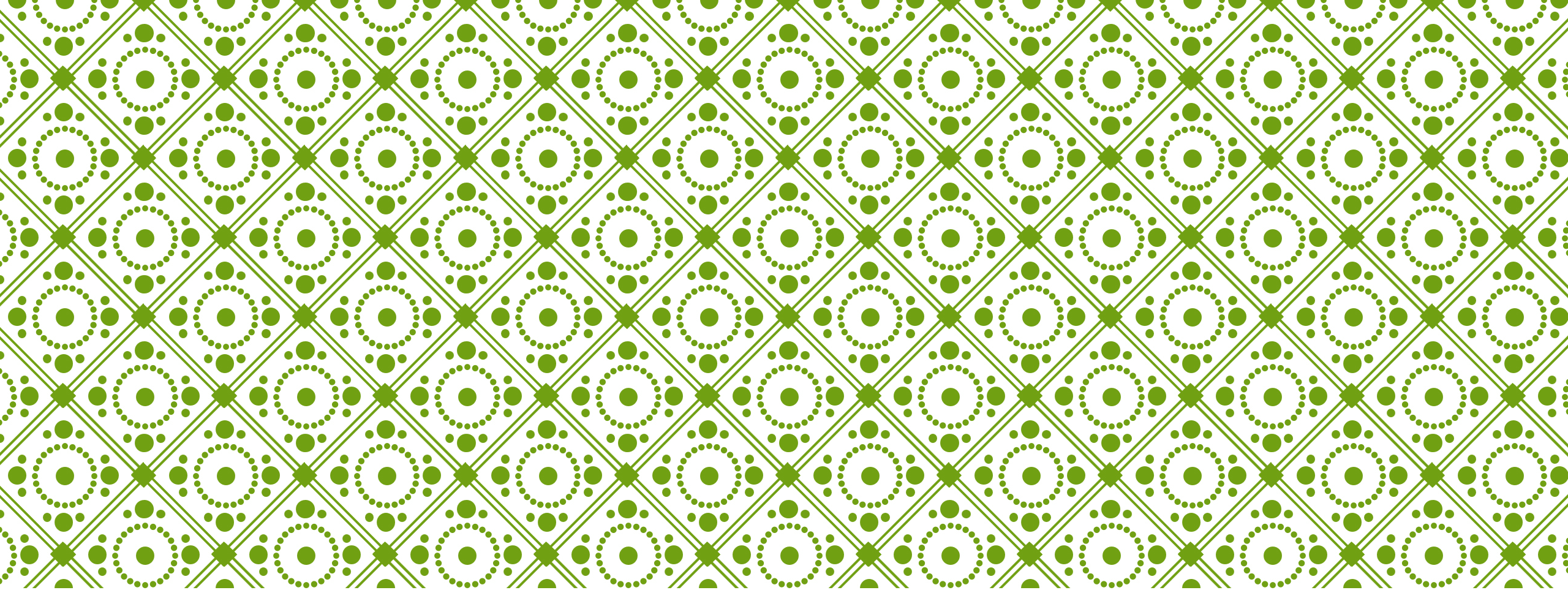


PROGRAMMING 2
CHAPTER 11

Gaza University
Musbah j. Mousa

INHERITANCE

- Inheritance in java is a mechanism in which one object acquires all the properties and behaviours of parent object.
- The idea behind inheritance in java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of parent class, and you can add new methods and fields also.
- Inheritance represents the IS-A relationship, also known as parent-child relationship.

WHY USE INHERITANCE IN JAVA

- For Method Overriding (so runtime polymorphism can be achieved).
- For Code Reusability.

