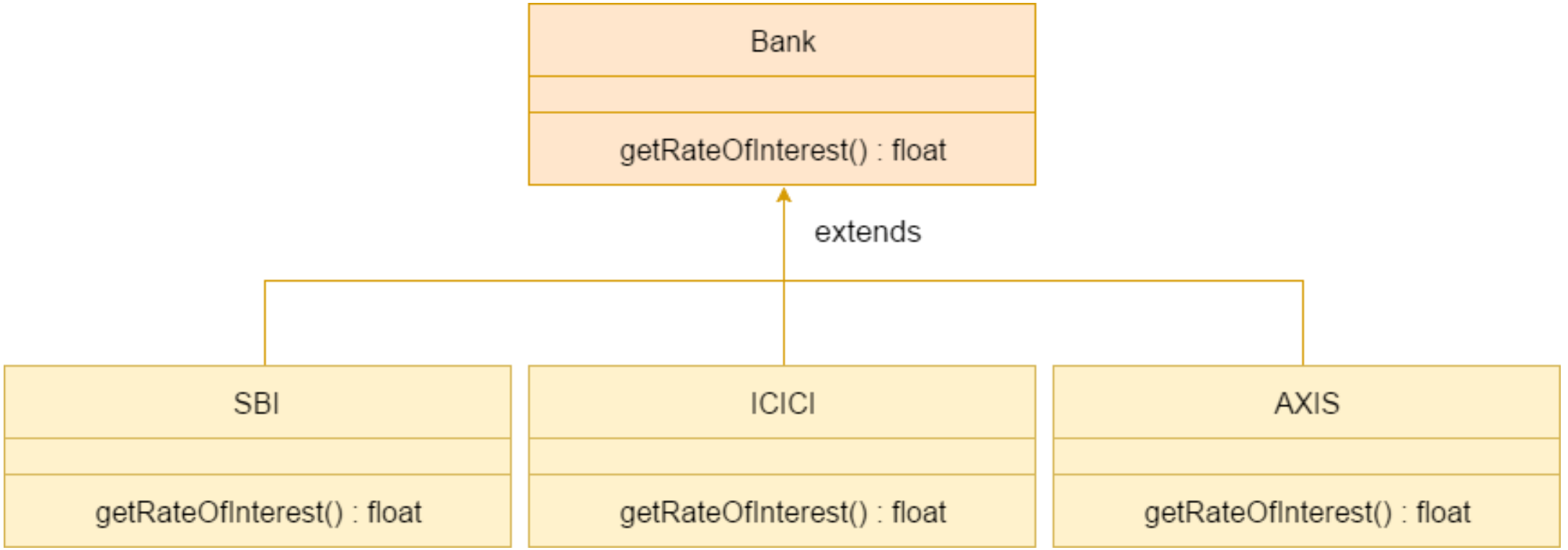
AMRITA VISHWA VIDYAPEETHAM
DEEMED TO BE UNIVERSITY

19CSE204
Object Oriented Paradigm
2-0-3-3

Inheritance : Method Overriding
- Run Time Polymorphism

Method Overriding

- In a class hierarchy, **when a method in a subclass has the same name and type signature as a method** in its superclass, then the method in the subclass is said to **override** the method in the superclass.
- When an overridden method is called from within a subclass, it will always refer to the version of that method defined by the subclass. **The version of the method defined by the superclass will be hidden**.



Example Overriding

```
1  package overriding;
2
3  //Method overriding.
4  class A {
5  int i, j;
6  A(int a, int b) {
7  i = a;
8  j = b;
9  }
10 //display i and j
11 void show() {
12 System.out.println("i and j: " + i + " " + j);
13 }
14 }
15 class B extends A
16 {
17 int k;
18 B(int a, int b, int c) {
19 super(a, b);
20 k = c;
21 }
22 // display k – this overrides show() in A
23 void show() {
24 System.out.println("k: " + k);
25 }
26 }
```

```
29 public class override {
30
31     public static void main(String[] args) {
32         B subOb = new B(1, 2, 3);
33         subOb.show();
34
35     }
36
37 }
```

Output:
K:3

When **show( )** is invoked on an object of type **B**, the version of **show( )** defined
within **B** is used. That is, the version of **show( )** inside **B** overrides the version
declared in **A**.

Example Overriding

```
1  package overriding;
2
3  //Method overriding.
4  class A {
5  int i, j;
6  A(int a, int b) {
7  i = a;
8  j = b;
9  }
10 //display i and j
11 void show() {
12 System.out.println("i and j: " + i + " " + j);
13 }
14 }
15
16 class B extends A {
17 int k;
18 B(int a, int b, int c) {
19 super(a, b);
20 k = c;
21 }
22
23 void show() {
24 super.show(); // this calls A's show()
25 System.out.println("k: " + k);
26 }
27 }
```

```
29 public class override {
30
31     public static void main(String[] args) {
32         B subOb = new B(1, 2, 3);
33         subOb.show();
34
35     }
36
37 }
```

Output:
i and j: 1 2
k: 3

Here, **super.show( )** calls the superclass version of **show( )**.
Method overriding occurs *only* when the names and the type signatures of the two methods are identical. If they are not, then the two methods are simply overloaded.

