6 Working with inheritance

Exam objectives covered in this chapter	What you need to know
[7.1] Implement inheritance.	The need for inheriting classes. How to implement inheritance using classes.
[7.2] Develop code that demonstrates the use of polymorphism.	How to implement polymorphism with classes and interfaces. How to define polymorphic or overridden methods.
[7.3] Differentiate between the type of a reference and the type of an object.	How to determine the valid types of the variables that can be used to refer to an object. How to determine the differences in the members of an object, which ones are accessible, and when an object is referred to using a variable of an inherited base class or an implemented interface.
[7.4] Determine when casting is necessary.	The need for casting. How to cast an object to another class or an interface.
[7.5] Use `super` and `this` to access objects and constructors.	How to access variables, methods, and constructors using `super` and `this`. What happens if a derived class tries to access variables of a base class when they're not accessible to the derived class.
[7.6] Use abstract classes and interfaces.	The role of abstract classes and interfaces in implementing polymorphism.

All living beings inherit the characteristics and behaviors of their parents. The offspring of a fly looks and behaves like a fly, and that of a lion looks and behaves like

a lion. But despite being similar to their parents, all offspring are also different and unique in their own ways. Additionally, a single action may have different meanings for different beings. For example, the action "eat" has different meanings for a fly and a lion. A fly eats nectar, whereas a lion eats antelope.

Something similar happens in Java. The concept of inheriting characteristics and behaviors from parents can be compared to classes inheriting variables and methods from a parent class. Being different and unique in one's own way is similar to how a class can both inherit from a parent and also define additional variables and methods. Single actions having different meanings can be compared to polymorphism in Java.

In the OCA Java SE 7 Programmer I exam, you'll be asked questions on how to implement inheritance and polymorphism and how to use classes and interfaces. Hence, this chapter covers the following:

- Understanding and implementing inheritance
- Developing code that demonstrates the use of polymorphism
- Differentiating between the type of a reference and an object
- Determining when casting is required
- Using `super` and `this` to access objects and constructors
- Using abstract classes and interfaces

6.1 Inheritance with classes

[7.1] Implement inheritance

When we discuss inheritance in the context of an object-oriented programming language such as Java, we talk about how a class can inherit the properties and behavior of another class. The class that inherits from another class can also define additional properties and behaviors. The exam will ask you explicit questions about the need to inherit classes and how to implement inheritance using classes.

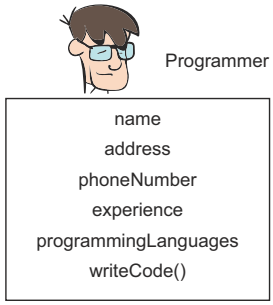
Let's get started with the need to inherit classes.

6.1.1 Need to inherit classes

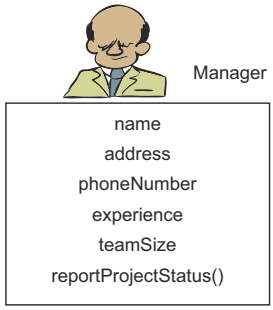
Imagine the positions *Programmer* and *Manager* within an organization. Both of these positions have a common set of properties, including name, address, and phone number. These positions also have different properties. A *Programmer* may be concerned about a project's programming languages, whereas a *Manager* may be concerned with project status reports.

Let's assume you're supposed to store details of all Programmers and Managers in your office. Figure 6.1 shows the properties and behavior that you may have identified for a Programmer and a Manager, together with their representations as classes.

Did you notice that the classes `Programmer` and `Manager` have common properties, namely, `name`, `address`, `phoneNumber`, and `experience`? The next step is to pull out these common properties into a new position and name it something like *Employee*. This step is shown in figure 6.2.




```
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() {}
}
```

Figure 6.1 Properties and behavior of a Programmer and a Manager, together with their representations as classes

This new position, Employee, can be defined as a new class, `Employee`, which is inherited by the classes `Programmer` and `Manager`. A class uses the keyword `extends` to *inherit* a class, as shown in figure 6.3.

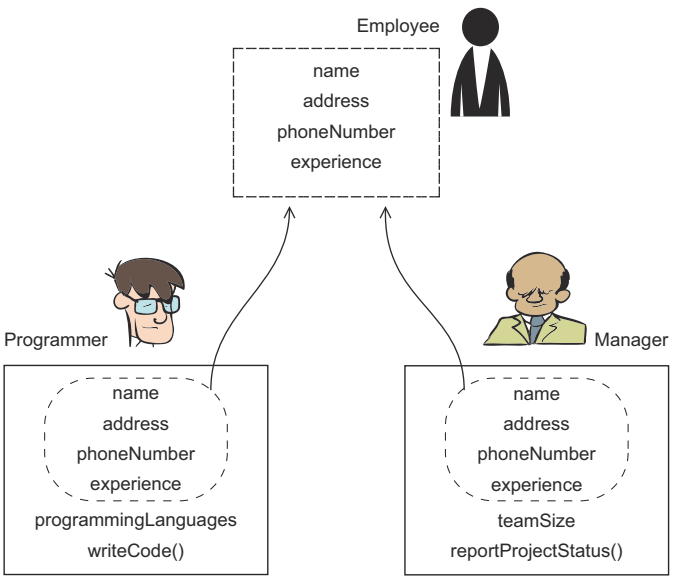
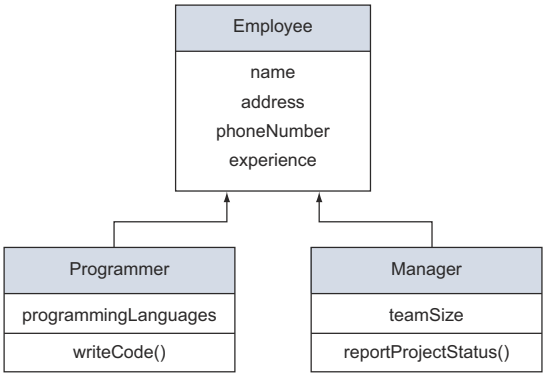


Figure 6.2 Identify common properties and behaviors of a Programmer and a Manager, pull them out into a new position, and name it Employee




```
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() {}
}
```

Figure 6.3 The classes `Programmer` and `Manager` extend the class `Employee`

Inheriting a class is also referred to as *subclassing*. In figure 6.3, the inherited class `Employee` is also referred to as the *superclass*, *base class*, or *parent class*. The classes `Programmer` and `Manager` that inherit the class `Employee` are called *subclasses, derived classes, extended classes*, or *child classes*.

Why do you think you need to pull out the common properties and behaviors into a separate class `Employee` and make the `Programmer` and `Manager` classes inherit it?

SMALLER DERIVED CLASS DEFINITIONS

What if you were supposed to write more specialized classes, such as `Astronaut` and `Doctor`, which have the same common characteristics and behaviors as those of the class `Employee`? With the class `Employee` in place, you only need to define the variables and methods that are specific to the classes `Astronaut` and `Doctor`, and have the classes inherit `Employee`.

Figure 6.4 is a UML representation of the classes `Astronaut`, `Doctor`, `Programmer`, and `Manager`, both with and without inheritance from class `Employee`. As you can see in this figure, the definitions of these classes is smaller when they inherit the class `Employee`.

EASE OF MODIFICATION TO COMMON PROPERTIES AND BEHAVIOR

What happens if your boss steps in and tells you that all of these specialized classes—`Astronaut`, `Doctor`, `Programmer`, and `Manager`—should now have a property `facebookId`? Figure 6.5 shows that with the base class `Employee` in place, you just need to add this variable to that base class. If you haven't inherited from the class `Employee`, you need to add the variable `facebookId` to *each* of these four classes.

Note that common code can be modified and deleted from the base class `Employee` fairly easily.

EXTENSIBILITY

Code that works with the base class in a hierarchy tree can work with all classes that are added using inheritance later.

Assume that an organization needs to send out invitations to all its employees and that it uses the following method to do so:

```
class HR {
    void sendInvitation(Employee emp) {
        System.out.println("Send invitation to" +
                                            emp.name + " at " + emp.address);
    }
}
```

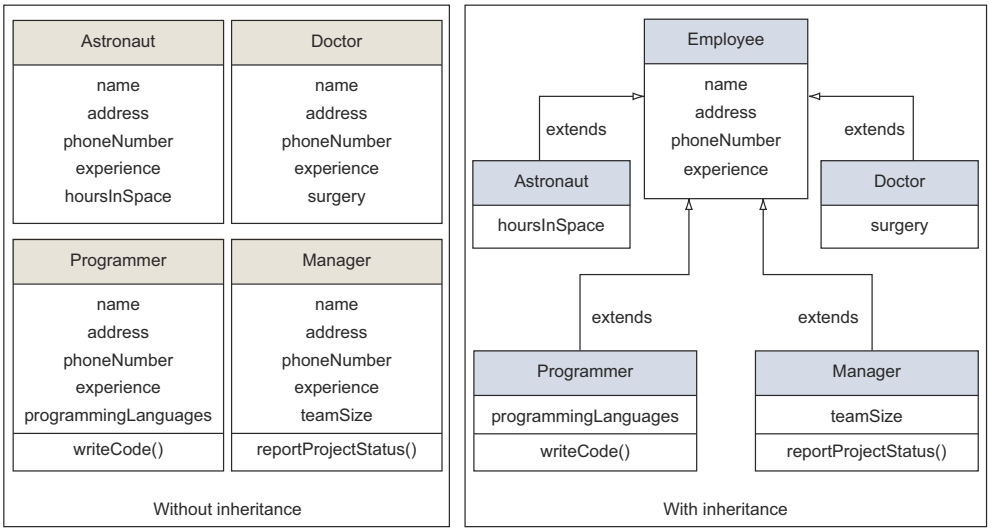


Figure 6.4 Differences in the size of the classes `Astronaut`, `Doctor`, `Programmer`, and `Manager`, both with and without inheriting from the class `Employee`

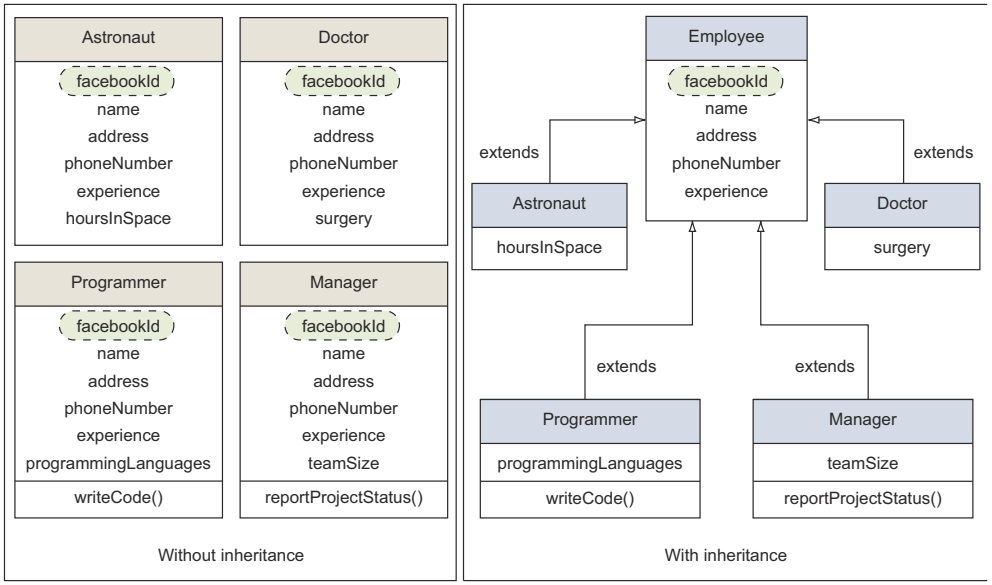


Figure 6.5 Adding a new property, `facebookId`, to all classes, with and without the base class `Employee`

Because method `sendInvitation` accepts an argument of type `Employee`, you can also pass to it a subclass of `Employee`. Essentially, this design means that you can use the previous method with a class defined later that has `Employee` as its base class. Inheritance makes code extensible.