SYNTAX OF JAVA INHERITANCE

- **class** Subclass-name **extends** Superclass-name {

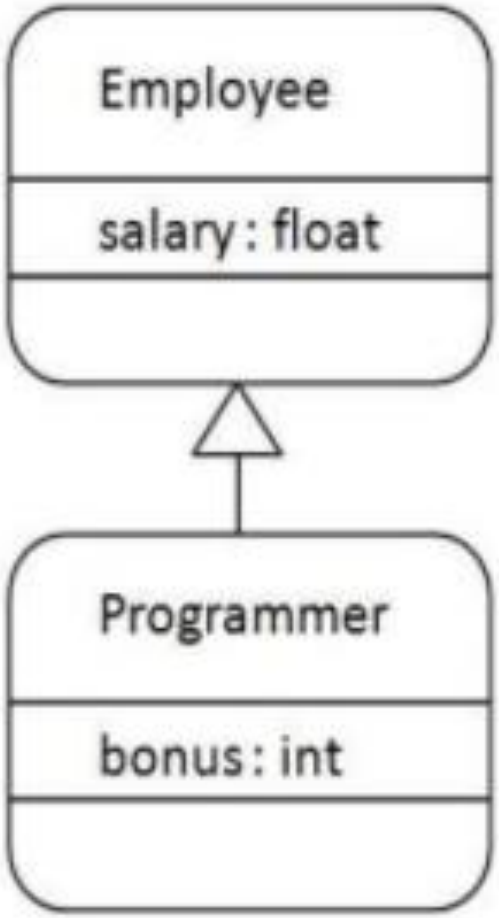
//methods and fields

}

INHERITANCE

- The extends keyword indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality. In the terminology of Java, a class which is inherited is called **parent** or **superclass** and the new class is called **child** or **subclass**.

INHERITANCE

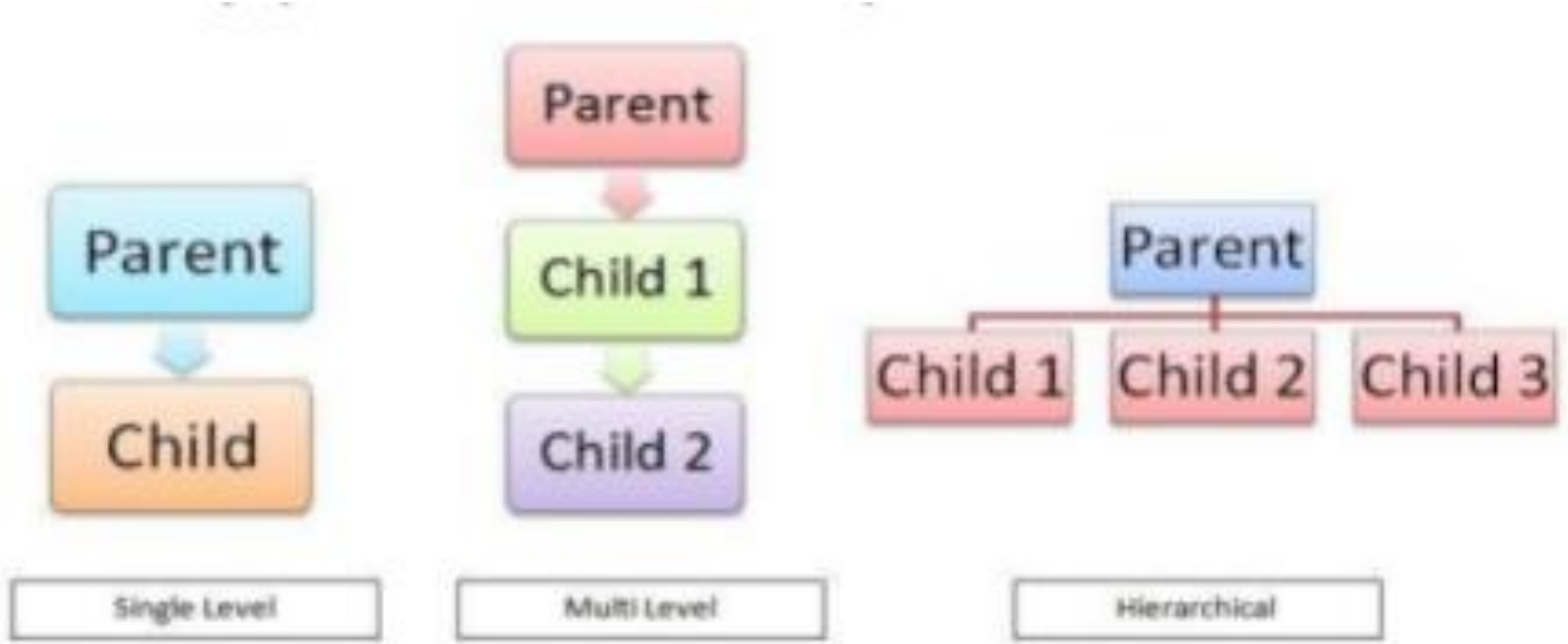



```
•class Employee {
 float salary=40000;
}
class Programmer extends Employee{
int bonus=10000;
public static void main(String args[]){
Programmer p=new Programmer();
System.out.println("Programmer salary is:"+p.salary); System.out.println("Bonus of Programmer is:"+p.bonus);
}}
```

