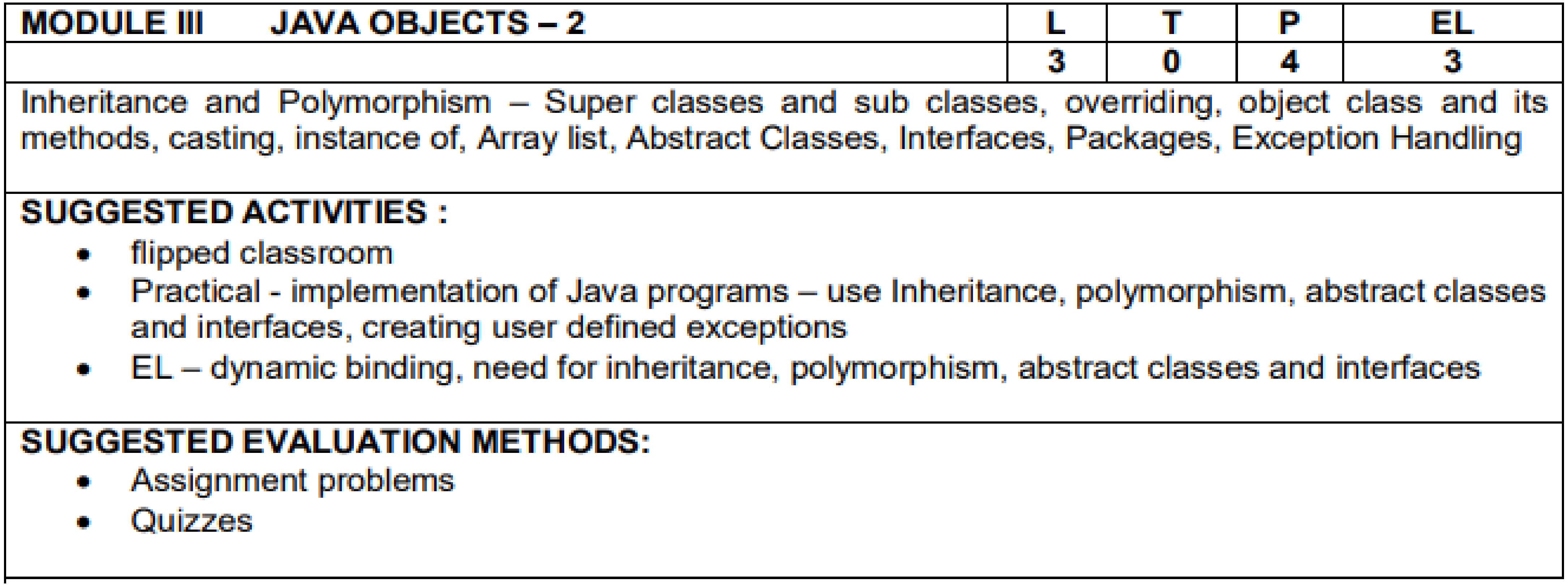
CS6308- Java Programming

V P Jayachitra
Assistant Professor
Department of Computer Technology
MIT Campus
Anna University

Syllabus



MODULE III JAVA OBJECTS – 2	L	T	P	EL
	3	0	4	3

Inheritance and Polymorphism – Super classes and sub classes, overriding, object class and its methods, casting, instance of, Array list, Abstract Classes, Interfaces, Packages, Exception Handling

SUGGESTED ACTIVITIES :

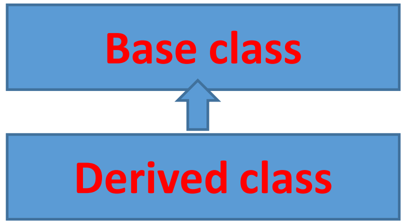
- flipped classroom
- Practical - implementation of Java programs – use Inheritance, polymorphism, abstract classes and interfaces, creating user defined exceptions
- EL – dynamic binding, need for inheritance, polymorphism, abstract classes and interfaces

SUGGESTED EVALUATION METHODS:

- Assignment problems
- Quizzes

What is inheritance?

- The mechanism by which one class acquires the properties(attributes) and functionalities(methods) of another class is called **inheritance**.
- Allows subclass of a class to inherit all of its member elements and methods from its superclass as well as creates its own.
- A special key word **extends** is used to implement this mechanism.
- Inheritance is that it expresses an "is-a" relationship.
  - An eagle *is a* bird.
  - **is-a relationship**: Each object of the subclass also "is a(n)" object of the superclass.



- Subclass:
  - A class that is derived from another class is called a *subclass* (also a *derived class*, *extended class*, or *child class*).
  - Child class which inherits properties of the parent class and defines its own.
- Superclass:
  - The class from which the subclass is derived is called a *superclass* (also a *base class* or a *parent class*).
  - Parent class with some functionality.

```
class  ClassName
[extends SuperClassName ]
[ implements Interface ] {

            [declaration of member
elements ]

            [ declaration of methods ]
            }
```

```
class Superclass
{
 // Superclass attributes
 // Superclass methods
}
class Subclass extends Superclass
{
 // Subclass attributes
 // Subclass methods
}
```

Inheritance in Java

- Specify one superclass for any subclass
- Java does not support the inheritance of multiple superclasses into a single subclass.
- No class can be a superclass of itself.

Why Inheritance?