Note:
Method overriding occurs only when the names and the type signatures of the two methods are identical
Methods with differing type signatures are overloaded – not overridden

Is this example a case of overriding or overloading?

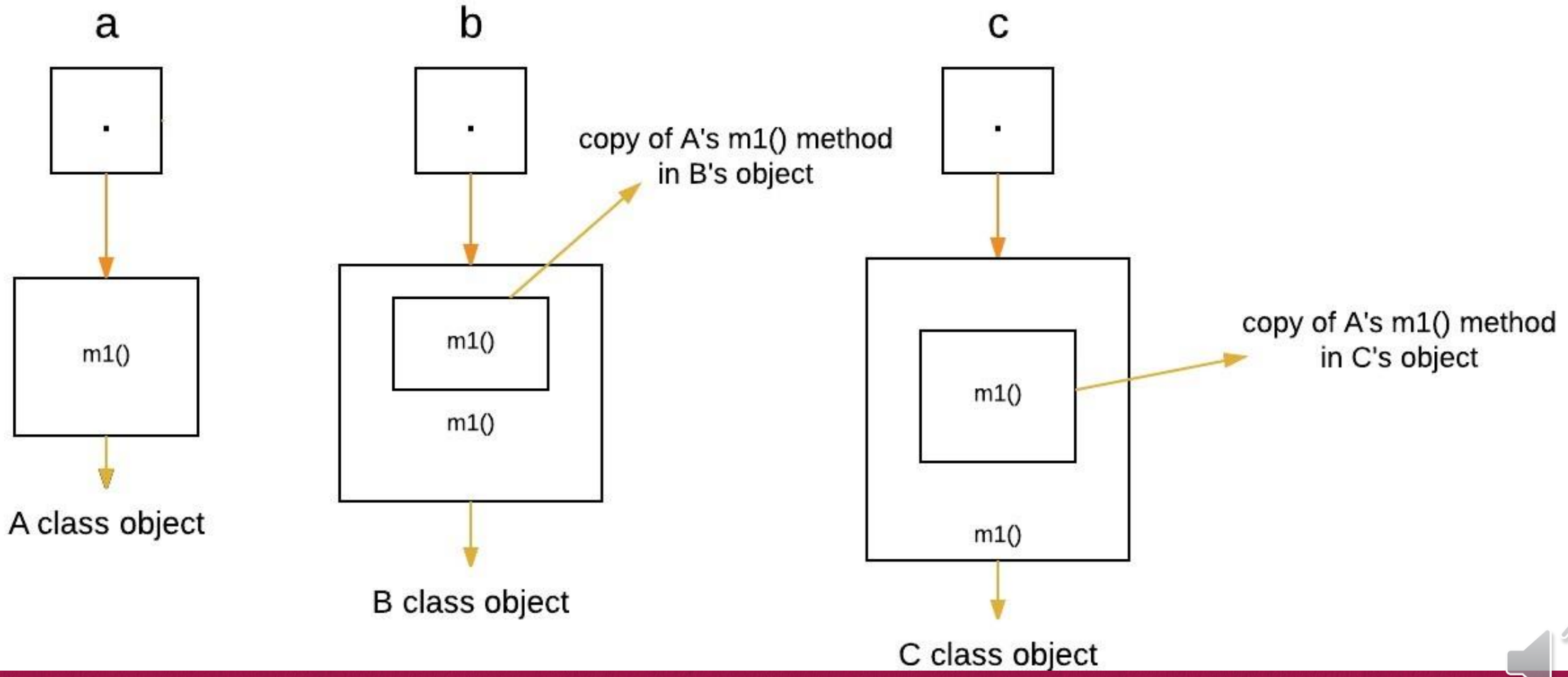
```
// Create a subclass by extending class A.
class B extends A {
int k;
B(int a, int b, int c) {
super(a, b);
k = c;
}
void show(String msg) {
System.out.println(msg + k);
}
}
class Override {
public static void main(String args[]) {
B subOb = new B(1, 2, 3);
subOb.show("This is k: "); // this calls show() in B
subOb.show(); // this calls show() in A
}
}
```

This is a case of Overloading
Methods with differing type signatures are overloaded – not overridden

Output:
This is k: 3
i and j: 1 2

Dynamic Method Dispatch (Run Time Polymorphism)

If a superclass contains a method that is overridden by a subclass, then when different types of objects are referred to through a superclass reference variable, different versions of the method are executed. Here is an example that illustrates dynamic method dispatch:



Dynamic Method Dispatch (Run Time Polymorphism)

- While the examples in the succeeding section demonstrate the **mechanics of method overriding.**
- **Method overriding** forms the basis for one of Java's most powerful concepts: **dynamic method dispatch**.
- Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time., this is how Java implements **run-time polymorphism**
- When an overridden method is called through a superclass reference, **Java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs**. Thus, this determination is made at run time.
- Therefore, if a superclass contains a method that is overridden by a subclass, then when different types of objects are referred to through a superclass reference variable, different versions of the method are executed. **It is the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed**.

