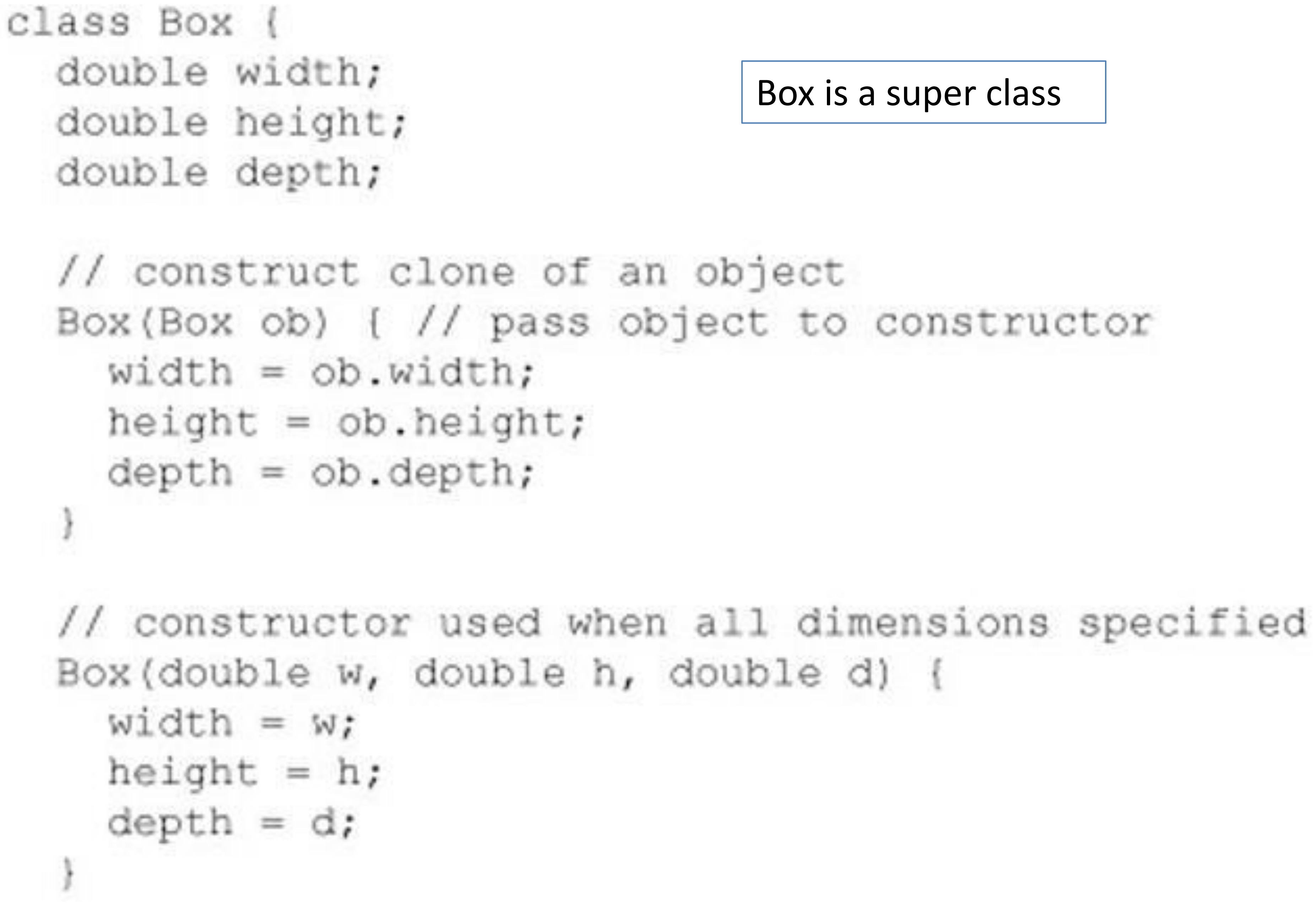
Programming Language II
CSE-215

Prof. Dr. Mohammad Abu Yousuf
yousuf@juniv.edu

A Superclass Variable Can Reference a Subclass Object

- A reference variable of a superclass can be assigned a reference to any subclass derived from that superclass.