- **Code reuse**:
  - Used to eliminate redundant coding
  - To reuse the functionality of a class like reusing function libraries.
  - Smaller derived class definitions
- **Extensibility**
- **Ease of modification to common properties and behaviour**
- **Logical structures and grouping**
- **Protected visibility**

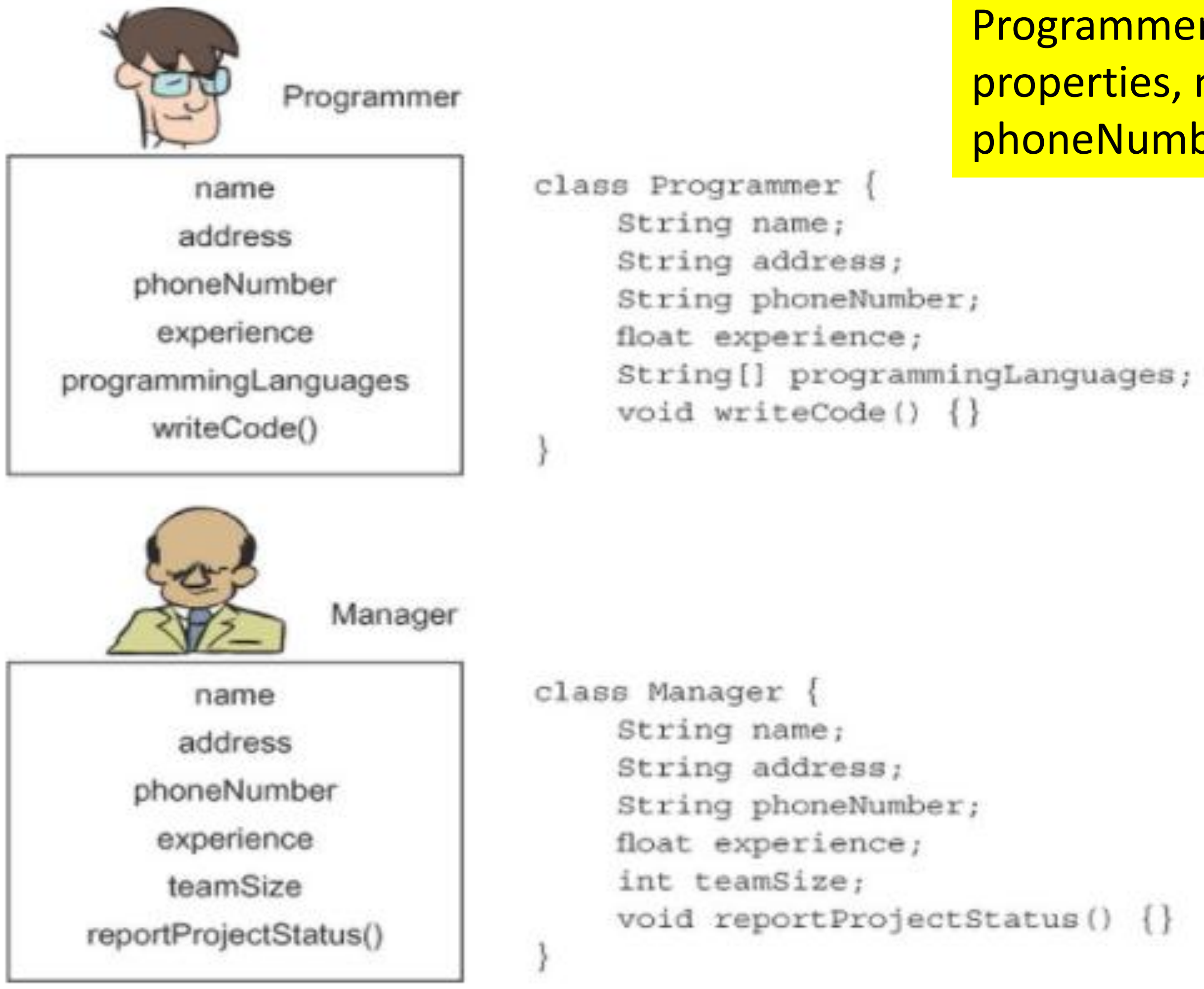
Modifiers

- **Public** – Accessible by any other class in any package.
- **Private** – Accessible only within the class. Hidden from all sub classes
- **Protected** – Accessible only by classes within the same package and any subclasses in other packages.
  - (For this reason, some choose not to use protected, but use private with accessors)
- Default (No Modifier) – Accessible by classes in the same package but not by classes in other packages.

The need to inherit classes

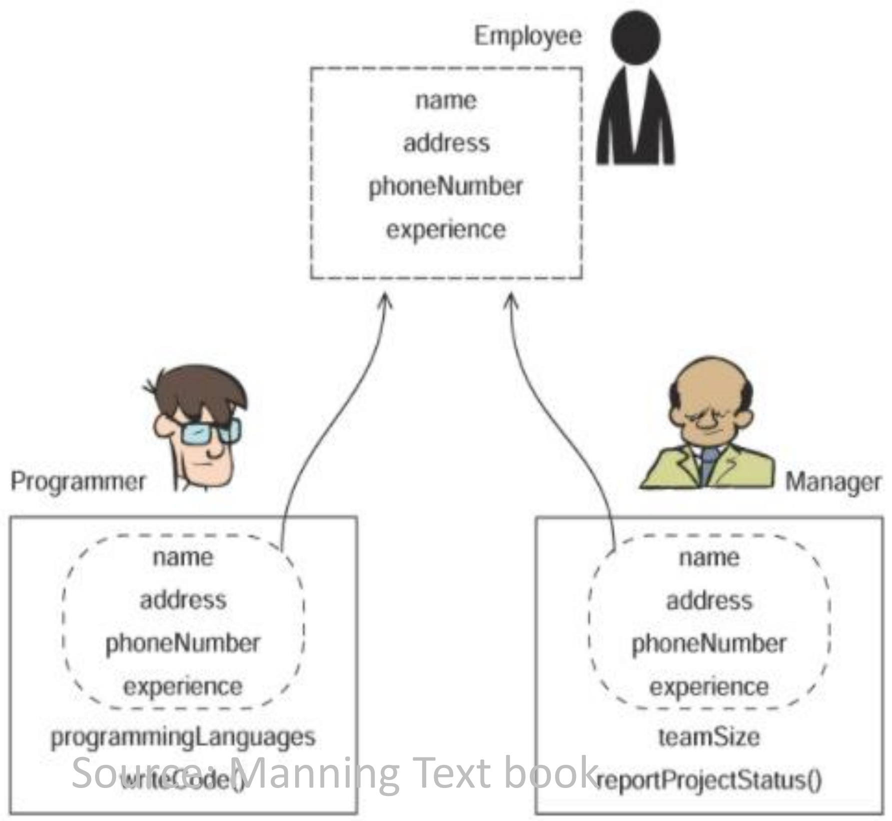
Properties and behavior of a Programmer and a Manager, together with their representations as classes