TYPES OF INHERITANCE IN JAVA

- On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.
- In java programming, multiple and hybrid inheritance is supported through interface only. We will learn about interfaces later.

TYPES OF INHERITANCE IN JAVA



USING THE **SUPER** KEYWORD

- *The keyword* **super** *refers to the superclass and can be used to invoke the superclass's methods and constructors.*
- It can be used in two ways:
  1. To call a superclass constructor
  2. To call a superclass method
- The syntax to call a superclass's constructor is:

super() or **super**(arguments);

CALLING SUPERCLASS METHODS

- The keyword **super** can also be used to reference a method other than the constructor in the

superclass. The syntax is

super.method(arguments);

You could rewrite the **printCircle()** method in the **Circle** class as follows:

```
public void printCircle() {
System.out.println("The circle is created " +
super.getDateCreated() + " and the radius is " + radius);
}
```

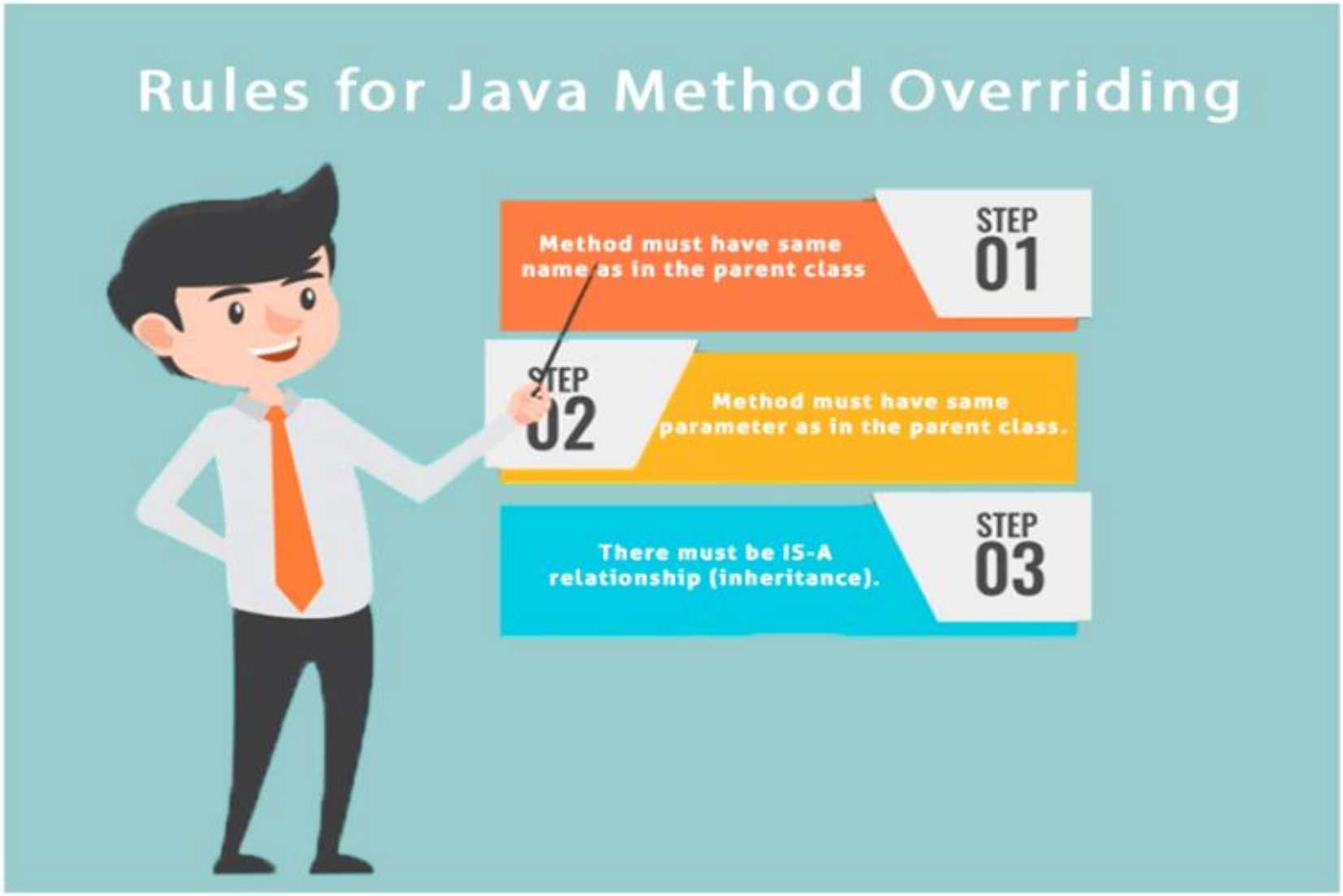
OVERRIDING METHODS

- *To override a method, the method must be defined in the subclass using the same signature as in its superclass.*

- Rules for Java Method Overriding

1. The method must have the same name as in the parent class
2. The method must have the same parameter as in the parent class.
3. There must be an IS-A relationship (inheritance).

OVERRIDING METHODS



OVERRIDING METHODS

Example:

```
class Vehicle{
    void run(){System.out.println("Vehicle is running");}
}

class Bike2 extends Vehicle{
    void run(){System.out.println("Bike is running safely");}

public static void main(String args[]){
  Bike2 obj = new Bike2();//creating object
   obj.run();//calling method
}
```

OVERRIDING METHODS