Example to illustrate this concept follows

Object reference

```
public static void main(String args[]){

        Box mybox = new Box();
        Box mybox1= mybox;
        Box mybox2 = new Box();

        System.out.println(" The address of the mybox
object is:" +System.identityHashCode(mybox));
        System.out.println(" The value of the mybox
object is:"+mybox);

        System.out.println(" The address of the object
mybox1 is :"+ System.identityHashCode(mybox1));
        System.out.println(" The value of the mybox1
object is:"+mybox1);

        System.out.println(" The address of the object
mybox2 is :"+ System.identityHashCode(mybox2));
        System.out.println(" The value of the mybox2
object is:"+mybox2);}}
```

Output

The address of the mybox object is:1956725890

The value of the mybox object is:Box@74a14482

The address of the object mybox1 is :1956725890

The value of the mybox1 object is:Box@74a14482

The address of the object mybox2 is :356573597

The value of the mybox2 object is:Box@1540e19d

In order to get the address of the object, we used System.identityHashCode function. In order to get the value of the reference stored in that memory address, we are directly printing the object

Example – Dynamic Method Dispatch

```
// Dynamic Method Dispatch
class A {
void callme() {
System.out.println("Inside A's callme
method");
}
}
class B extends A {
// override callme()
void callme() {
System.out.println("Inside B's callme
method");
}
}
class C extends A {
// override callme()
void callme() {
System.out.println("Inside C's callme
method");
}
```

```
class Dispatch {
public static void main(String args[]) {
A a = new A(); // object of type A
B b = new B(); // object of type B
C c = new C(); // object of type C
A r; // obtain a reference of type A
r = a; // r refers to an A object
r.callme(); // calls A's version of callme
r = b; // r refers to a B object
r.callme(); // calls B's version of callme
r = c; // r refers to a C object
r.callme(); // calls C's version of callme
}
}
```

The output from the program is shown here:

```
Inside A's callme method
Inside B's callme method
Inside C's callme method
```

Explanation of previous slide example program (Slide 9)

- **The program creates**
  - One superclass called **A** and two subclasses of it, called **B** and **C**.
  - Subclasses **B** and **C** override **callme( )** declared in **A**.
  - Inside the **main( )** method Objects of type **A**, **B**, and **C** are declared.
  - Also, a reference of type **A**, called **r**, is declared.
- **The program then assigns a reference to each type of object to r and uses that reference to**
  - invoke **callme( )**. As the output shows, the version of **callme( )** executed is determined by
    - the type of object being referred to at the time of the call.
    - Had it been determined by the type of the reference variable, **r**, you would see three calls to **A**'s **callme( )** method.

Applying Method Overriding

```
// Using run-time polymorphism.
class Figure {
double dim1;
double dim2;
Figure(double a, double b) {
dim1 = a;
dim2 = b;
}
double area() {
System.out.println("Area for Figure is
undefined.");
return 0;
}
}
class Rectangle extends Figure {
Rectangle(double a, double b) {
super(a, b);
}
// override area for rectangle
double area() {
System.out.println("Inside Area for
Rectangle.");
return dim1 * dim2;
}
}
class Triangle extends Figure {

Triangle(double a, double b) {
super(a, b);
}
// override area for right triangle
double area() {
System.out.println("Inside Area for
Triangle.");
return dim1 * dim2 / 2;
}
}
```

```
class FindAreas {
public static void main(String args[]) {
Figure f = new Figure(10, 10);
Rectangle r = new Rectangle(9, 5);
Triangle t = new Triangle(10, 8);
Figure figref;
figref = r;
System.out.println("Area is " +
figref.area());
figref = t;
System.out.println("Area is " +
figref.area());
figref = f;
System.out.println("Area is " +
figref.area());
}
}
```

The output from the program is shown here:

```
Inside Area for Rectangle.
Area is 45
Inside Area for Triangle.
Area is 40
Area for Figure is undefined.
Area is 0
```

Explanation:

Through the dual mechanisms of inheritance and run-time polymorphism, it is possible to define one consistent interface that is used by several different, yet related, types of objects.

In this case, if an object is derived from Figure, then its area can be obtained by calling area( ).

The interface to this operation is the same no matter what type of figure is being used.

Namah Shivaya!