USE TRIED-AND-TESTED CODE FROM A BASE CLASS

You don't need to reinvent the wheel. With inheritance in place, subclasses can use tried-and-tested code from a base class.

CONCENTRATE ON THE SPECIALIZED BEHAVIOR OF YOUR CLASSES

Inheriting a class enables you to concentrate on the variables and methods that define the special behavior of your class. Inheritance lets you make use of existing code from a base class without having to define it yourself.

LOGICAL STRUCTURES AND GROUPING

When multiple classes inherit a base class, it creates a logical group. For an example, see figure 6.5. The classes `Astronaut`, `Doctor`, `Programmer`, and `Manager` are all grouped as types of the class `Employee`.

EXAM TIP Inheritance enables you to reuse code that has already been defined by a class. Inheritance can be implemented by extending a class.

The next section solves the mystery of how you can access the inherited members of a base class directly in a derived class.

6.1.2 A derived class contains within it an object of its base class

The classes `Programmer` and `Manager` inherit the variables and methods defined in the class `Employee` and use them directly, as if they were defined in their own classes. Examine the following code:

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

Derived class Programmer can directly access members of its base class.

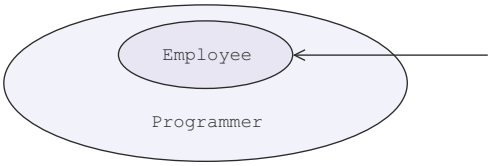


Figure 6.6 An object of a derived class contains an object of its base class.

How can the class `Programmer` assign a value to a variable that's defined in the class `Employee`? You can think of this arrangement as follows: when a class inherits another class, it encloses within it an object of the inherited class. Hence, all the members (variables and methods) of the inherited class are available to the class, as shown in figure 6.6.

But a derived class can't inherit all the members of its base class. The next two sections discuss which base class members are and aren't inherited by a derived class.

6.1.3 Which base class members are inherited by a derived class?

The access modifiers play an important role in determining the inheritance of base class members in derived classes. A derived class inherits all the non-`private` members of its base class. A derived class inherits base class members with the following accessibility modifiers:

- *Default*—Members with default access can be accessed in a derived class only if base and derived classes reside in the same package.
- `protected`—Members with `protected` access are accessible to all the derived classes, regardless of the packages in which the base and derived classes are defined.
- `public`—Members with `public` access are visible to all the other classes.

6.1.4 Which base class members aren't inherited by a derived class?

A derived class doesn't inherit the following members:

- `private` members of the base class.
- Base class members with default access, if the base class and derived classes exist in separate packages.
- Constructors of the base class. A derived class can call a base class's constructors, but it doesn't inherit them (section 6.5 discusses how a derived class can call a base class's constructors using the implicit reference `super`).

Apart from inheriting the properties and behavior of its base class, a derived class can also define additional properties and behaviors, as discussed in the next section.

6.1.5 Derived classes can define additional properties and behaviors

Though derived classes are similar to their base classes, they generally also have differences. Derived classes can define additional properties and behaviors. You may see explicit questions on the exam about how a derived class can differ from its base class.

Take a quick look back at figure 6.5. All the derived classes—`Manager`, `Programmer`, `Doctor`, and `Astronaut`—define additional variables, methods, or both. Derived classes can also define their own constructors and `static` methods and variables. A derived class can also *hide* or *override* its base class's members.

When a derived class defines an instance or class variable with the same name as one defined from its base class, only these new variables and methods are visible to code using the derived class. When a derived class defines different code for a method inherited from a base class by defining the method again, this method is treated as a special method—an *overridden* method.

You can implement inheritance by using either a concrete class or an `abstract` class as a base class, but there are some important differences that you should be aware of. These are discussed in the next section.

6.1.6 Abstract base class versus concrete base class

Figures 6.2 and 6.3 showed how you can pull out the common properties and behavior of a `Programmer` and `Manager` and represent these as a new class, `Employee`. You can define class `Employee` as an `abstract` class, if you think that it's only a categorization and no real employee exists in real life—that is, if all employees are really either *Programmers* or *Managers*. That's the essence of an `abstract` class: it groups the common properties and behavior of its derived classes, but it prevents itself from being instantiated. Also, an `abstract` class can force all its derived classes to define their own implementations for a behavior by defining it as an `abstract` method (a method without a body).

It isn't mandatory for an `abstract` class to define an `abstract` method. It may or may not define any `abstract` methods. But if an `abstract` base class defines one or more `abstract` methods, the class must be marked as `abstract` and the `abstract` methods must be implemented in all its derived classes. If a derived class doesn't implement all the `abstract` methods defined by its base class, then it also needs to be an `abstract` class.

For the exam, you need to remember the following important points about implementing inheritance using an `abstract` base class:

- You can never create objects of an `abstract` class.
- A base class can be defined as an `abstract` class, even if it doesn't define any `abstract` methods.
- A derived class should implement all the `abstract` methods of its base class. If it doesn't, it must be defined as an `abstract` derived class.
- You can use variables of an `abstract` base class to refer to objects of its derived class (discussed in detail in section 6.3).

The first Twist in the Tale exercise for this chapter queries you on the relationship between base and derived classes (answer in the appendix).

Twist in the Tale 6.1

Let's modify the code used in the previous example as follows. Which of the options is correct for this modified code?

```
class Employee {
    private String name;
    String address;
    protected String phoneNumber;
    public float experience;
}
class Programmer extends Employee {
    Programmer (String val) {
        name = val;
    }
    String getName() {
        return name;
    }
}
class Office {
    public static void main(String args[]) {
        new Programmer ("Harry").getName();
    }
}
```

a The class `Office` prints `Harry`.

b The derived class `Programmer` can't define a getter method for a variable defined in its base class `Employee`.

c The derived class `Programmer` can't access variables of its base class in its constructors.

d `new Programmer ("Harry").getName();` isn't the right way to create an object of class `Programmer`.

e Compilation error.

TERMS AND DEFINITIONS TO REMEMBER

Following is a list of terms and their corresponding definitions that you should remember; they're used throughout the chapter, and you'll come across them while answering questions on inheritance in the OCA Java SE 7 Programmer I exam.

- *Base class*—A class inherited by another class. The class `Employee` is a *base class* for the classes `Programmer` and `Manager` in the previous examples.
- *Superclass*—A base class is also known as a *superclass*.
- *Parent class*—A base class is also known as a *parent class*.
- *Derived class*—A class that inherits from another class. The classes `Programmer` and `Manager` are *derived classes* in the previous example.
- *Subclass*—A derived class is also known as a *subclass*.
- *Extended class*—A derived class is also known as an *extended class*.

- *Child class*—A derived class is also known as a *child class*.
- *IS-A relationship*—A relationship shared by base and derived classes. In the previous examples, a `Programmer` IS-A `Person`. A `Manager` IS-A `Person`. Because a derived class represents a specialized type of a base class, a derived IS-A class is a kind of base class.
- `extends`—The keyword used by a class to inherit another class and by an interface to inherit another interface.
- `implements`—The keyword used by a class to implement an interface (interfaces are covered in the next section).

NOTE The terms *base class*, *superclass*, and *parent class* are used interchangeably. Similarly, the terms *derived class* and *subclass* are also used interchangeably.

In this section, you learned that an `abstract` class may define `abstract` methods. Let's take it a step further to interfaces, which can define only `abstract` methods and constants. In the next section, we'll discuss why you need interfaces and how to use them.

6.2 Use interfaces

[7.6] Use abstract classes and interfaces

Interfaces are `abstract` classes taken to extremes. An interface can define only `abstract` methods and constants. All the members of an interface are implicitly `public`.

Let's go back to the example used in section 6.1 with classes `Employee`, `Programmer`, and `Manager`, where `Programmer` and `Manager` inherit the class `Employee`. Imagine that your boss steps in and commands that `Programmer` and `Manager` *must* support additional behaviors, as listed in table 6.1.

Table 6.1 Additional behaviors that need to be supported by the classes `Programmer` and `Manager`

Entity	New expected behavior
`Programmer`	Attend training
`Manager`	Attend training, conduct interviews

How will you accomplish this task? One approach you can take is to define all the relevant methods in class `Employee`. Because both `Programmer` and `Manager` extend the class `Employee`, they'd be able to access these methods. But wait: `Programmer` doesn't need the behavior of the conducting interview task; only Manager should support the functionality of conducting interviews.

Another obvious approach would be to define the relevant methods in the desired classes. You could define methods to conduct interviews in `Manager` and methods to attend training in both `Programmer` and `Manager`. Again, this is not an ideal solution. What will happen if your boss later informs you that all the `Employees` who attend

training should accept a *training schedule*; that is, there is a change in the signature of the method that defines the behavior "attends training"? Can you define separate classes for this behavior and make the classes `Programmer` and `Manager` implement them? No, you can't. Java doesn't allow a class to inherit multiple classes.

Let's try interfaces. Create two interfaces to define the specified behavior:

```
interface Trainable {
    public void attendTraining();
}
interface Interviewer {
    public void conductInterview();
}
```

Though Java does not allow a class to inherit from more than one class, it allows a class to implement multiple interfaces. A class uses the keyword `implements` to implement an interface. In the following code, the classes `Programmer` and `Manager` implement the relevant interfaces (modified code in bold):

```
class Employee {
    String name;
    String address;
    String phoneNumber;
    float experience;
}
class Manager extends Employee implements Interviewer, Trainable {
    int teamSize;
    void reportProjectStatus() {}
    public void conductInterview() {
        System.out.println("Mgr - conductInterview");
    }
    public void attendTraining() {
        System.out.println("Mgr - attendTraining");
    }
}
class Programmer extends Employee implements Trainable{
    String[] programmingLanguages;
    void writeCode() {}
    public void attendTraining() {
        System.out.println("Prog - attendTraining");
    }
}
```

Manager implements Interviewer and Trainable

Programmer implements only Trainable

Figure 6.7 displays the relationships between these classes in a UML diagram.

An `interface` can be represented using either a rectangle with the text <<interface>> or a circle. Both these notations are popular; you may see them in various websites or books. The same relationships can also be represented as depicted in figure 6.8, where the interfaces are defined as circles.

Each class can implement these methods in its own particular manner. As mentioned previously, let's change the method `attendTraining` in the interface `Trainable` so it accepts a `trainingSchedule`, as follows:

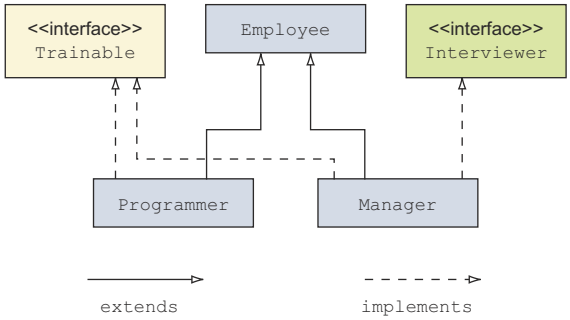


Figure 6.7 Relationships between the classes `Employee`, `Programmer`, and `Manager` and the interfaces `Trainable` and `Interviewer`

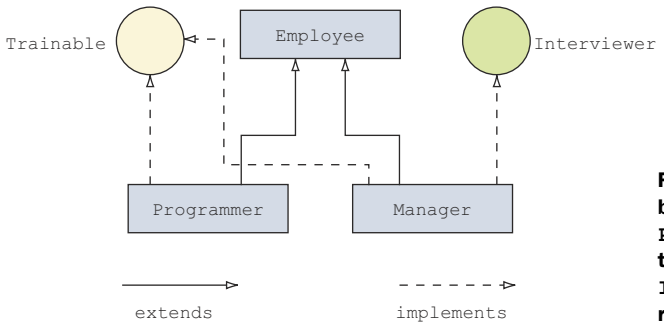


Figure 6.8 Relationships between the classes `Employee`, `Programmer`, and `Manager` and the interfaces `Trainable` and `Interviewer`, with interfaces represented using circles

```
interface Trainable {
    public void attendTraining(String[] trainingSchedule);
}
```

Method attendTraining now accepts a trainingSchedule as an array of String

Does a change in the signature of a method in an interface have any impact on the definition of this method in the classes that implement it? Yes, it does. If the signature of a method is changed in an interface, all classes that implement the interface will fail to compile. Now that the method signature has been modified, the classes `Programmer` and `Manager` must be modified so that the signature of the method `attendTraining` matches in all three: the interface `Trainable` and the classes `Programmer` and `Manager`.