■ The overriding method must have the same signature as the overridden method and same or compatible return type. Compatible means that the overriding method's return type is a subtype of the overridden method's return type.

■ An instance method can be overridden only if it is accessible. Thus, a private method cannot be overridden, because it is not accessible outside its own class. If a method defined in a subclass is private in its superclass, the two methods are completely unrelated.

■ Like an instance method, a static method can be inherited. However, a static method cannot be overridden. If a static method defined in the superclass is redefined in a subclass, the method defined in the superclass is hidden. The hidden static methods can be invoked using the syntax **SuperClassName.staticMethodName**.

OVERRIDING VS. OVERLOADING

- *Overloading means to define multiple methods with the same name but different signatures. Overriding means to provide a new implementation for a method in the subclass.*

```
public class TestOverriding {
  public static void main(String[] args) {
    A a = new A();
    a.p(10);
    a.p(10.0);
  }
}

class B {
  public void p(double i) {
    System.out.println(i * 2);
  }
}

class A extends B {
  // This method overrides the method in B
  public void p(double i) {
    System.out.println(i);
  }
}
```

(a)

```
public class TestOverloading {
  public static void main(String[] args) {
    A a = new A();
    a.p(10);
    a.p(10.0);
  }
}

class B {
  public void p(double i) {
    System.out.println(i * 2);
  }
}

class A extends B {
  // This method overloads the method in B
  public void p(int i) {
    System.out.println(i);
  }
}
```

(b)

OVERRIDING VS. OVERLOADING

Note the following:

■ Overridden methods are in different classes related by inheritance; overloaded methods

can be either in the same class, or in different classes related by inheritance.

