a=new BabyDog();  //upcasting
a.eat();
} }
```

Output:

```
A dog is eating
```

Since, BabyDog is not overriding the eat() method, so eat() method of Dog class is invoked.

Static Binding and Dynamic Binding

- Connecting a method call to the method body is known as binding.
- There are two types of binding
  - Static Binding (also known as Early Binding).
  - Dynamic Binding (also known as Late Binding).
- *Variables have a type*. **int data = 10**;
- *References have a type*. **Class Animal { …}; Animal a1;**
- *Objects have a type*. **Class Animal {…}; Class Cat extends Animal {..};**

  Cat a1 = new Cat(); // a1 is an object of Cat and also of Animal
- When type of the object is determined at compiled time(by the compiler), it is known as **static binding.**
  - If there is any private, final or static method in a class, there is static binding.
- When type of the object is determined at run-time, it is known as **dynamic binding**

Examples of Static and Dynamic Bindings

```
class Dog{
 private void eat(){
  System.out.println("dog is eating...");
  }

 public static void main(String args[]){
  Dog d1=new Dog();
  d1.eat();
 }
}
```

- **Static binding**
- Type of object d1 is known during compile time

```
class Animal{
 void eat(){System.out.println("animal is ea
ting...");}
}
class Dog extends Animal{
 void eat(){System.out.println("dog is eatin
g...");}

 public static void main(String args[]){
  Animal d1 = new Dog();
  d1.eat();
 }  }
```

Dynamic binding

- Object type cannot be determined by the compiler, because the instance of Dog d1 is also an instance of Animal.So compiler doesn't know its type, only its base type.

instanceof operator

- The **instanceof** operator is used to determine if an object is of a particular class.
- The instanceof is also known as type *comparison operator* because it compares the instance with type. It returns either true or false.
- **Example**

  ```
  if (shape1 instanceof Circle)
  ```

  Returns **true** if the object to which **shape1** points "is a" **Circle**
- **Another example:**

  ```
  Simple1 s = new Simple1();
  System.out.println(s instanceof Simple1);
  ```
- An object of subclass type is also a type of parent class.
- For example, `if Dog extends Animal;`

  then object of Dog can be referred by either Dog or Animal class.
- 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 with java instanceof operator

- When Subclass type refers to the object of Parent class, it is known as downcasting.
- If we perform it directly, compiler gives Compilation error.
  - Cat c = new Animal();//downcasting - Compilation error
  - Animal a = new Cat() // Upcasting, no compilation error
- But if we use instanceof operator, downcasting is possible.

```
class Animal { }

 class Cat extends Animal {
  static void method(Animal a)
 {
    if (a instanceof Cat){
       Cat c = (Cat) a;//downcasting is working using instanceof
       System.out.println("ok downcasting performed");
    }
  }
  public static void main (String [] args) {
    Animal a=new Cat();  //upcasting
    Cat.method(a);
  }
```

Downcasting without the use of java instanceof

- Downcasting can also be performed without the use of instanceof operator.

```
class Animal { }
class Cat extends Animal
{
  static void method(Animal a)
{  Cat c = (Cat) a;//downcasting
       System.out.println("ok downcasting performed");}
   public static void main (String [] args) {
    Animal a = new Cat(); //upcasting
    Cat.method(a);
  }  }
```

- Let's take closer look at this, actual object that is referred by a, is an object of Cat class. So if we downcast it, it is fine.
- But what will happen if we write:

```
Animal a=new Animal();
Cat.method(a);
//Now ClassCastException but not in case of instanceof operator
```

Difference between Upcasting and Downcasting

- Upcasting is casting to a supertype, while downcasting is casting to a subtype.
- Upcasting is always allowed, but downcasting involves a type check and can throw a ClassCastException.
- A cast from a sub class to a super class is an upcast, because a sub class object is also a super class object.
- You can upcast whenever there is an **is-a relationship** between two classes.
- Upcasting would be something like this:

  Superclass object name = new Subclass();\\upcasting

  Parent p = new Child(); \\upcasting

- Here **p** is a parent class reference but point to the child object.
- This reference p can access all the methods and variables of parent class but only overridden methods in child class.
- **Downcasting**

  Subclass object name = (Subclass) superclass; \\downcasting

  Child c = (Child) p; \\downcasting

- Here **p** is pointing to the object of child class as we saw earlier and now we cast this parent reference **p** to child class reference **c**.
- Now this child class reference c can access all the methods and variables of child class as well as parent class.

Example of Upcasting and Downcasting

- *For example, if we have two classes, say Machine and Laptop which extends Machine class. Now for upcasting, every laptop will be a machine*
- *For downcasting, every machine may not be a laptop because there may be some machines which can be Printer, Mobile, etc.*
- *Hence downcasting is not always safe, and we explicitly write the class names before doing downcasting.*
- *So that it won't give an error at compile time but it may throw ClassCastExcpetion at run time, if the parent class reference is not pointing to the appropriate child class.*
- To get rid of ClassCastException we can use instanceof operator to check right type of class reference in case of downcasting .
- For example,

```
if(machine instanceof Laptop){
Laptop laptop = machine;
 //here machine must be pointing to Laptop class object .
}
```

Summary

- 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
- Polymorphism promotes and supports reuse of methods and flexibility of programming
- Compile-time and run-time polymorphism
- Static and dynamic binding
- It also supports versatility of code.
- Upcasting and downcasting
- instanceof operator.