Programmer and Manager have common properties, namely, name, address, phoneNumber, and experience



```
class Programmer {
    String name;
    String address;
    String phoneNumber;
    float experience;
    String[] programmingLanguages;
    void writeCode() {}
}
```

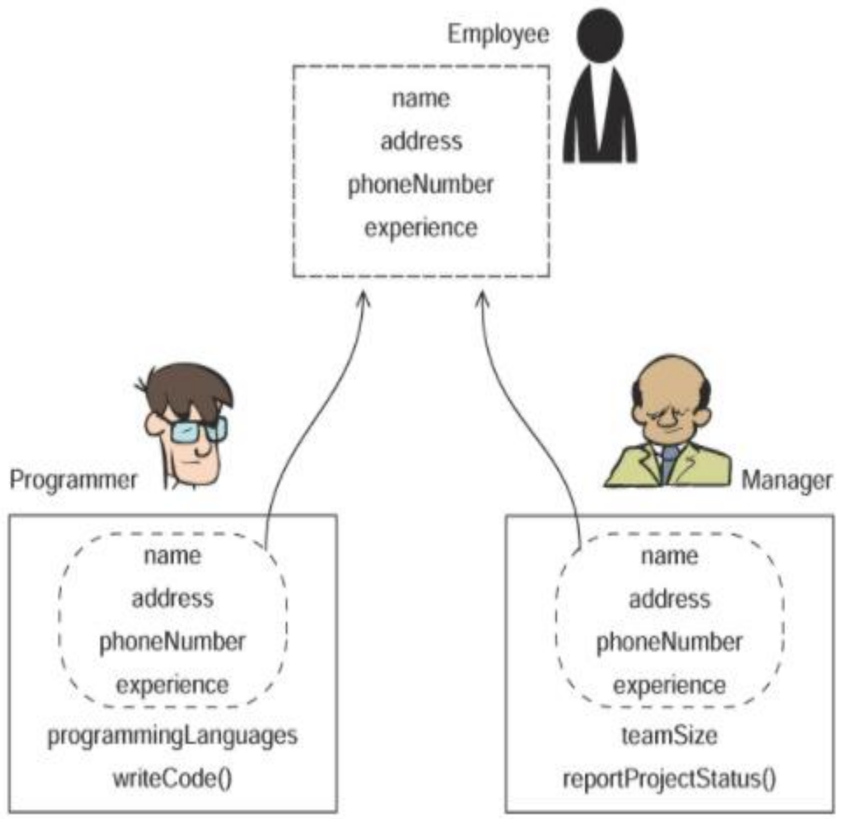
```
class Manager {
    String name;
    String address;
    String phoneNumber;
    float experience;
    int teamSize;
    void reportProjectStatus() {}
}
```



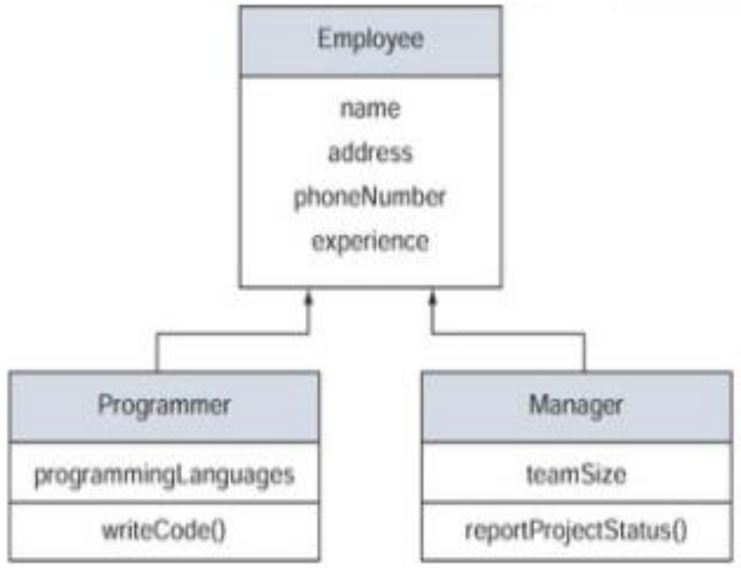
Source: Manning Text book

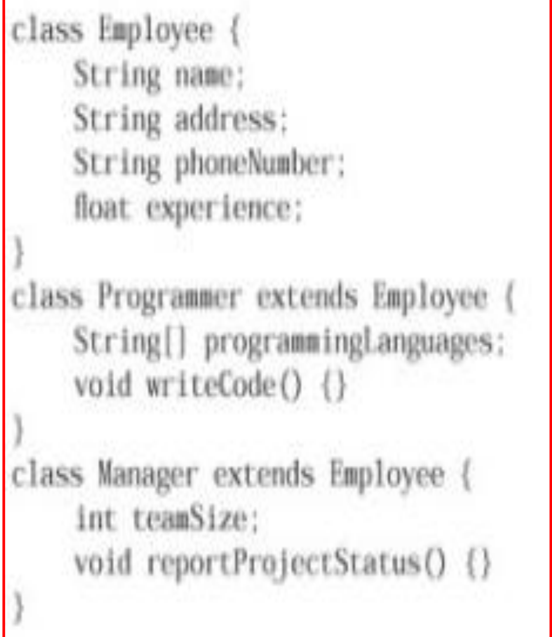
The need to inherit classes

Identify common properties and behaviors of a Programmer and a Manager, pull them out into a new position, and name it Employee.