■ Overridden methods have the same signature; overloaded methods have the same

name but different parameter lists.

OVERRIDING VS. OVERLOADING

To avoid mistakes, you can use a special Java syntax, called *override annotation*, to place

@Override before the overriding method in the subclass. For example,

```
 public class Circle extends GeometricObject {
 // Other methods are omitted


 @Override
 public String toString() {
 return super.toString() + "\n radius is " + radius;
}
}
```

POLYMORPHISM

- *Polymorphism means that a variable of a supertype can refer to a subtype object.*
- A subclass is a specialization of its superclass; every instance of a subclass is also an instance of its superclass, but not vice versa.
-

DYNAMIC BINDING

A method can be implemented in several classes along the inheritance chain. The JVM decides which method is invoked at runtime.

```
Object o = new Shape();

System.out.println(o.toString());
```

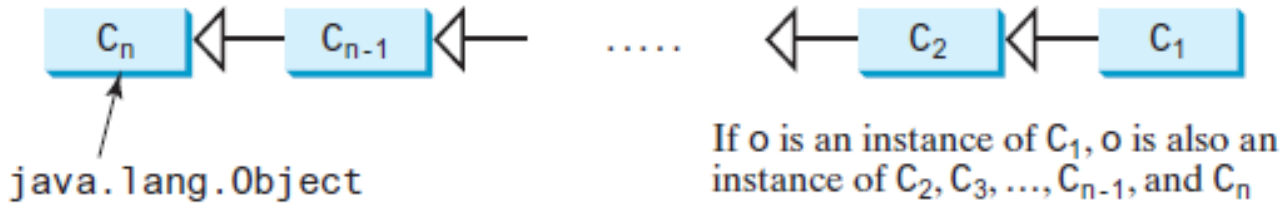


FIGURE 11.2 The method to be invoked is dynamically bound at runtime.

OBJECT CASTING

- *One object reference can be typecast into another object reference. This is called casting object.*

```
Object o = new Student(); // Implicit casting

m(o);


Student b = o; //compile Error


Student b = (Student)o; // Explicit casting
```

OBJECT CASTING

- It is always possible to cast an instance of a subclass to a variable of a superclass (known as *upcasting*) because an instance of a subclass is *always* an instance of its superclass. When casting an instance of a superclass to a variable of its subclass (known as *downcasting*).

- For the casting to be successful, you must make sure the object to be cast is an instance of the subclass.

OBJECT CASTING

```
public static void displayObject(Object o){
        if (o instanceof Circle) {
            ((Circle) o).print();
        } else if (o instanceof Shap) {
            ((Shap) o).print();
    }}
    public static void main(String[] args) {
displayObject(new Circle(5));
        displayObject(new Shap(0,0));
}
```