The changes are as follows (in bold):

```
interface Trainable {
    public void attendTraining(String[] trainingSchedule);
}
interface Interviewer {
    public void conductInterview();
}
class Employee {
    String name;
    String address;
    String phoneNumber;
    float experience;
}
```

Signature of method attendTraining should match in interface Trainable, class Manager, and class Programmer

```
class Manager extends Employee implements Interviewer, Trainable {
    int teamSize;
    void reportProjectStatus() {}
    public void conductInterview() {
        System.out.println("Mgr - conductInterview");
    }
    public void attendTraining(String[] trainingSchedule) {
        System.out.println("Mgr - attendTraining");
    }
}
class Programmer extends Employee implements Trainable{
    String[] programmingLanguages;
    void writeCode() {}
    public void attendTraining(String[] trainingSchedule) {
        System.out.println("Prog - attendTraining");
    }
}
```

Signature of method attendTraining should match in interface Trainable, class Manager, and class Programmer

EXAM TIP The method signatures of a method defined in an interface and the classes that implement the interface must match, or the classes won't compile.

6.2.1 Properties of members of an interface

An interface can only define constants. Once it's assigned, you can't change the value of a constant. The variables of an interface are implicitly `public`, `final`, and `static`.

For example, the following definition of the interface `MyInterface`,

```
interface MyInterface {
    int age = 10;
}
```

is equivalent to the following definition:

```
interface MyInterface {
    public static final int AGE = 10;
}
```

public, static, and final modifiers are implicitly added to variables defined in an interface.

You should initialize all variables in an interface, or your code won't compile:

```
interface MyInterface {
    int AGE;
}
```

Won't compile. Should assign a value to a final variable.

The methods of an interface are implicitly `public`. When you implement an interface, you must implement all its methods by using the access modifier `public`. A class that implements an interface can't make the interface's methods more restrictive. Although the following class and interface definitions look acceptable, they're not:

```
interface Relocatable {
    void move();
}
class CEO implements Relocatable {
    void move() {}
}
```

Implicitly public

Won't compile. Can't assign weaker access (default access) to public method move in class CEO.

The following code is correct and compiles happily:

```
interface Relocatable {
    void move();
}
class CEO implements Relocatable {
    public void move() {}
}
```

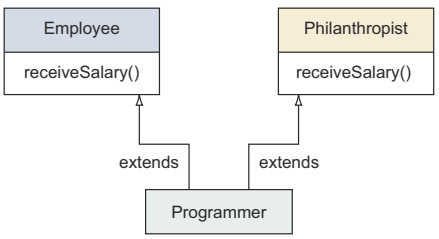
Implicitly public

Will compile

Unlike a class, an interface can't define constructors.

6.2.2 Why a class can't extend multiple classes

In Java, a class can't extend multiple classes. Why do you think this is so? Let's examine this issue using an example, in which the class `Programmer` *is* allowed to inherit two classes: `Employee` and `Philanthropist`. Figure 6.9 shows the relationship between these classes and the corresponding code.




```
class Employee {
    public void receiveSalary() {
        system.out.println("PayDues");
    }
}
class Philanthropist {
    public void receiveSalary() {
        system.out.println("Donate");
    }
}
class Programmer extends Employee, Philanthropist {}
```

Figure 6.9 What happens if a class is allowed to extend multiple classes?

If class `Programmer` inherited the method `receiveSalary`, defined in both `Employee` and `Philanthropist`, what do you think a `Programmer` would do with his or her salary: pay dues (like an `Employee`) or donate it (like a `Philanthropist`)? What do you think would be the output of the following code?

```
class Test {
    public static void main(String args[]) {
        Programmer p = new Programmer();
        p.receiveSalary();
    }
}
```

Would this print "PayDues" or "Donate"?

In this case, class `Programmer` can access two `receiveSalary` methods with identical method signatures but different implementations, so it is impossible to resolve this method call. This is why classes aren't allowed to inherit multiple classes in Java.

EXAM TIP Because a derived class may inherit different implementations for the same method signature from multiple base *classes*, multiple inheritance is not allowed in Java.

6.2.3 Implementing multiple interfaces

In the previous section, you learned that a class can't inherit multiple classes, but a class can implement multiple interfaces. Also, an interface can extend multiple interfaces.

Why is this allowed, when Java doesn't allow a class to extend multiple classes? What's the catch?

An interface defines barebones methods—methods without any body—and when an interface extends another interface, it inherits the methods defined in the base interface. What happens if the base interface and subinterface define methods with the same signatures? Or when an interface extends more than one interface that defines the same method?

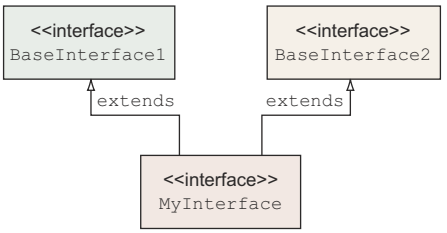


Figure 6.10 The interface `MyInterface` extends the interfaces `BaseInterface1` and `BaseInterface2`.

Consider the following code, whose UML representation is shown in figure 6.10. Which of the `getName` methods will be inherited by the interface `MyInterface`? Will `MyInterface` inherit the `getName` method defined in `BaseInterface1` or the one defined in `BaseInterface2`?

```
interface BaseInterface1 {
    String getName();
}
interface BaseInterface2 {
    String getName();
}
interface MyInterface extends BaseInterface1, BaseInterface2 {}
```

Because neither of the `getName` methods defined in `BaseInterface1` and `BaseInterface2` define a method body (as shown in figure 6.11), the question of which of the methods `MyInterface` inherits is irrelevant. Interface `MyInterface` has access to a single `getName` method, which should be implemented by all the classes that implement `MyInterface`.

Let's make the `Employee` class implement the interface `MyInterface`, as follows:

```
class Employee implements MyInterface {
    String name;
    public String getName() {
        return name;
    }
}
```

`Employee` defines a body for method `getName`, inherited from interface `MyInterface`

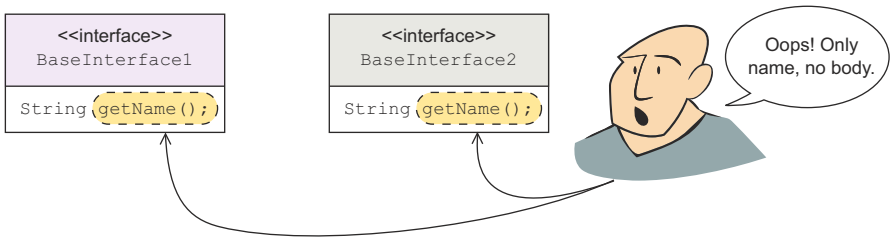


Figure 6.11 Methods defined in an interface don't have a method body.

POINTS TO NOTE ABOUT CLASS AND INTERFACE INHERITANCE:

- A class can inherit zero or one class.
- A class uses the keyword `extends` to inherit a class.
- A class can implement multiple interfaces.
- A class uses the keyword `implements` to implement an interface.
- An interface can't implement any class.
- An interface can inherit zero or more interfaces.
- An interface uses the keyword `extends` to inherit interfaces.
- An abstract class can extend a concrete class and vice versa.
- An abstract class can implement interfaces.
- An abstract class can extend another abstract class.
- The first concrete class in the hierarchy must supply actual method implementations for all `abstract` methods.

You can use a reference variable of a base class to refer to an object of its derived class. Similarly, you can use a reference variable of an interface to refer to an object of a class that implements it. It's interesting to note that these variables can't access all the variables and methods defined in the derived class or the class that implements the interface.

Let's dig into some more details about this in the next section.

6.3 Reference variable and object types

[7.3] Differentiate between the type of a reference and the type of an object

For this exam objective, you need to understand that when you refer to an object, the type of the *object reference variable* and the type of the *object* being referred to may be different. But there are rules on *how different* these can be. This concept may take a while to sink in, so don't worry if you don't get it on your first attempt.

In the same way in which you can refer to a person using their first name, last name, or both names, objects of derived classes can be referred to using a reference variable of either of the following types:

- *Its own type*—An object of a class `HRExecutive` can be referred to using an object reference variable of type `HRExecutive`.
- *Its base class*—If the class `HRExecutive` inherits the class `Employee`, an object of the class `HRExecutive` can be referred to using a variable of type `Employee`. If the class `Employee` inherits the class `Person`, an object of the class `HRExecutive` can also be referred to using a variable of type `Person`.
- *Implemented interfaces*—If the class `HRExecutive` implements the interface `Interviewer`, an object of the class `HRExecutive` can be referred using a variable of type `Interviewer`.

There are differences, however, when you try to access an object using a reference variable of its own type, its base class, or an implemented interface. Let's start with accessing an object with a variable of its own type.

6.3.1 Using a variable of the derived class to access its own object

Let's start with the code of the class `HRExecutive`, which inherits the class `Employee` and implements the interface `Interviewer`, as follows:

```
class Employee {
    String name;
    String address;
    String phoneNumber;
    float experience;
}
interface Interviewer {
    public void conductInterview();
}
class HRExecutive extends Employee implements Interviewer {
    String[] specialization;
    public void conductInterview() {
        System.out.println("HRExecutive - conducting interview");
    }
}
```

Class HRExecutive inherits class Employee and implements interface Interview

Here's some code that demonstrates that an object of class `HRExecutive` can be referred to using a variable of type `HRExecutive`:

```
class Office {
    public static void main(String args[]) {
        HRExecutive hr = new HRExecutive();
    }
}
```

A variable of type HRExecutive can be used to refer to its object.

You can access variables and methods defined in the class `Employee`, the class `HRExecutive`, and the interface `Interviewer` using the variable `hr` (with the type `HRExecutive`), as follows:

```
class Office {
    public static void main(String args[]) {
        HRExecutive hr = new HRExecutive();

        hr.specialization = new String[] {"Staffing"};
        System.out.println(hr.specialization[0]);

        hr.name = "Pavni Gupta";
        System.out.println(hr.name);

        hr.conductInterview();
    }
}
```

Access variable defined in class HRExecutive

Access variable defined in class Employee

Access method defined in interface Interviewer

When you access an object of the class `HRExecutive` using its own type, you can access all the variables and methods that are defined in its base class and interface—the class

Employee and the interface Interviewer. Can you do the same if the type of the reference variable is changed to the class Employee, as defined in the next section?

6.3.2 Using a variable of the base class to access an object of a derived class

Let's access an object of type HRExecutive using a reference variable of type Employee, as follows:

```
class Office {
    public static void main(String args[]) {
        Employee emp = new HRExecutive();
    }
}
```

Variable of type Employee can also be used to refer to an object of class HRExecutive because class HRExecutive extends Employee.

Now let's see whether changing the type of the reference variable makes any difference when accessing the members of the class Employee, the class HRExecutive, or the interface Interviewer. Will the following code compile successfully?