Programmer and Manager have common properties, namely, name, address, phoneNumber, and experience

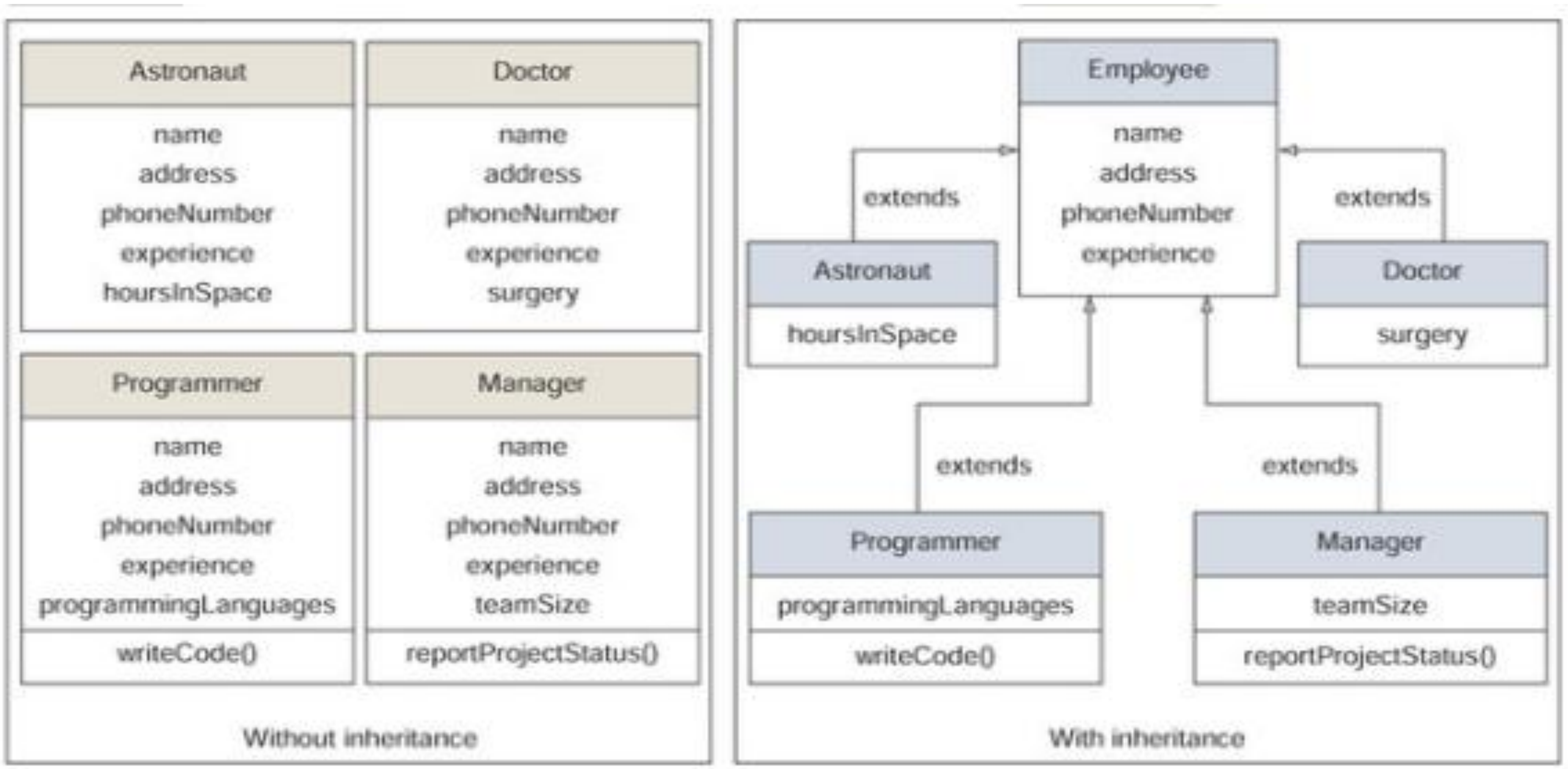




```
class Employee {
    String name;
    String address;
    String phoneNumber;
    float experience;
}
class Programmer extends Employee {
    String[] programmingLanguages;
    void writeCode() {}
}
class Manager extends Employee {
    int teamSize;
    void reportProjectStatus() {}
}
```