THE ARRAYLIST CLASS

An **ArrayList** *object can be used to store a list of objects.*

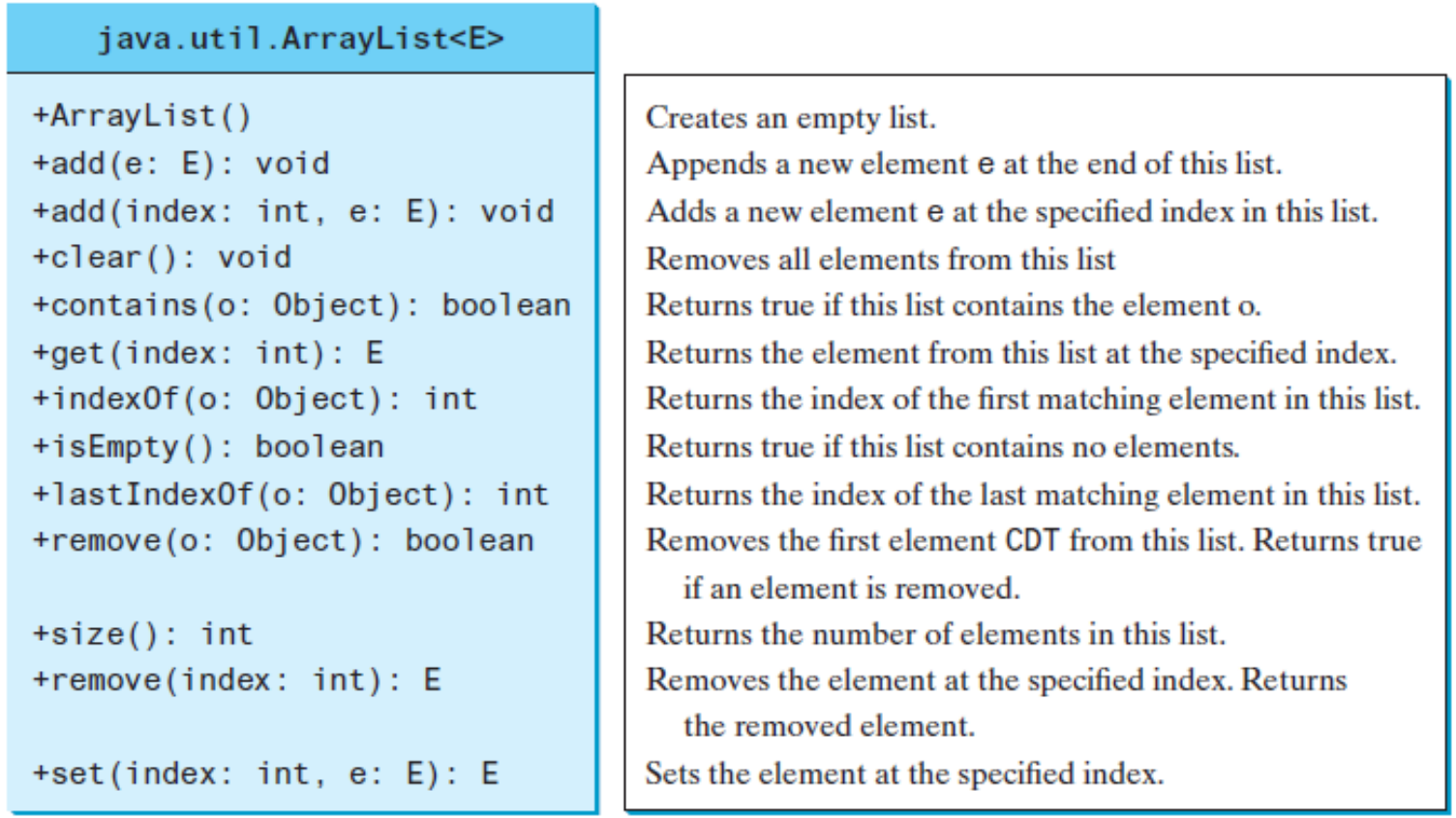


FIGURE 11.3 An `ArrayList` stores an unlimited number of objects.

THE ARRAYLIST CLASS

ArrayList is known as a generic class with a generic type **E**. You can specify a concrete type to replace **E** when creating an **ArrayList**.

```
ArrayList<String> cities = new ArrayList<String>();

ArrayList<java.util.Date> dates = new ArrayList<java.util.Date>();
```

PREVENTING EXTENDING AND OVERRIDING

Neither a final class nor a final method can be extended. A final data field is a constant.

For example, the following class **A** is final and

cannot be extended:

```
public final class A {
// Data fields, constructors, and methods omitted
{
```

PREVENTING EXTENDING AND OVERRIDING

You also can define a method to be final; a final method cannot be overridden by its subclasses.

For example, the following method **m** is final and cannot be overridden:

```
public class Test {

// Data fields, constructors, and methods omitted

public final void m() {

// Do something

{

{
```

PREVENTING EXTENDING AND OVERRIDING

Note

The modifiers **public, protected, private, static, abstract,** and **final** are used

on classes and class members (data and methods), except that the **final** modifier can also be used on local variables in a method. A **final** local variable is a constant inside a method.

QUESTIONS?