Type of variable emp is Employee.

```
class Office {
    public static void main(String args[]) {
        Employee emp = new HRExecutive();

        emp.specialization = new String[] {"Staffing"};
        System.out.println(emp.specialization[0]);

        emp.name = "Pavni Gupta";
        System.out.println(emp.name);

        emp.conductInterview();
    }
}
```

❶ **Variable emp can't access member specialization defined in class HRExecutive.**

Variable emp can access member name defined in class Employee.

Variable emp can't access method conductInterview defined in interface Interviewer.

The code at ❶ fails to compile because the type of the variable emp is defined as Employee. Picture it like this: the variable emp can see only the Employee object. Hence, it can access only the variables and methods defined in the class Employee, as illustrated in figure 6.12.

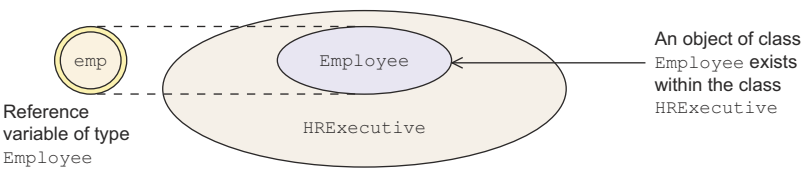


Figure 6.12 A variable of type Employee can see only the members defined in the class Employee.

6.3.3 Using a variable of an implemented interface to access a derived class object

Here's another interesting equation: what happens when you change the type of the reference variable to the interface Interviewer? A variable of type Interviewer can

also be used to refer to an object of the class `HRExecutive` because the class `HRExecutive` implements `Interviewer`. See the following code:

```
class Office {
    public static void main(String args[]) {
        Interviewer interviewer = new HRExecutive();
    }
}
```

Now try to access the same set of variables and methods using the variable `interviewer`, which refers to an object of the class `HRExecutive`:

```
class Office {
    public static void main(String args[]) {
        Interviewer interviewer = new HRExecutive();

        interviewer.specialization = new String[] {"Staffing"};
        System.out.println(interviewer.specialization[0]);

        interviewer.name = "Pavni Gupta";
        System.out.println(interviewer.name);

        interviewer.conductInterview();
    }
}
```

Type of variable interviewer is Interviewer.

Variable interviewer can't access members of class Employee or HRExecutive.

Variable interviewer can access the method conductInterview defined in interface Interviewer.

The code at ❶ doesn't compile because the type of the variable `interviewer` is defined as `Interviewer`. Picture it like this: the variable `interviewer` can only *access* the methods defined in the interface `Interviewer`, as illustrated in figure 6.13.

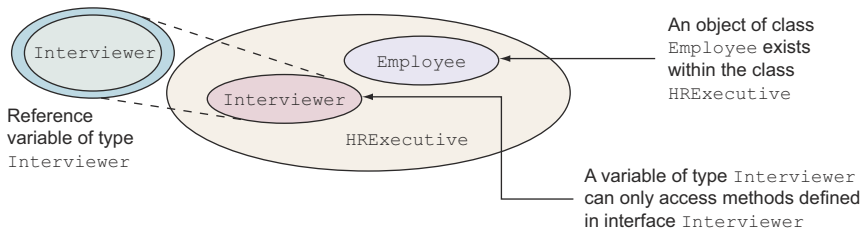


Figure 6.13 A variable of type `Interviewer` can see only the members defined in the interface `Interviewer`.

6.3.4 The need for accessing an object using the variables of its base class or implemented interfaces

You may be wondering why you need a reference variable of a base class or an implemented interface to access an object of a derived class if a variable can't access all the members that are available to an object of a derived class? The simple answer is that you might not be interested in *all* the members of a derived class.

Confused? Compare it with the following situation. When you enroll in flying classes, do you care whether the instructor can cook Italian cuisine or knows how to swim? No! You don't care about characteristics and behavior that are unrelated to flying. Here is

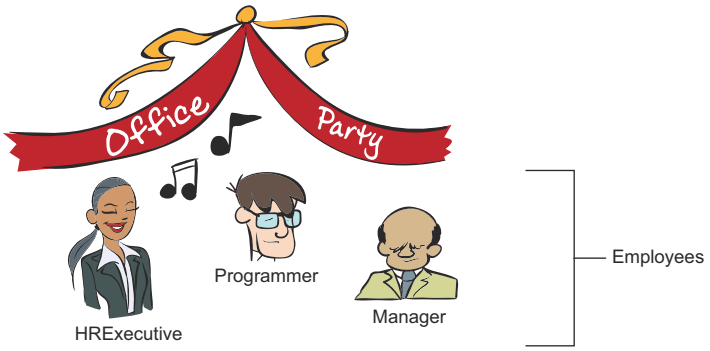


Figure 6.14 All types of employees can attend an office party.

another example. At an office party, all the employees are welcome, whether they are `Programmers`, `HRExecutives`, or `Managers`, as shown in figure 6.14.

The same logic applies when you access an object of the class `HRExecutive` using a reference variable of type `Interviewer`. When you do so, you're only concerned about the behavior of `HRExecutive` that relates to its capability as an `Interviewer`.

This arrangement also makes it possible to create an array (or a list) of the objects that refers to different types of objects grouped by a common base class or an interface. The following code segment defines an array of type `Interviewer` and stores in it objects of the classes `HRExecutive` and `Manager`:

```
class OfficeInheritanceList {
    public static void main(String args[]) {
        Interviewer[] interviewers = new Interviewer[2];

        interviewers[0] = new Manager();

        interviewers[1] = new HRExecutive();

        for (Interviewer interviewer : interviewers) {
            interviewer.conductInterview();
        }
    }
}
```

Array of type Interviewer—an interface

Because Manager implements interface Interviewer, it can be stored here.

Because HRExecutive implements interface Interviewer, it can be stored here.

Loop through the array and call method conductInterview

The class `HRExecutive` extends the class `Employee` and implements the interface `Interviewer`. Hence, you can assign an object of `HRExecutive` to any of the following types of variables:

- `HRExecutive`
- `Employee`
- `Interviewer`

Please note that the reverse of these assignments will fail compilation. To start with, you can't refer to an object of a base class by using a reference variable of its derived class. Because *all* members of a derived class can't be accessed using an object of the base class, it isn't allowed. The following statement will not compile:

```
HRExecutive hr = new Employee();
```

Not allowed—won't compile

Because you can't create an object of an interface, the following line of code will also fail to compile:

```
HRExecutive hr = new Interviewer();          <-- Not allowed—won't compile
```

It's now time for you to try to add objects of the previously defined related classes—`Employee`, `Manager`, and `HRExecutive`—to an array in your next Twist in the Tale exercise (answers in the appendix).

Twist in the Tale 6.2

Given the following definition of the classes and interface `Employee`, `Manager`, `HRExecutive`, and `Interviewer`, select the correct options for the class `TwistInTale2`:

```
class Employee {}
interface Interviewer {}
class Manager extends Employee implements Interviewer {}
class HRExecutive extends Employee implements Interviewer {}

class TwistInTale2 {
    public static void main (String args[]) {
        Interviewer[] interviewer = new Interviewer[] {
                new Manager(),           // Line 1
                new Employee(),          // Line 2
                new HRExecutive(),       // Line 3
                new Interviewer()        // Line 4
            };
    }
}
```

a An object of the class `Manager` can be added to an array of the interface `Interviewer`. Code on line 1 will compile successfully.

b An object of the class `Employee` can be added to an array of the interface `Interviewer`. Code on line 2 will compile successfully.

c An object of the class `HRExecutive` can be added to an array of the interface `Interviewer`. Code on line 3 will compile successfully.

d An object of the interface `Interviewer` can be added to an array of the interface `Interviewer`. Code on line 4 will compile successfully.

EXAM TIP You may see multiple questions in the exam that try to assign an object of a base class to a reference variable of a derived class. Note that a derived class can be referred to using a reference variable of its base class. The reverse is not allowed and won't compile.

In this section, you learned that the variables of a base class or interface are not able to access all the members of the object to which they refer. Don't worry; this can be resolved by *casting* a reference variable of a base class or an interface to the exact type of the object they refer to, as discussed in the next section.

6.4 Casting

[7.4] Determine when casting is necessary

Casting is the process of forcefully making a variable behave as a variable of another type. If a class shares an IS-A or inheritance relationship with another class or interface, their variables can be cast to each other's type.

In section 6.3, you learned that you can't access all the members of the class `HRExecutive` (derived class) if you refer to it via a variable of type `Interviewer` (implemented interface) or `Employee` (base class). In this section, you'll learn how to cast a variable of type `Interviewer` to access variables defined in the class `HRExecutive`, and why you'd want to.

6.4.1 How to cast a variable to another type

We'll start with the definitions of interface `Interviewer` and classes `HRExecutive` and `Manager`:

```
class Employee {}
interface Interviewer {
    public void conductInterview();
}
class HRExecutive extends Employee implements Interviewer {
    String[] specialization;
    public void conductInterview() {
        System.out.println("HRExecutive - conducting interview");
    }
}
class Manager implements Interviewer{
    int teamSize;
    public void conductInterview() {
        System.out.println("Manager - conducting interview");
    }
}
```

Create a variable of type `Interviewer` and assign to it an object of type `HRExecutive` (as depicted in figure 6.15).

```
Interviewer interviewer = new HRExecutive();
```

Try to access the variable `specialization` defined in the class `HRExecutive` using the previous variable:

```
interviewer.specialization = new String[] {"Staffing"};
```

⟵ **Won't compile**

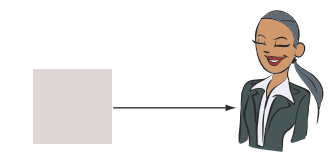


Figure 6.15 A reference variable of interface `Interviewer` referring to an object of the class `HRExecutive`

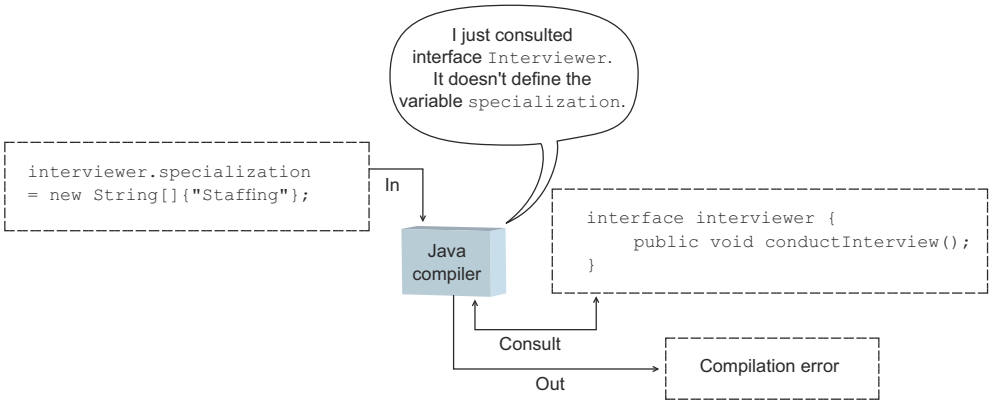


Figure 6.16 The Java compiler doesn't compile code if you try to access the variable `specialization`, defined in the class `HRExecutive`, by using a variable of the interface `Interviewer`.

The previous line of code won't compile. The compiler knows that the type of the variable `interviewer` is `Interviewer` and that the interface `Interviewer` doesn't define any variable with the name `specialization` (as shown in figure 6.16).