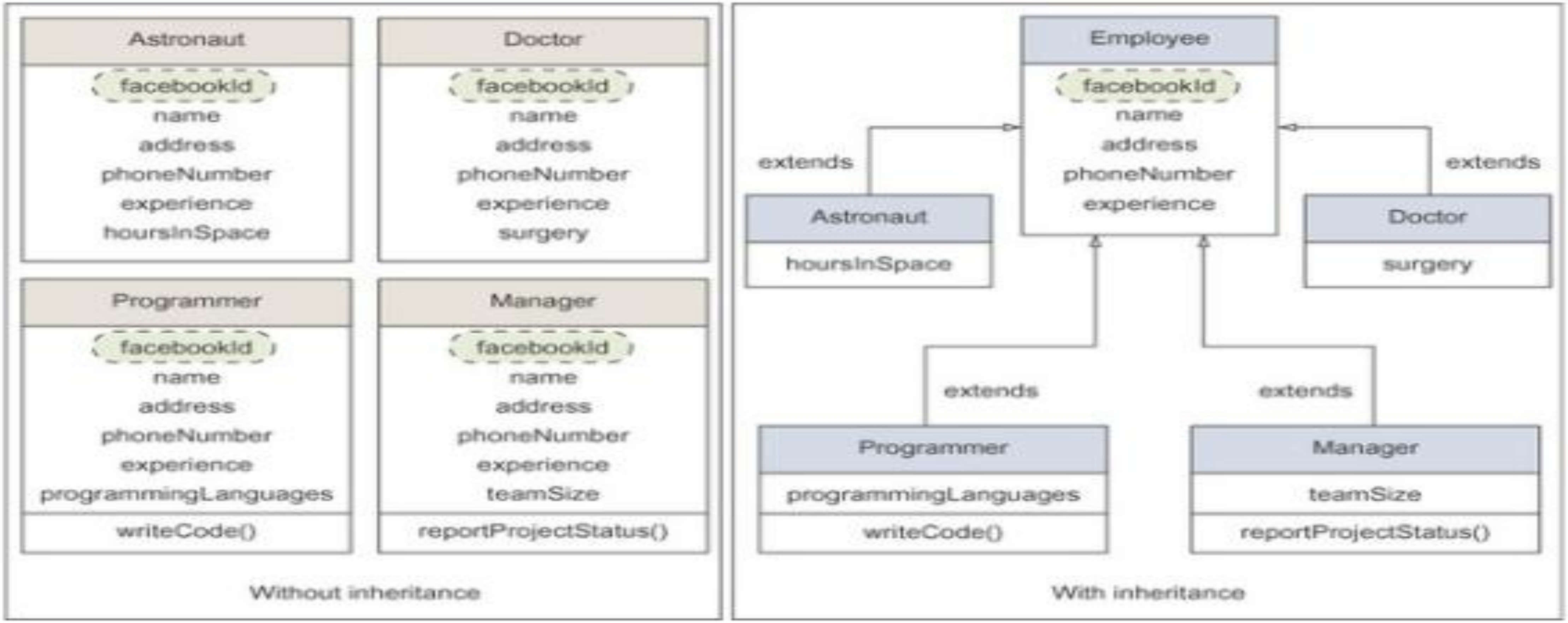
Source: Manning Text book

Extensibility: Smaller derived class definitions



Source: Manning Text book

Ease of modification to common properties and behavior



Adding a new property, facebookId, to all classes, with and without the base class Employee

Source: Manning Text book

What all is inherited?

- A derived class method can access
  - All public member functions and fields of base
  - All protected member functions and fields of base
  - All methods and fields of itself
  - Subclass can add new methods and fields.
  - Constructors can be invoked by the subclass
- A subclass or derived class method cannot access
  - Any private methods or fields of base
  - Any protected or private members of any other class
  - Constructors are not inherited

Super

```
super.method(parameters)
  super(parameters);
```

- Constructors are not inherited, even though they have public visibility
- The `super` reference can be used to refer to the parent class, and is used to invoke the parent's constructor
  1. To call a parent’s method, use **super.*methodName*(…)**
  2. To call a parent’s constructor, use super(some parameter) from the child class’ constructor
- still use *this* (super not needed) to access parent’s fields: **this.parentVar**

What You Can Do in a Subclass?

- A subclass inherits all of the *public* and *protected* members of its parent, no matter what package the subclass is in.
- If the subclass is in the same package as its parent, it also inherits the *package-private* members of the parent.
  - The inherited fields can be used directly, just like any other fields.
  - You can declare a field in the subclass with the same name as the one in the superclass, thus *hiding* it (not recommended).
  - You can declare new fields in the subclass that are not in the superclass.
  - The inherited methods can be used directly as they are.
  - You can write a new *instance* method in the subclass that has the same signature as the one in the superclass, thus *overriding* it.
  - You can write a new *static* method in the subclass that has the same signature as the one in the superclass, thus *hiding* it.
  - You can declare new methods in the subclass that are not in the superclass.
  - You can write a subclass constructor that invokes the constructor of the superclass, either implicitly or by using the keyword super.

Private Members in a Superclass

- Private Members in a Superclass
- A subclass does not inherit the private members of its parent class.
- However, if the superclass has public or protected methods for accessing its private fields, these can also be used by the subclass.
- A nested class has access to all the private members of its enclosing class—both fields and methods.
- Therefore, a public or protected nested class inherited by a subclass has indirect access to all of the private members of the superclass.

```java
/* A simple example of inheritance. */
// superclass.
class A {
        int i, j;
        void showij() {
                System.out.println("i and j: " + i + " " + j);
        }
}
// subclass by extending class A.
class B extends A {
        int k;
        void showk() {
                System.out.println("k: " + k);
        }
        void sum() {
                System.out.println("i+j+k: " + (i + j + k));
        }
}
```

```java
class Demo{
        public static void main(String args[]) {
                A superOb = new A();
                B subOb = new B();
                // The superclass may be used by itself.
                superOb.i = 10;
                superOb.j = 20;
                System.out.println("Contents of superOb: ");
                superOb.showij();
                System.out.println();
        /* The subclass has access to all public members of its superclass. */
                subOb.i = 7;
                subOb.j = 8;
                subOb.k = 9;
                System.out.println("Contents of subOb: ");
                subOb.showij();
                subOb.showk();
                System.out.println();
                System.out.println("Sum of i, j and k in subOb: ");
                subOb.sum();
        }   }
```

OUTPUT:

```
Contents of superOb:
i and j: 10 20
Contents of subOb:
i and j: 7 8
k: 9
Sum of i, j and k in subOb:
i+j+k: 24
```

```
/* Simple example of access modifier.
This program will not compile. */


// Create a superclass.
class A {
		int i; // public by default
		private int j; // private to A

		void setij(int x, int y) {
				i = x;
				j = y;
		}
}
```

```
// A's j is not accessible here.
class B extends A {
		int total;
		void sum() {
				total = i + j; // ERROR, j is not accessible here
		}
}
class Demo{
		public static void main(String args[]) {
				B subOb = new B();
				subOb.setij(10, 12);
				subOb.sum();
				System.out.println("Total is " + subOb.total);
		}
}
```

```
/* Example of access modifier with public, private and protected data */
class BaseClass {
            public int x = 10;
            private int y = 10;
            protected int z = 10;
            int a = 10;
            public int getX() {
            return x;
}
public void setX(int x) {
            this.x = x;            }
private int getY() {
            return y;            }
private void setY(int y) {
            this.y = y;            }
protected int getZ() {
            return z;            }
protected void setZ(int z)  {
            this.z = z;            }
int getA() {
            return a;            }
void setA(int a) {
            this.a = a;            }
}
```