- Here, **weightbox** is a reference to **BoxWeight** objects, and **plainbox** is a reference to **Box** objects. Since **BoxWeight** is a subclass of **Box**, it is permissible to assign **plainbox** a reference to the **weightbox** object.



```
class Box {
  double width;
  double height;
  double depth;

  // construct clone of an object
  Box(Box ob) { // pass object to constructor
    width = ob.width;
    height = ob.height;
    depth = ob.depth;
  }

  // constructor used when all dimensions specified
  Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }
```

Box is a super class

```
  // constructor used when no dimensions specified
  Box() {
    width = -1;  // use  -1 to indicate
    height = -1; // an uninitialized
    depth = -1;  // box
  }

  // constructor used when cube is created
  Box(double len) {
    width = height = depth = len;
  }

  // compute and return volume
  double volume() {
    return width * height * depth;
  }
}
```

Box is a super class

```
  // Here, Box is extended to include weight.
  class BoxWeight extends Box {
  double weight; // weight of box

  // constructor for BoxWeight
  BoxWeight(double w, double h, double d, double m) {
    width = w;
    height = h;
    depth = d;
    weight = m;
  }
}

class DemoBoxWeight {
  public static void main(String args[]) {
    BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
    BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076);
    double vol;

    vol = mybox1.volume();
    System.out.println("Volume of mybox1 is " + vol);
    System.out.println("Weight of mybox1 is " + mybox1.weight);
    System.out.println();

    vol = mybox2.volume();
    System.out.println("Volume of mybox2 is " + vol);
    System.out.println("Weight of mybox2 is " + mybox2.weight);
  }
}
```

BoxWeight is a super class

```
class RefDemo {
  public static void main(String args[]) {
    BoxWeight weightbox = new BoxWeight(3, 5, 7, 8.37);
    Box plainbox = new Box();
    double vol;

    vol = weightbox.volume();
    System.out.println("Volume of weightbox is " + vol);
    System.out.println("Weight of weightbox is " +
                       weightbox.weight);
    System.out.println();

    // assign BoxWeight reference to Box reference
    plainbox = weightbox;

    vol = plainbox.volume(); // OK, volume() defined in Box
    System.out.println("Volume of plainbox is " + vol);

    /* The following statement is invalid because plainbox
       does not define a weight member. */
//  System.out.println("Weight of plainbox is " + plainbox.weight);
  }
}
```

A Superclass Variable Can Reference a Subclass Object (Contd..)

- It is important to understand that it is the type of the reference variable—not the type of the object that it refers to—that determines what members can be accessed.

- That is, when a reference to a subclass object is assigned to a superclass reference variable, you will have access only to those parts of the object defied by the superclass. This is why plainbox can’t access weight even when it refers to a BoxWeight object.

Using super

- It is **used** inside a sub-class method definition to call a method defined in the **super**class.
- The **super** keyword in java is a reference variable which is used to refer immediate parent class object.
- Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.
- Private methods of the **super**-class cannot be called. Only public and protected methods can be called by the **super keyword**.
- It is also **used** by class constructors to invoke constructors of its parent class.

Usage of java super Keyword

1. super can be used to refer immediate parent class instance variable.
2. super can be used to invoke immediate parent class method.
3. super() can be used to invoke immediate parent class constructor.

Super is used to refer immediate parent class instance variable.

```
class Animal{
String color="white";
}
class Dog extends Animal{
String color="black";
void printColor(){
System.out.println(color);//prints color of Dog class
System.out.println(super.color);//prints color of Animal
 class
}
}
class TestSuper1{
public static void main(String args[]){
Dog d=new Dog();
d.printColor();
}}
```

We can use super keyword to access the data member or field of parent class. It is used if parent class and child class have same fields.

Syntax: **super.member**

Output:

```
black
white
```

Super is used to refer immediate parent class instance variable.

- In the above example, Animal and Dog both classes have a common property color. If we print color property, it will print the color of current class by default. To access the parent property, we need to use super keyword.

super can be used to invoke parent class method

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void eat(){System.out.println("eating bread...");}

void bark(){System.out.println("barking...");}
void work(){
super.eat();
bark();
}
}
class TestSuper2{
public static void main(String args[]){
Dog d=new Dog();
d.work();
}}
```

The super keyword can also be used to invoke parent class method. It should be used if subclass contains the same method as parent class. In other words, it is used if method is overridden.

Output:

```
eating...
barking...
```

super can be used to invoke parent class method

- In the above example Animal and Dog both classes have eat() method if we call eat() method from Dog class, it will call the eat() method of Dog class by default because priority is given to local.

- To call the parent class method, we need to use super keyword.

super is used to invoke parent class constructor.

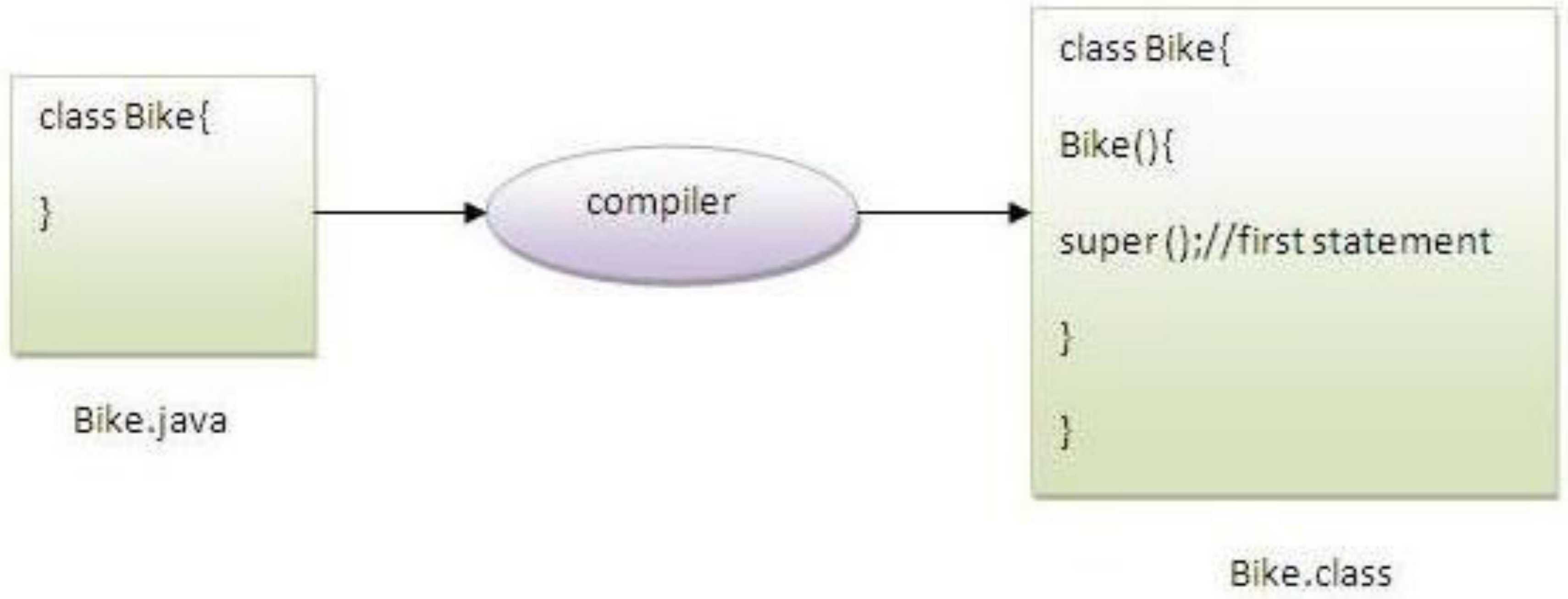
```
class Animal{
Animal(){System.out.println("animal is created");}
}
class Dog extends Animal{
Dog(){
super();
System.out.println("dog is created");
}
}
class TestSuper3{
public static void main(String args[]){
Dog d=new Dog();
}}
```

Output:

```
animal is created
dog is created
```

Another example of super keyword where super() is provided by the compiler implicitly.

- **Note: super() is added in each class constructor automatically by compiler if there is no super() or this().**



Another example of super keyword where super() is provided by the compiler implicitly.

```
class Animal{
Animal(){System.out.println("animal is created");}
}
class Dog extends Animal{
Dog(){
System.out.println("dog is created");
}
}
class TestSuper4{
public static void main(String args[]){
Dog d=new Dog();
}}
```

Output:

```
animal is created
dog is created
```

Super example: real use

```
class Person{
  int id;
  String name;
  Person(int id,String name){
      this.id=id;
      this.name=name;
} }
class Emp extends Person{
   float salary;
   Emp(int id,String name,float salary){
      super(id,name);//reusing parent constructor
      this.salary=salary;
}
void display(){System.out.println(id+" "+name+" "+salary);
} }
class TestSuper5{
public static void main(String[] args){
Emp e1=new Emp(1,"ankit",45000);
e1.display();
}}
```

Output:

```
1 ankit 45000
```

The real use of super keyword. Here, Emp class inherits Person class so all the properties of Person will be inherited to Emp by default. To initialize all the property, we are using parent class constructor from child class. In such way, we are reusing the parent class constructor.

When Constructors Are Called in Multilevel Inheritance

- in a class hierarchy, constructors are called in order of derivation, from superclass to subclass.

- Further, since super( ) must be the first statement executed in a subclass’ constructor, this order is the same whether or not super( ) is used. If super( ) is not used, then the default or parameter less constructor of each superclass will be executed.
- Example: Next slide

```
// Create a super class.
class A {
  A() {
    System.out.println("Inside A's constructor.");
  }
}
// Create a subclass by extending class A.
class B extends A {
  B() {
    System.out.println("Inside B's constructor.");
  }
}

// Create another subclass by extending B.
class C extends B {
  C() {
    System.out.println("Inside C's constructor.");
  }
}

class CallingCons {
  public static void main(String args[]) {
    C c = new C();
  }
}
```

When Constructors Are Called in Multilevel Inheritance

The output from this program is shown here:

```
Inside A's constructor
Inside B's constructor
Inside C's constructor
```

Abstract Classes

- There are situations in which you will want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method.

- That is, sometimes you will want to create a superclass that only defines a generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details. Such a class determines the nature of the methods that the subclasses must implement.

Abstract Classes

- One way this situation can occur is when a superclass is unable to create a meaningful implementation for a method.

- A class that is declared with abstract keyword, is known as abstract class in java. It can have abstract and non-abstract methods (method with body).

Abstract Classes

- **Abstract classes** cannot be instantiated,
- But they can be subclassed. When an **abstract class** is subclassed, the subclass usually provides implementations for all of the **abstract** methods in its parent **class**. However, if it does not, then the subclass must also be declared **abstract** .

Abstract Classes

Example abstract class:

abstract class A{}

Example abstract method:

abstract void printStatus();//no body and abstract

Note:
You cannot **declare abstract methods** in a **non-abstract class**

Example of abstract class that has abstract method

Output:
running safely..

```
abstract class Bike{
 abstract void run();
}
class Honda4 extends Bike{
 void run(){System.out.println("running safely..");}
 public static void main(String args[]){
      Bike obj = new Honda4();
       obj.run();
  }
}
```

In this example, Bike the abstract class that contains only one abstract method run. It implementation is provided by the Honda class.

Real scenario of abstract class

- In this example (next slide), Shape is the abstract class, its implementation is provided by the Rectangle and Circle classes. Mostly, we don't know about the implementation class (i.e. hidden to the end user) and object of the implementation class is provided by the **factory method**.
- A **factory method** is the method that returns the instance of the class. We will learn about the factory method later.
- In this example, if you create the instance of Rectangle class, draw() method of Rectangle class will be invoked.

```
abstract class Shape{
   abstract void draw();
}
//In real scenario, implementation is provided by others i.e. unknown
 by end user
class Rectangle extends Shape{
    void draw(){System.out.println("drawing rectangle");}
}
class Circle1 extends Shape{
    void draw(){System.out.println("drawing circle");}
}
//In real scenario, method is called by programmer or user
class TestAbstraction1{
public static void main(String args[]){
Shape s=new Circle1();//In real scenario, object is provided through
method e.g. getShape() method
s.draw();
}
}
```

```
Output:
drawing circle
```

Another example of abstract class in java

```
abstract class Bank{
    abstract int getRateOfInterest();
}
class SBI extends Bank{
   int getRateOfInterest(){return 7;}
}
class PNB extends Bank{
  int getRateOfInterest(){return 8;}
}

class TestBank{
public static void main(String args[]){
    Bank b;
    b=new SBI();
   System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
   b=new PNB();
   System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
}}
```

```
Output:
Rate of Interest is: 7 %
Rate of Interest is: 8 %
```

Abstract class having constructor, data member, methods etc.

- An abstract class can have data member, abstract method, method body, constructor and even main() method.

- **Rule 1: If there is any abstract method in a class, that class must be abstract.**
- **Rule 2: If you are extending any abstract class that have abstract method, you must either provide the implementation of the method or make this class abstract.**

```
//example of abstract class that have method body
 abstract class Bike{
   Bike(){System.out.println("bike is created");}
   abstract void run();
   void changeGear(){System.out.println("gear changed");}
 }

 class Honda extends Bike{
    void run(){System.out.println("running safely..");}
 }
 class TestAbstraction2{
 public static void main(String args[]){
  Bike obj = new Honda();
  obj.run();
  obj.changeGear();
 }
}
```

```
Output:
bike is created
running safely..
gear changed
```

Thank you