On the other hand, the JRE knows that the object referred to by the variable `interviewer` is of type `HRExecutive`, so you can use casting to get past the Java compiler and access the members of the object being referred to, as follows (see also figure 6.17):

```
((HRExecutive)interviewer).specialization = new String[] {"Staffing"};
```

In the previous example code, `(HRExecutive)` is placed just before the name of the variable, `interviewer`, to cast it to `HRExecutive`. A pair of parentheses surrounds

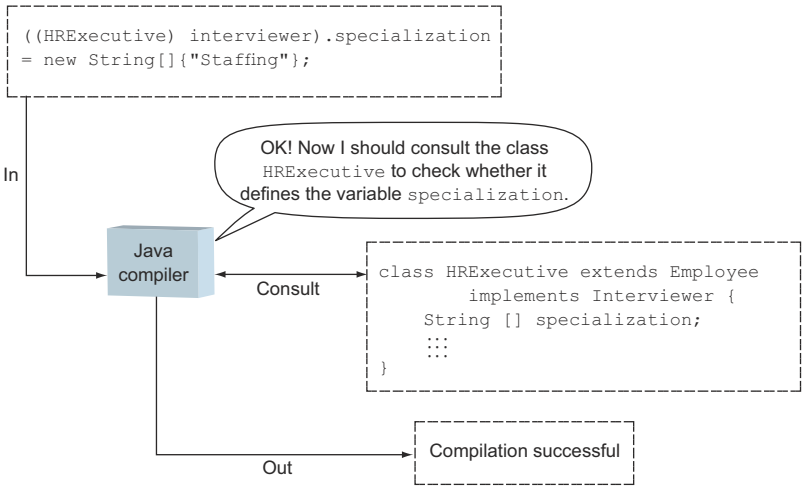


Figure 6.17 Casting can be used to access the variable `specialization`, defined in the class `HRExecutive`, by using a variable of interface `Interviewer`.

`(HRExecutive)interviewer`, which lets Java know you're sure that the object being referred to is an object of the class `HRExecutive`. Casting is another method of telling Java: "Look, I know that the actual object being referred to is `HRExecutive`, even though I'm using a reference variable of type `Interviewer`."

6.4.2 Need for casting

In section 6.3.4, I discussed the need to use a reference variable of an inherited class or an implemented interface to refer to an object of a derived class. I also used an example of enrolling in flying classes, where you don't care about whether the instructor can cook Italian cuisine or knows how to swim. You don't care about characteristics and behavior that are unrelated to flying.

But think about a situation in which you do care about the swimming skills of your instructor. Imagine that when you're attending flying classes, you friend enquires about whether your flying instructor also conducts swimming classes, and if so, whether your friend could enroll. In this case, a *need* arises to enquire about the swimming skills of your flying instructor.

Let's apply this situation to Java. You can't access all the members of an object if you access it using a reference variable of any of its implemented interfaces or of a base class. But if a need arises, you *might* choose to access some of the members of a derived class, which are not explicitly available, by using the reference variable of the base type or the implemented interface. This is where casting comes in!

Time to see this in code. Here's an example that exhibits the need for casting. An application maintains a list of interviewers, and depending on the type of interviewer (`HRExecutive` or `Manager`), it performs a different set of actions. If the interviewer is a `Manager`, the code calls `conductInterview` only if the value for the Manager's `teamSize` is greater than `10`. Here's the code that implements this logic:

```
class OfficeWhyCasting {
    public static void main(String args[]) {
        Interviewer[] interviewers = new Interviewer[2];

        interviewers[0] = new Manager();

        interviewers[1] = new HRExecutive();

        for (Interviewer interviewer : interviewers) {

            if (interviewer instanceof Manager) {
                int teamSize =((Manager)interviewer).teamSize;

                if (teamSize > 10) {
                    interviewer.conductInterview();
                }
                else {
                    System.out.println("Mgr can't " +
                          "interview with team size less than 10");
                }
            }
```

Array to store objects of classes that implement interface Interviewer

Store object of HRExecutive at array position 1

Store object of Manager at array position 0

Loop through values of array interviewers

If object referred to by interviewer is of class Manager, use casting to retrieve value for its teamSize

If interviewer's teamSize > 10, call conductInterview

If interviewer's teamSize <= 10, print message

```
            else if (interviewer instanceof HRExecutive) {
                interviewer.conductInterview();
            }
        }
    }
}
```

Otherwise, if object stored is of class HRExecutive, call conductInterview method on the object; no casting is required in this case

6.5 Use this and super to access objects and constructors

[7.5] Use super and this to access objects and constructors

In this section, you'll use the `this` and `super` keywords to access objects and constructors. `this` and `super` are *implicit* object references. These variables are defined and initialized by the JVM for every object in its memory.

Let's examine the capabilities and use of each of these reference variables.

6.5.1 Object reference: this

The `this` reference always points to an object's *own instance*. Any object can use the `this` reference to refer to its own instance. Think of the words *me*, *myself*, and *I*: anyone using those words is always referring to themselves, as shown in figure 6.18.

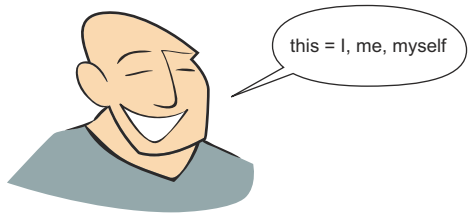


Figure 6.18 The keyword `this` can be compared to the words *me*, *myself*, and *I*.

USING THIS TO ACCESS VARIABLES AND METHODS

You can use the keyword `this` to refer to all methods and variables that are accessible to a class. For example, here's a modified definition of the class `Employee`:

```
class Employee {
    String name;
}
```

The variable `name` can be accessed in the class `Programmer` (which extends the class `Employee`) as follows:

```
class Programmer extends Employee {
    void accessEmployeeVariables() {
         name = "Programmer";
    }
}
```

Because there exists an object of class `Employee` within the class `Programmer`, the variable `name` is accessible to an object of `Programmer`. The variable `name` can also be accessed in the class `Programmer` as follows:

```
class Programmer extends Employee {
    void accessEmployeeVariables() {
         this.name = "Programmer";
    }
}
```

The `this` reference may be used only when code executing within a method block needs to differentiate between an instance variable and its local variable or method parameters. But some developers use the keyword `this` all over their code, even when it's not required. Some use `this` as a means to differentiate instance variables from local variables or method parameters.

Figure 6.19 shows the constructor of the class `Employee`, which uses the reference variable `this` to differentiate between local and instance variables `name`, which are declared with the same name.

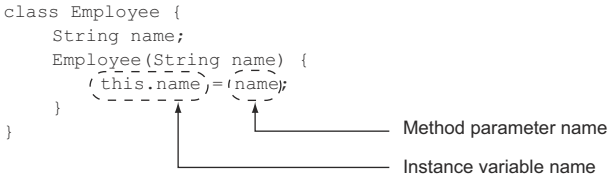


Figure 6.19 Using the keyword `this` to differentiate between method parameter and instance variables

In the previous example, the class `Employee` defines an instance variable with the name `name`. The `Employee` class constructor also defines a method parameter `name`, which is effectively a local variable defined within the scope of the method block. Hence, within the scope of the previously defined `Employee` constructor, there's a clash of names, and the local variable will take precedence. Using `name` within the scope of the `Employee` class constructor block will implicitly refer to that method's parameter, not the instance variable. In order to refer to the instance variable `name` from within the scope of the `Employee` class constructor, you are obliged to use a `this` reference.

USING THIS TO ACCESS CONSTRUCTORS

You can also differentiate one constructor from another by using the keyword `this`. Here's an example in which the class `Employee` defines two constructors, with the second constructor calling the first one:

```
class Employee {
    String name;
    String address;

    Employee(String name) {
        this.name = name;
    }

    Employee(String name, String address) {
        this(name);

        this.address = address;
    }
}
```

Instance variables are name and address

Constructor that accepts only name

Constructor that accepts name and address

Calls constructor that accepts only name

Assigns value of method parameter address to instance variable

To call the default constructor (one that doesn't accept any method parameters), call `this()`. Here's an example:

```
class Employee {
    String name;
    String address;

    Employee() {
        name = "NoName";
        address = "NoAddress";
    }

    Employee(String name, String address) {
        this();

        if (name != null) this.name = name;
        if (address != null) this.address = address;
    }
}
```

Instance variables are name and address

Constructor that doesn't accept any arguments

Calls constructor that doesn't accept any arguments. Must be the first statement in this method.

Constructor that accepts name and address

Assigns value of not null method parameters

If present, a call to a constructor from another constructor must be done on the first line of code of the calling constructor.

EXAM TIP `this` refers to the instance of the class in which it's used. `this` can be used to access the inherited members of a base class in the derived class.

6.5.2 Object reference: super

In the previous section, I discussed how `this` refers to the object instance itself. Similarly, `super` is also an object reference, but `super` refers to the parent or base class of a class. Think of the words *my parent*, *my base*: anyone using those terms is always referring to their parent or the base class, as shown in figure 6.20.

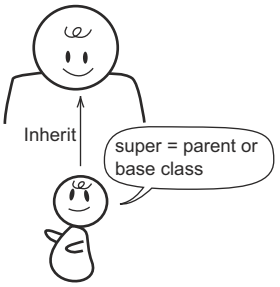


Figure 6.20 When a class mentions `super`, it refers to its parent or the base class.

USING SUPER TO ACCESS VARIABLES AND METHODS OF THE BASE CLASS

The variable reference `super` can be used to access a variable or method from the base class if there's a clash between these names. This situation normally occurs when a derived class defines variables and methods with the same name as the base class.

Here's an example:

```
class Employee {
    String name;
}
class Programmer extends Employee {
    String name;

    void setNames() {
        this.name = "Programmer";

        super.name = "Employee";
    }
```

Instance variable—name, in Employee

Instance variable—name, in Programmer

Assign value to instance variable—name, defined in Programmer

Assign value to instance variable—name, defined in Employee

```
    void printNames() {
        System.out.println(super.name);

        System.out.println(this.name);
    }
}
class UsingThisAndSuper {
    public static void main(String[] args) {
        Programmer programmer = new Programmer();
        programmer.setNames();
        programmer.printNames();
    }
}
```

Print value of instance variable—name, defined in Employee

Print value of instance variable—name, defined in Programmer

Create an object of class Programmer

The output of the preceding code is as follows:

```
Employee
Programmer
```

Similarly, you can use the reference variable `super` to access a method defined with the same name in the base or the parent class.

USING SUPER TO ACCESS CONSTRUCTORS OF BASE CLASS

The reference variable `super` can also be used to refer to the constructors of the base class in a derived class.

Here's an example in which the base class, `Employee`, defines a constructor that assigns default values to its variables. Its derived class calls the base class constructor in its own constructor.

```
class Employee {
    String name;
    String address;

    Employee(String name, String address) {
        this.name = name;
        this.address = address;
    }
}
class Programmer extends Employee {
    String progLanguage;

    Programmer(String name, String address, String progLang) {
        super(name, address);
        this.progLanguage = progLang;
    }
}
```

Instance variables—name and address

Constructor that accepts name and address

Instance variable—progLanguage

Constructor that accepts values for superclass variables also

Calls Employee constructor ❶

The code at ❶ calls the superclass constructor by passing it the reference variables, `name` and `address`, which it accepts itself.

EXAM TIP If present, a call to a superclass's constructor must be the first statement in a derived class's constructor. Otherwise, a call to `super();` (the no-arg constructor) is inserted automatically by the compiler.

USING SUPER AND THIS IN STATIC METHODS

The keywords `super` and `this` are implicit object references. Because `static` methods belong to a class, not to objects of a class, you can't use `this` and `super` in `static` methods. Code that tries to do so won't compile:

```
class Employee {
    String name;
}
class Programmer extends Employee {
    String name;

    static void setNames() {
        this.name = "Programmer";

        super.name = "Employee";
    }
}
```

Instance variable—name, in Employee

Instance variable—name, in Programmer

Won't compile—can't use this in static method

Won't compile—can't use super in static method

It's time to attempt the next Twist in the Tale exercise, using `this` and `super` keywords (answer in the appendix).