OUTPUT:

```
Value of x is : 10
Value of x is : 20
Value of z is : 10
Value of z is : 30
Value of x is : 10
Value of x is : 20
```

```
public class Demo extends BaseClass {
            public static void main(String args[]) {
                        BaseClass Br = new BaseClass();
                        Br.z = 0;
                        Demo subClassObj = new Demo();
                        //Access Modifiers - Public
                        System.out.println("Value of x is : " + subClassObj.x);
                        subClassObj.setX(20);
                        System.out.println("Value of x is : " + subClassObj.x);
                        // Access Modifiers - Private
                        // compilaton error as the fields and methods are private

                        /* System.out.println("Value of y is : "+subClassObj.y);
                        subClassObj.setY(20);
                        System.out.println("Value of y is : "+subClassObj.y);*/

                        //Access Modifiers - Protected
                        System.out.println("Value of z is : " + subClassObj.z);
                        subClassObj.setZ(30);
                        System.out.println("Value of z is : " + subClassObj.z);
                        //Access Modifiers - Default
                        System.out.println("Value of x is : " + subClassObj.a);
                        subClassObj.setA(20);
                        System.out.println("Value of x is : " + subClassObj.a);
            }
}
```

The Object Class

- A class called `Object` is defined in the `java.lang` package of the Java standard class library
- All classes are derived from the `Object` class
- If a class is not explicitly defined to be the child of an existing class, it is assumed to be the child of the `Object` class
- Therefore, the `Object` class is the ultimate root of all class hierarchies

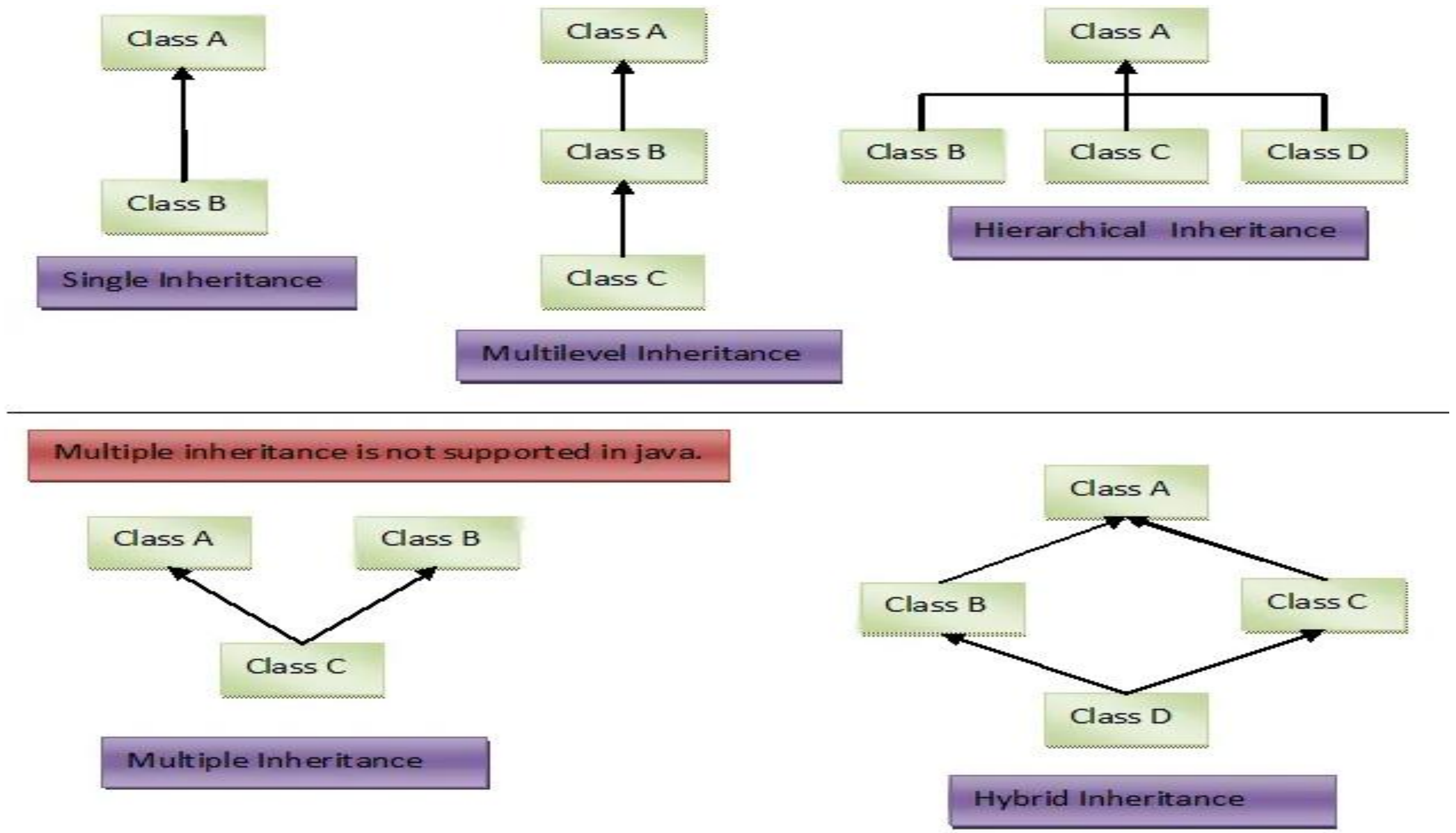
The Object Class

- All objects are guaranteed to have a `toString` method via inheritance
- Thus the `println` method can call `toString` for any object that is passed to it
- The `equals` method of the `Object` class returns true if two references are aliases
- We can override `equals` in any class to define equality in some more appropriate way
- The `String` class (as we've seen) defines the `equals` method to return true if two `String` objects contain the same characters
- Therefore the `String` class has overridden the `equals` method inherited from `Object` in favor of its own version
- If a class header does not include the extends clause the class extends the `Object` class by default

  ```
  public class Die
  ```

  - `Object` is an ancestor to all classes
  - it is the only class that does not extend some other class

Types of Inheritance



Multiple inheritance

- Java supports *single inheritance*, meaning that a derived class can have only one parent class
- *Multiple inheritance* allows a class to be derived from two or more classes, inheriting the members of all parents
- Collisions, such as the same variable name in two parents, have to be resolved
- Java does not support multiple inheritance
- In most cases, the use of interfaces gives us aspects of multiple inheritance without the overhead

overriding

- Method overriding:
  - 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
- A parent method can be invoked explicitly using the `super` reference
- If a method is declared with the `final` modifier, it cannot be overridden
- Any method that is not `final` may be overridden by a descendant class
- Same signature as method in ancestor
- May not reduce visibility
- May use the original method if simply want to add more behavior to existing
- The concept of overriding can be applied to data and is called *shadowing variables*
- Shadowing variables should be avoided because it tends to cause unnecessarily confusing code

Overriding vs Overloading

- Overloading deals with multiple methods with the same name in the same class, but with different signatures
- Overriding deals with two methods, one in a parent class and one in a child class, that have the same signature
- Overloading lets you define a similar operation in different ways for different data
- Overriding lets you define a similar operation in different ways for different object types
- **overriding** a method is when a subclass has method with the same signature (name and parameter list) as its superclass
  - Mover’s act() and Bouncer’s act()
- **Overloading** a method is when two methods have the same name, but different parameter lists
  - Arrays.sort(array, begin, end) and Arrays.sort(array)

```java
// Method overriding.
class A {
  int i, j;
  A(int a, int b) {
    i = a;
    j = b;
  }

  // display i and j
  void show() {
    System.out.println("i and j: " + i + " " + j);
  }
}
```

```java
class B extends A {
  int k;

  B(int a, int b, int c) {
    super(a, b);
    k = c;
  }

  // display k - this overrides show() in A
  void show() {
    System.out.println("k: " + k);
  }
}

class Override {
  public static void main(String args[]) {
    B subOb = new B(1, 2, 3);

    subOb.show(); // this calls show() in B
  }
}
```

The output produced by this program is shown here: k: 3

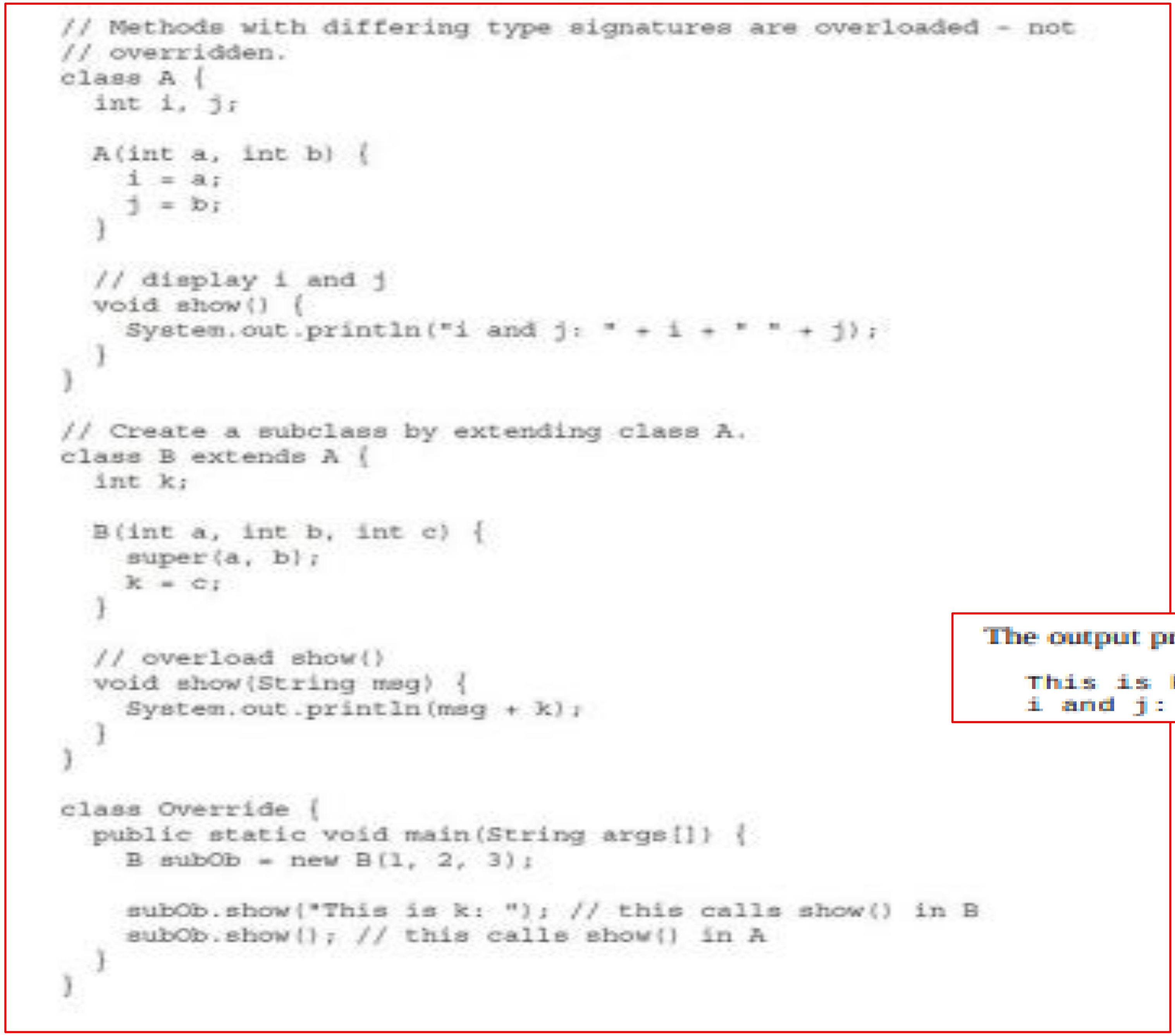
```java
// Method overriding.
class A {
  int i, j;
  A(int a, int b) {
    i = a;
    j = b;
  }

  // display i and j
  void show() {
    System.out.println("i and j: " + i + " " + j);
  }
}
```

```java
class B extends A {
  int k;

  B(int a, int b, int c) {
    super(a, b);
    k = c;
  }

  void show() {
    super.show(); // this calls A's show()
    System.out.println("k: " + k);
  }
}
```

```java
class Override {
  public static void main(String args[]) {
    B subOb = new B(1, 2, 3);

    subOb.show(); // this calls show() in B
  }
}
```