Twist in the Tale 6.3

Let's modify the definition of the `Employee` and `Programmer` classes as follows. What is the output of class `TwistInTale3`?

```
class Employee {
    String name = "Emp";
    String address = "EmpAddress";
}
class Programmer extends Employee{
    String name = "Prog";
    void printValues() {
        System.out.print(this.name + ":");
        System.out.print(this.address + ":");
        System.out.print(super.name + ":");
        System.out.print(super.address);
    }
}
class TwistInTale3 {
    public static void main(String args[]) {
        new Programmer().printValues();
    }
}
```

a `Prog:null:Emp:EmpAddress`

b `Prog:EmpAddress:Emp:EmpAddress`

c `Prog::Emp:EmpAddress`

d Compilation error

6.6 Polymorphism

[7.6] Use abstract classes and interfaces

[7.2] Develop code that demonstrates the use of polymorphism

The literal meaning of the word "polymorphism" is "many forms." At the beginning of this chapter, I used a practical example to explain the meaning of polymorphism; the same action may have different meanings for different living beings. The action *eat* has a different meaning for a *fly* and a *lion*. A *fly* may eat *nectar*, whereas a *lion* may eat an antelope. Reacting to the same action in one's own unique manner in living beings can be compared to polymorphism in Java.

For the exam, you need to know what polymorphism in Java is, why you need it, and how to implement it in code.

6.6.1 Polymorphism with classes

Polymorphism comes into the picture when a class inherits another class and both the base and the derived classes define methods with the same method signature (the same method name and method parameters). As discussed in the previous section, an object can also be referred to using a reference variable of its base class. In this case, depending upon the type of the object used to execute a method, the Java runtime executes the method defined in the base or derived class.

Let's consider polymorphism using the classes `Employee`, `Programmer`, and `Manager`, where the classes `Programmer` and `Manager` inherit the class `Employee`. Figure 6.21 shows a UML diagram depicting the relationships among these classes.

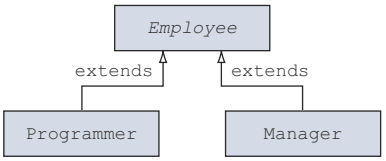


Figure 6.21 Relationships among the classes `Employee`, `Programmer`, and `Manager`

Let's start with the `Employee` class, which is not quite sure about what must be done to start work on a project (execute method `startProjectWork`). Hence, the method `startProjectWork` is defined as an `abstract` method, and the class `Employee` is defined as an `abstract` class, as follows:

```
abstract class Employee {
        public void reachOffice() {
        System.out.println("reached office - Gurgaon, India");
    }
    public abstract void startProjectWork();
}
```

Doesn't know how to work on a project

The class `Programmer` extends the class `Employee`, which essentially means that it has access to the method `reachOffice` defined in `Employee`. `Programmer` must also implement the `abstract` method `startProjectWork`, inherited from `Employee`. How do you think a programmer will typically start work on a programming project? Most probably, he will define classes and unit test them. This behavior is contained in the

definition of the class `Programmer`, which implements the method `startProjectWork`, as follows:

```
class Programmer extends Employee {
    public void startProjectWork() {
        defineClasses();
        unitTestCode();
    }
    private void defineClasses() { System.out.println("define classes"); }
    private void unitTestCode() { System.out.println("unit test code"); }
}
```

We are fortunate to have another special type of employee, a manager, who knows how to start work on a project. How do you think a manager will typically start work on a programming project? Most probably, she will meet with the customers, define a project schedule, and assign work to the team members. Here's the definition of class `Manager` that extends class `Employee` and implements the method `startProjectWork`:

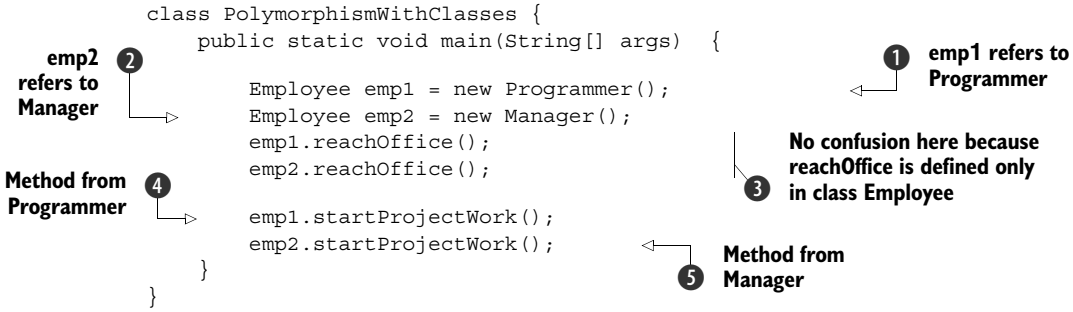
```
class Manager extends Employee {
    public void startProjectWork() {
        meetingWithCustomer();
        defineProjectSchedule();
        assignRespToTeam();
    }
    private void meetingWithCustomer() {
        System.out.println("meet Customer");
    }
    private void defineProjectSchedule() {
        System.out.println("Project Schedule");
    }
    private void assignRespToTeam() {
        System.out.println("team work starts");
    }
}
```

Let's see how this method behaves with different types of employees. Here's the relevant code:

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

        Employee emp1 = new Programmer();
        Employee emp2 = new Manager();
        emp1.reachOffice();
        emp2.reachOffice();

        emp1.startProjectWork();
        emp2.startProjectWork();
    }
}
```



Here's the output of the code (blank lines added for clarity):

```
reached office - Gurgaon, India
reached office - Gurgaon, India
```

```
define classes
unit test code

meet Customer
Project Schedule
team work starts
```

The code at ❶ creates an object of the class `Programmer` and assigns it to a variable of type `Employee`. ❷ creates an object of the class `Manager` and assigns it to a variable of type `Employee`. So far, so good!

Now comes the complicated part. ❸ executes the method `reachOffice`. Because this method is defined only in the class `Employee`, there isn't any confusion and the same method executes, printing the following:

```
reached office - Gurgaon, India
reached office - Gurgaon, India
```

The code at ❹ executes the code `emp1.startProjectWork()` and calls the method `startProjectWork` defined in the class `Programmer`, because `emp1` refers to an object of the class `Programmer`. Here's the output of this method call:

```
define classes
unit test code
```

The code at ❺ executes `emp2.startProjectWork()` and calls the method `startProjectWork` defined in the class `Manager`, because `emp2` refers to an object of the class `Manager`. Here's the output of this method call:

```
meet Customer
Project Schedule
team work starts
```

Figure 6.22 illustrates this code.

As discussed in the beginning of this section, the usefulness of polymorphism lies in the ability of an object to behave in its own specific manner when the same action is passed to it. In the previous example, reference variables (`emp1` and `emp2`) of type `Employee` are used to store objects of class `Programmer` and `Manager`. When the same action—that is, method call `startProjectWork`—is invoked on these reference variables (`emp1` and `emp2`), each method call results in the method defined in the respective classes being executed.

POLYMORPHIC METHODS ARE ALSO CALLED OVERRIDDEN METHODS

Take a quick look at the method `startProjectWork`, as defined in the following classes `Employee`, `Programmer`, and `Manager` (only the relevant code is shown):

```
abstract class Employee {
    public abstract void startProjectWork();
}
class Programmer extends Employee {
    public void startProjectWork() {
        ...
    }
}
class Manager extends Employee {
```

Method startProjectWork in class Employee

Method startProjectWork in class Programmer

```
    public void startProjectWork() {
        ...
    }
}
```

Method startProjectWork in class Manager

Note that the name of method `startProjectWork` is same in all these classes. Also, it accepts the same number of method arguments and defines the same return type in all the three classes: `Employee`, `Programmer`, and `Manager`. This is a contract specified to define overridden methods. Failing to use the same method name, same argument list, or same return type won't mark a method as an overridden method.

RULES TO REMEMBER WHEN DEFINING OVERRIDDEN METHODS

- Overridden methods are defined by classes and interfaces that share inheritance relationships.
- The name of the overridden method must be the same in both the base class and the derived class.
- The argument list passed to the overridden method must be the same in both the base class and derived class.
- The return type of an overriding method in the derived class can be the same, or a subclass of the return type of the overridden method in the base class. When the overriding method returns a subclass of the return type of the overridden method, it is known as a *covariant return type*.
- An overridden method defined in the base class can be an `abstract` method or a non-`abstract` method.
- Access modifiers for an overriding method can be the same or less restrictive than the method being overridden, but they can't be more restrictive.

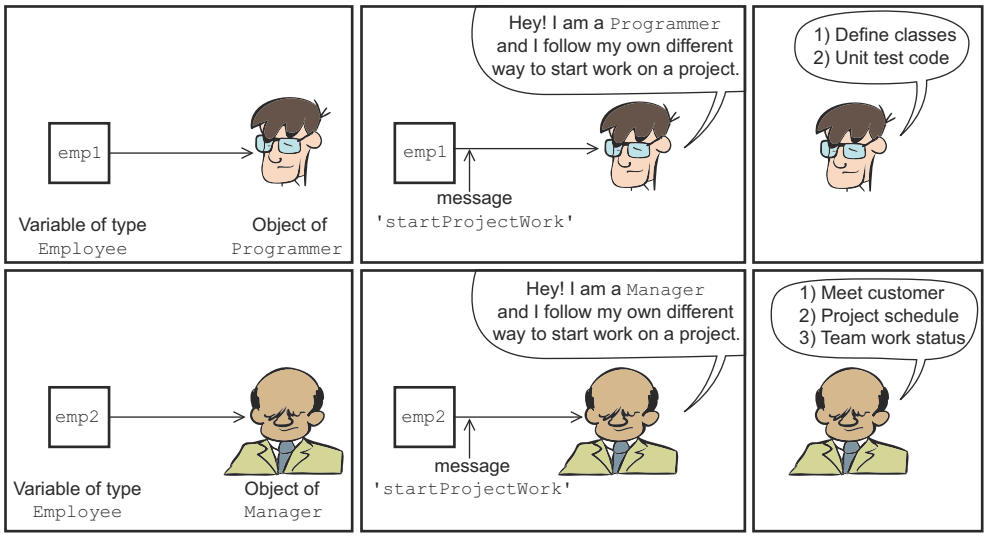


Figure 6.22 The objects are aware of their own type and execute the overridden method defined in their own class, even if a base class variable is used to refer to them.

DO POLYMORPHIC METHODS ALWAYS HAVE TO BE ABSTRACT?

No, polymorphic methods don't always have to be `abstract`. You can define the class `Employee` as a concrete class and the method `startProjectWork` as a non-`abstract` method and still get the same results (changes in bold):

```
class Employee {
    public void reachOffice() {
        System.out.println("reached office - Gurgaon, India");
    }
    public void startProjectWork() {
        System.out.println("procure hardware");
        System.out.println("install software");
    }
}
```

Because there's no change in the definition of the rest of the classes—`Programmer`, `Manager`, and `PolymorphismWithClasses`—I haven't listed them here. If you create an object of the class `Employee` (not of any of its derived classes), you can execute the method `startProjectWork` as follows:

```
Employee emp = new Employee();
emp.startProjectWork();
```

Object of class Employee, not its derived classes

Call method startProjectWork, defined in Employee

EXAM TIP To implement polymorphism with classes, you can define `abstract` or non-`abstract` methods in the base class and override them in the derived classes.

CAN POLYMORPHISM WORK WITH OVERLOADED METHODS?

No, polymorphism works only with overridden methods. Overridden methods have the same number and type of method arguments, whereas overloaded methods define a method argument list with either a different number or type of method parameters.

Overloaded methods only share the same name; the JRE treats them like different methods. In the case of overridden methods, the JRE decides at runtime which method should be called based on the exact type of the object on which it's called.

It's time for the next Twist in the Tale exercise. As usual, you can find the answers in the appendix.

Twist in the Tale 6.4

Given the following definition of classes `Employee` and `Programmer`, which of the options when inserted at `//INSERT CODE HERE//` will define the method `run` as a polymorphic method?