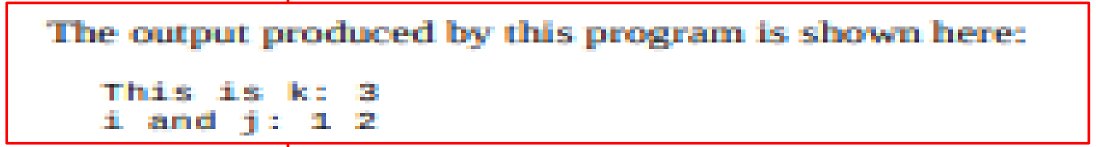
The output produced by this program is shown here:

```
i and j: 1 2
k: 3
```



```
// Methods with differing type signatures are overloaded - not
// overridden.
class A {
  int i, j;

  A(int a, int b) {
    i = a;
    j = b;
  }

  // display i and j
  void show() {
    System.out.println("i and j: " + i + " " + j);
  }
}

// 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;
  }

  // overload show()
  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
  }
}
```

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.



The output produced by this program is shown here:

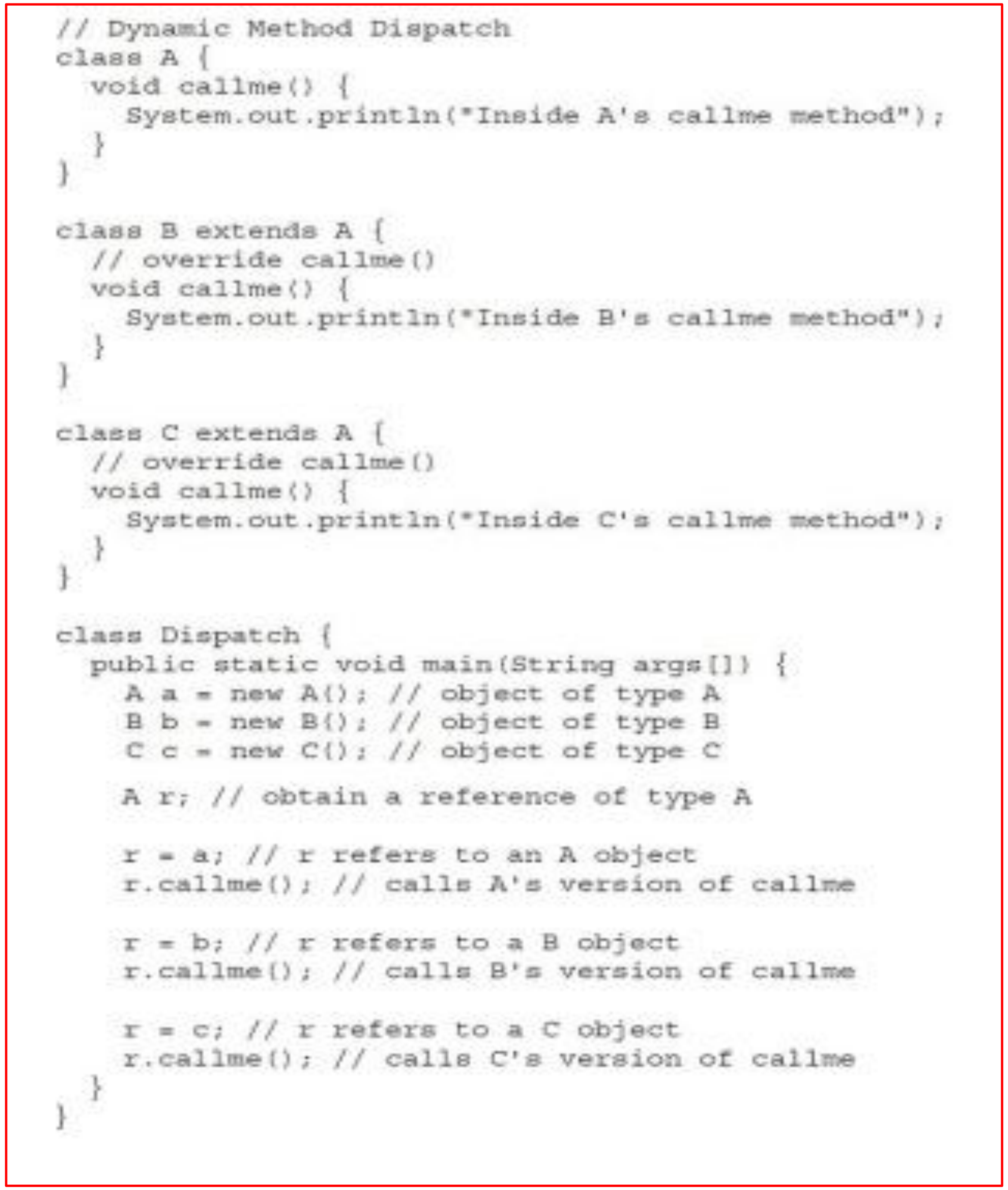
```
This is k: 3
i and j: 1 2
```

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.
- Dynamic method dispatch implements run-time polymorphism.

Why Overridden Methods?

- Overridden methods allow Java to support run-time polymorphism.
- Polymorphism allows a general class to specify methods that will be common to all of its derivatives, while allowing subclasses to define the specific implementation of some or all of those methods.
- Overridden methods are another way that Java implements the “one interface, multiple methods” aspect of polymorphism.



```java
// 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
```

The program then in turn assigns a reference to each type of object to r and uses that reference to invoke callme( ).
The output shows, the version of callme( ) executed is determined by the type of object being referred to at the time of the call.

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
```

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