```
class Employee {
    //INSERT CODE HERE// {
        System.out.println("Emp-run");
        return null;
    }
}
class Programmer extends Employee{
    String run() {
        System.out.println("Programmer-run");
```

```
        return null;
    }
}
class TwistInTale4 {
    public static void main(String args[]) {
        new Programmer().run();
    }
}
```

a `String run()`

b `void run(int meters)`

c `void run()`

d `int run(String race)`

6.6.2 Binding of variables and methods at compile time and runtime

You can use reference variables of a base class to refer to an object of a derived class. But there's a major difference in how Java accesses the variables and methods for these objects. With inheritance, the instance variables bind at compile time and the methods bind at runtime.

Examine the following code:

```
class Employee {
    String name = "Employee";
    void printName() {
        System.out.println(name);
    }
}
class Programmer extends Employee {
    String name = "Programmer";
    void printName() {
        System.out.println(name);
    }
}
class Office1 {
    public static void main(String[] args) {
        Employee emp = new Employee();
        Employee programmer = new Programmer();

        System.out.println(emp.name);
        System.out.println(programmer.name);
        emp.printName();
        programmer.printName();
    }
}
```

1 Reference variable of type Employee, object of type Employee

2 Reference variable of type Employee, object of type Programmer

3 Accesses variable name defined in class Employee

4 Variables are bound at compile time. Because type of variable programmer is Employee, this accesses variable name defined in class Employee.

5 Accesses method in class Employee

6 Methods are bound at runtime; which method executes depends on the type of object on which it's called. This code calls method printName in class Programmer.

The output of the preceding code is as follows:

```
Employee
Employee
Employee
Programmer
```

Let's see what's happening in the code, step by step:

- ❶ creates an object of class `Employee`, referenced by a variable of its own type—`Employee`.
- ❷ creates an object of class `Programmer`, referenced by a variable of its base type—`Employee`.
- ❸ accesses variable `name` defined in class `Employee` and prints `Employee`.
- ❹ also prints `Employee`. The type of the variable `programmer` is `Employee`. Because the variables are bound at compile time, the type of the object that's referenced by the variable `emp` doesn't make a difference. `programmer.name` will access the variable `name` defined in the class `Employee`.
- ❺ prints `Employee`. Because the type of the reference variable `emp` and the type of object referenced by it are the same (`Employee`), there's no confusion with the method call.
- ❻ prints `Programmer`. Even though the method `printName` is called using a reference of type `Employee`, the JRE is aware that the method is invoked on a `Programmer` object and hence executes the overridden `printName` method in the class `Programmer`.

EXAM TIP Watch out for code in the exam that uses variables of the base class to refer to objects of the derived class and then accesses variables and methods of the referenced object. Remember that variables bind at compile time, whereas methods bind at runtime.

6.6.3 Polymorphism with interfaces

Polymorphism can also be implemented using interfaces. Whereas polymorphism with classes has a class as the base class, polymorphism with interfaces requires a class to implement an interface. Polymorphism with interfaces *always* involves `abstract` methods from the implemented interface because interfaces can define only `abstract` methods.

Let's start with an example. Here's an interface, `MobileAppExpert`, which defines a method, `deliverMobileApp`:

```
interface MobileAppExpert {
    void deliverMobileApp();
}
```

As you know, all the methods defined in an interface are implicitly `abstract` and `public`. Here are the classes `Programmer` and `Manager` that implement this interface and the method `deliverMobileApp`:

```
class Employee {}
class Programmer extends Employee implements MobileAppExpert {
    public void deliverMobileApp() {
        System.out.println("testing complete on real device");
    }
}
```

```
class Manager extends Employee implements MobileAppExpert {
    public void deliverMobileApp() {
        System.out.println("QA complete");
        System.out.println("code delivered with release notes");
    }
}
```

NOTE I've deliberately removed the methods previously defined in `Employee`, `Programmer`, and `Manager` because they are not relevant in this section.

The relationships among the two classes and the interface are shown in figure 6.23.

In the real world, the delivery of a mobile application would have different meaning for a programmer and a manager. For a *programmer*, the delivery of a mobile application may require the completion of testing on the real mobile device. However, for a *manager*, the delivery of a mobile application may mean completing the QA process and handing over code to the client along with any release notes. The bottom line is that the same message, `deliverMobileApp`, results in the execution of different sets of steps for a programmer and a manager.

Here's a class `PolymorphismWithInterfaces` that creates objects of the classes `Programmer` and `Manager` and calls the method `deliverMobileApp`:

```
class PolymorphismWithInterfaces {
    public static void main(String[] args)  {
        MobileAppExpert expert1 = new Programmer();
        MobileAppExpert expert2 = new Manager();

        expert1.deliverMobileApp();
        expert2.deliverMobileApp();
    }
}
```

❶ **Reference type of variables expert1 and expert2 is MobileAppExpert**

The output of the preceding code is as follows:

```
testing complete on real device
QA complete
code delivered with release notes
```

At ❶, the type of the variable is `MobileAppExpert`. Because the classes `Manager` and `Programmer` implement the interface `MobileAppExpert`, a reference variable of type

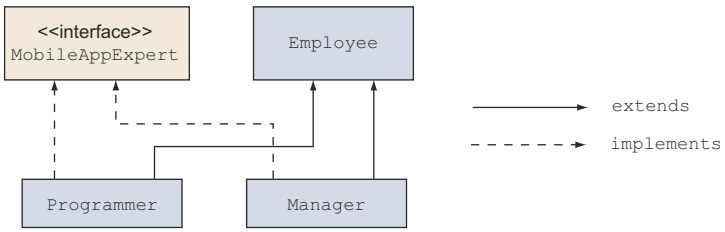


Figure 6.23 Relationships among classes `Employee`, `Programmer`, and `Manager` and the interface `MobileAppExpert`

MobileAppExpert can also be used to store objects of the classes Programmer and Manager.

Because both these classes also extend the class Employee, you can use a variable of type Employee to store objects of the classes Programmer and Manager. But in this case you won't be able to call method deliverMobileApp because it isn't visible to the class Employee. Examine the following code:

```
class PolymorphismWithInterfaces {
    public static void main(String[] args)  {
        Employee expert1 = new Programmer();
        Employee expert2 = new Manager();

        expert1.deliverMobileApp();
        expert2.deliverMobileApp();
    }
}
```

Employee can't see deliverMobileApp

Won't compile

Let's see what happens if we modify the class Employee to implement the interface MobileAppExpert, as follows:

```
class Employee implements MobileAppExpert {
    // code
}
interface MobileAppExpert {
    // code
}
```

Now the classes Programmer and Manager can just extend the class Employee. They no longer need to implement the interface MobileAppExpert because their base class, Employee, implements it:

```
class Programmer extends Employee {
    // code
}
class Manager extends Employee {
    // code
}
```

With the modified code, the new relationships among the classes Employee, Manager, and Programmer and the interface MobileAppExpert are shown in figure 6.24.

Let's try to access the method deliverMobileApp using a reference variable of type Employee class, as follows:

```
class PolymorphismWithInterfaces {
    public static void main(String[] args)  {
        Employee expert1 = new Programmer();
        Employee expert2 = new Manager();

        expert1.deliverMobileApp();
        expert2.deliverMobileApp();
    }
}
```

Employee can see deliverMobileApp

Will work!

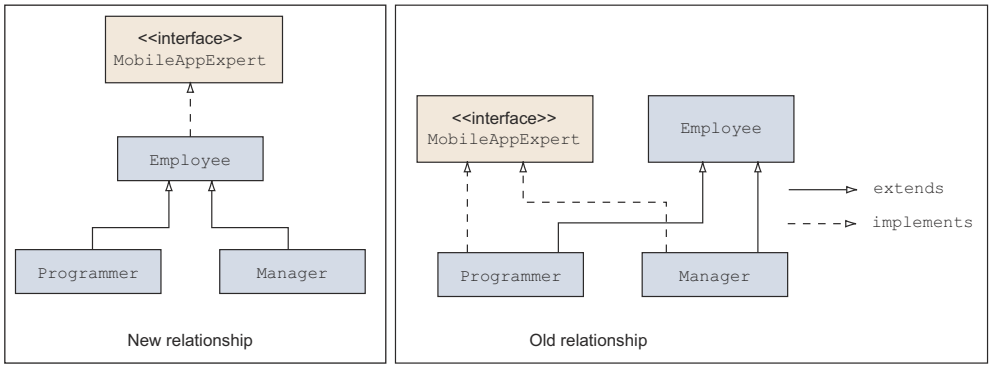


Figure 6.24 Modified relationships among the classes `Employee`, `Manager`, and `Programmer` and the interface `MobileAppExpert`

Figure 6.25 shows what's accessible to the variable `expert1`.

EXAM TIP Watch out for overloaded methods that seem to participate in polymorphism—overloaded methods don't participate in polymorphism. Only overridden methods—methods with the same method signatures—participate in polymorphism.

6.7 Summary

We started the chapter with a discussion of inheritance and polymorphism, using an example from everyday life: all creatures inherit properties and behavior of their parents, and the same action (such as *reproduce*) may have different meanings for different species. Inheritance enables the reuse of existing code, and it can be implemented using classes and interfaces. A class can't extend more than one class, but it can implement more than one interface. Inheriting a class is also called *subclassing*, and the inherited class is referred to as the *base* or *parent class*. A class that inherits another class or implements an interface is called a *derived class* or *subclass*.

Just as it's common to address someone using a last name or family name, an object of a derived class can be referred to with a variable of a base class or an interface that it implements. But when you refer to an object using a variable of the base class, the variable can access only the members defined in the base class. Similarly, a variable of type interface can access only the members defined in that interface. Even

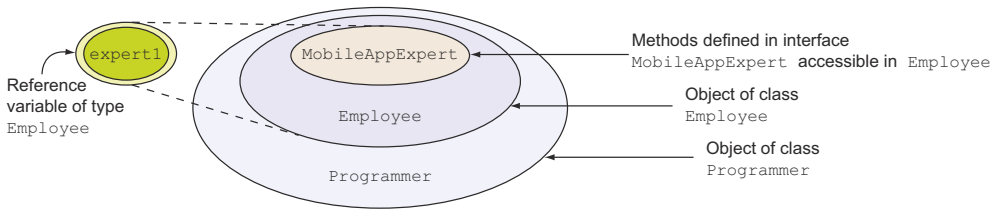


Figure 6.25 What's accessible to the variable `expert1`

with this limitation, you may wish to refer to objects using variables of their base class to work with multiple objects that have common base classes.

Objects of related classes—the ones that share an inheritance relationship—can be cast to another object. You may wish to cast an object when you wish to access its members that are not available by default using the variable that's used to refer to the object.

The keywords `this` and `super` are object references and are used to access an object and its base class respectively. You can use the keyword `this` to access a class's variables, methods, and constructors. Similarly, the keyword `super` is used to access a base class's variables, methods, and constructors.

Polymorphism is the ability of objects to execute methods defined in a superclass or base class, depending upon their type. Classes that share an inheritance relationship exhibit polymorphism. The polymorphic method should be defined in both the base class and the inherited class.

You can implement polymorphism by using either classes or interfaces. In the case of polymorphism with classes, the base class can be either an abstract class or a concrete class. The method in question here also need not be an abstract method. When you implement polymorphism using interfaces, you must use an `abstract` method from the interface.

6.8 Review notes

Inheritance with interfaces and classes:

- A class can inherit the properties and behavior of another class.
- A class can implement multiple interfaces.
- An interface can inherit zero or more interfaces. An interface cannot inherit a class.
- Inheritance enables you to use existing code.
- Inheriting a class is also known as subclassing.
- A class that inherits another class is called a derived class or subclass.
- A class that is inherited is called a parent or base class.
- Private members of a base class cannot be inherited in the derived class.
- A derived class can only inherit members with the default access modifier if both the base class and the derived class are in the same package.
- A class uses the keyword `extends` to inherit a class.
- An interface uses the keyword `extends` to inherit another interface.
- A class uses the keyword `implements` to implement an interface.
- A class can implement multiple interfaces but can inherit only one class.
- An interface can extend multiple interfaces.
- The method signatures of a method defined in an interface and in the class that implements the interface must match; otherwise, the class won't compile.
- An `abstract` class can inherit a concrete class, and a concrete class can inherit an `abstract` class.

Reference variable and object types:

- With inheritance, you can also refer to an object of a derived class using a variable of a base class or interface.
- An object of a base class can't be referred to using a reference variable of its derived class.
- When an object is referred to by a reference variable of a base class, the reference variable can only access the variables and members that are defined in the base class.
- When an object is referred to by a reference variable of an interface implemented by a class, the reference variable can access only the variables and methods defined in the interface.
- You may need to access an object of a derived class using a reference variable of the base class to group and use all the classes with common parent classes or interfaces.

The need for casting:

- Casting is the process of forcefully making a variable behave as a variable of another type.
- If the class `Manager` extends the class `Employee`, and a reference variable `emp` of type `Employee` is used to refer to an object of the class `Manager`, `((Manager)emp)` will cast the variable `emp` to `Manager`.

Using `super` and `this` to access objects and constructors:

- Keywords `super` and `this` are object references. These variables are defined and initialized by the JVM for every object in its memory.
- The `this` reference always points to an object's *own instance*.
- You can use the keyword `this` to refer to all methods and variables that are accessible to a class.
- If a method defines a local variable or method parameter with the same name as an instance variable, the keyword `this` must be used to access the instance variable in the method.
- You can call one constructor from another constructor by using the keyword `this`.
- `super`, an object reference, refers to the parent class or the base class of a class.
- The reference variable `super` can be used to access a variable or method from the base class if there is a clash of these names. This situation normally occurs when a derived class defines variables and methods with the same names as in the base class.
- The reference variable `super` can also be used to refer to the constructors of the base class in a derived class.

Polymorphism with classes:

- The literal meaning of the word "polymorphism" is "many forms."
- In Java, polymorphism comes into the picture when there's an inheritance relationship between classes, and both the base and derived classes define methods with the same name.
- The polymorphic methods are also called *overridden methods*.
- Overridden methods should define methods with the same name, same argument list, same list of method parameters. The return type of the overriding method can be the same, or a subclass of the return type of the overridden method in the base class, which is also known as covariant return type.
- Access modifiers for an overriding method can be the same or less restrictive but can't be more restrictive than the method being overridden.
- A derived class is said to override a method in the base class if it defines a method with the same name, same parameter list, and same return type as in the derived class.
- If a method defined in a base class is overloaded in the derived classes, then these two methods (in the base class and the derived class) are not called polymorphic methods.
- When implementing polymorphism with classes, a method defined in the base class may or may not be `abstract`.
- When implementing polymorphism with interfaces, a method defined in the base interface is always `abstract`.

6.9 Sample exam questions

Q6-1. What is the output of the following code?

```
class Animal {
    void jump() { System.out.println("Animal"); }
}
class Cat extends Animal {
    void jump(int a) { System.out.println("Cat"); }
}
class Rabbit extends Animal {
    void jump() { System.out.println("Rabbit"); }
}
class Circus {
    public static void main(String args[]) {
        Animal cat = new Cat();
        Rabbit rabbit = new Rabbit();
        cat.jump();
        rabbit.jump();
    }
}
```

a
```
Animal
Rabbit
```

b
```
Cat
Rabbit
```

c
```
Animal
Animal
```

d None of the above

Q6-2. Given the following code, select the correct statements:

```
class Flower {
    public void fragrance() {System.out.println("Flower"); }
}
class Rose {
    public void fragrance() {System.out.println("Rose"); }
}
class Lily {
    public void fragrance() {System.out.println("Lily"); }
}
class Bouquet {
    public void arrangeFlowers() {
        Flower f1 = new Rose();
        Flower f2 = new Lily();
        f1.fragrance();
    }
}
```

a The output of the code is:

```
Flower
```

b The output of the code is:

```
Rose
```

c The output of the code is:

```
Lily
```

d The code fails to compile.

Q6-3. Examine the following code and select the correct method declaration to be inserted at `//INSERT CODE HERE`:

```
interface Movable {
    void move();
}
class Person implements Movable {
    public void move() { System.out.println("Person move"); }
}
class Vehicle implements Movable {
    public void move() { System.out.println("Vehicle move"); }
}
class Test {
    // INSERT CODE HERE
        movable.move();
    }
}
```

a `void walk(Movable movable) {`
b `void walk(Person movable) {`
c `void walk(Vehicle movable) {`
d `void walk() {`

Q6-4. Select the correct statements:

a Only an `abstract` class can be used as a base class to implement polymorphism with classes.
b Polymorphic methods are also called overridden methods.
c In polymorphism, depending on the exact type of object, the JVM executes the appropriate method at compile time.
d None of the above.

Q6-5. Given the following code, select the correct statements:

```
class Person {}
class Employee extends Person {}
class Doctor extends Person {}
```

a The code exhibits polymorphism with classes.
b The code exhibits polymorphism with interfaces.
c The code exhibits polymorphism with classes and interfaces.
d None of the above.

Q6-6. Which of the following statements are true?

a Inheritance enables you to reuse existing code.
b Inheritance saves you from having to modify common code in multiple classes.
c Polymorphism passes special instructions to the compiler so that the code can run on multiple platforms.
d Polymorphic methods cannot throw exceptions.

Q6-7. Given the following code, which of the options are true?

```
class Satellite {
    void orbit() {}
}
class Moon extends Satellite {
    void orbit() {}
}
class ArtificialSatellite extends Satellite {
    void orbit() {}
}
```

a The method `orbit` defined in the classes `Satellite`, `Moon`, and `ArtificialSatellite` is polymorphic.
b Only the method `orbit` defined in the classes `Satellite` and `ArtificialSatellite` is polymorphic.

c Only the method `orbit` defined in the class `ArtificialSatellite` is polymorphic.

d None of the above.

Q6-8. Examine the following code:

```
class Programmer {
    void print() {
        System.out.println("Programmer - Mala Gupta");
    }
}
class Author extends Programmer {
    void print() {
        System.out.println("Author - Mala Gupta");
    }
}
class TestEJava {
    Programmer a = new Programmer();
    // INSERT CODE HERE
    a.print();
    b.print();
}
```

Which of the following lines of code can be individually inserted at `//INSERT CODE HERE` so that the output of the code is as follows:

```
Programmer - Mala Gupta
Author - Mala Gupta
```

a `Programmer b = new Programmer();`

b `Programmer b = new Author();`

c `Author b = new Author();`

d `Author b = new Programmer();`

e `Programmer b = ((Author)new Programmer());`

f `Author b = ((Author)new Programmer());`

Q6-9. Given the following code, which of the options, when applied individually, will make it compile successfully?

```
Line1>    interface Employee {}

Line2>    interface Printable extends Employee {
Line3>        String print();
Line4>    }

Line5>    class Programmer {
Line6>        String print() { return("Programmer - Mala Gupta"); }
Line7>    }

Line8>    class Author extends Programmer implements Printable, Employee {
Line9>        String print() { return("Author - Mala Gupta"); }
Line10>   }
```

a Modify code on line 2 to: `interface Printable{`

b Modify code on line 3 to: `publicStringprint();`

c Define the accessibility of the `print` methods to `public` on lines 6 and 9.

d Modify code on line 8 so that it implements only the interface `Printable`.

Q6-10. What is the output of the following code?

```
class Base {
    String var = "EJava";
    void printVar() {
        System.out.println(var);
    }
}
class Derived extends Base {
    String var = "Guru";
    void printVar() {
        System.out.println(var);
    }
}
class QReference {
    public static void main(String[] args) {
        Base base = new Base();
        Base derived = new Derived();

        System.out.println(base.var);
        System.out.println(derived.var);
        base.printVar();
        derived.printVar();
    }
}
```

a
```
EJava
EJava
EJava
Guru
```

b
```
EJava
Guru
EJava
Guru
```

c
```
EJava
EJava
EJava
EJava
```

d
```
EJava
Guru
Guru
Guru
```

6.10 Answers to sample exam questions

Q6-1. What is the output of the following code?

```
class Animal {
    void jump() { System.out.println("Animal"); }
}
class Cat extends Animal {
    void jump(int a) { System.out.println("Cat"); }
```

```
}
class Rabbit extends Animal {
    void jump() { System.out.println("Rabbit"); }
}
class Circus {
    public static void main(String args[]) {
        Animal cat = new Cat();
        Rabbit rabbit = new Rabbit();
        cat.jump();
        rabbit.jump();
    }
}
```

a **Animal**
Rabbit

b Cat
Rabbit

c Animal
Animal

d None of the above

Answer: a

Explanation: Although the classes `Cat` and `Rabbit` seem to override the method `jump`, the class `Cat` doesn't override the method `jump()` defined in the class `Animal`. The class `Cat` defines a method parameter with the method `jump`, which makes it an overloaded method, not an overridden method. Because the class `Cat` extends the class `Animal`, it has access to the following two overloaded `jump` methods:

```
void jump() { System.out.println("Animal"); }
void jump(int a) { System.out.println("Cat"); }
```

The following line of code creates an object of class `Cat` and assigns it to a variable of type `Animal`:

```
Animal cat = new Cat();
```

When you call the method `jump` on the previous object, it executes the method `jump`, which doesn't accept any method parameters, printing the following value:

```
Animal
```

The following code will also print `Animal` and not `Cat`:

```
Cat cat = new Cat();
cat.jump();
```

Q6-2. Given the following code, select the correct statements:

```
class Flower {
    public void fragrance() {System.out.println("Flower"); }
}
class Rose {
    public void fragrance() {System.out.println("Rose"); }
}
```

```
class Lily {
    public void fragrance() {System.out.println("Lily"); }
}
class Bouquet {
    public void arrangeFlowers() {
        Flower f1 = new Rose();
        Flower f2 = new Lily();
        f1.fragrance();
    }
}
```

a The output of the code is:

```
Flower
```

b The output of the code is:

```
Rose
```

c The output of the code is:

```
Lily
```

d **The code fails to compile.**

Answer: d

Explanation: Although the code seems to implement polymorphism using classes, note that neither of the classes `Rose` or `Lily` *extends* the class `Flower`. Hence, a variable of type `Flower` can't be used to store objects of the classes `Rose` or `Lily`. The following lines of code will fail to compile:

```
Flower f1 = new Rose();
Flower f2 = new Lily();
```

Q6-3. Examine the following code and select the correct method declaration to be inserted at `//INSERT CODE HERE`:

```
interface Movable {
    void move();
}
class Person implements Movable {
    public void move() { System.out.println("Person move"); }
}
class Vehicle implements Movable {
    public void move() { System.out.println("Vehicle move"); }
}
class Test {
    // INSERT CODE HERE
        movable.move();
    }
}
```

a **`void walk(Movable movable) {`**

b **`void walk(Person movable) {`**

c **`void walk(Vehicle movable) {`**

d `void walk() {`

Answer: a, b, c

Explanation: You need to insert code in the class `Test` that makes the following line of code work:

```
movable.move();
```

Hence, option (d) is incorrect. Because class `Test` doesn't define any instance methods, the only way that the question's line of code can execute is when a method parameter `movable` is passed to the method `walk`.

Option (a) is correct. Because the interface `Movable` defines the method `move`, you can pass a variable of its type to the method `move`.

Option (b) is correct. Because the class `Person` implements the interface `Movable` and defines the method `move`, you can pass a variable of its type to the method `walk`. With this version of the method `walk`, you can pass it an object of the class `Person` or any of its subclasses.

Option (c) is correct. Because the class `Vehicle` implements the interface `Movable` and defines the method `move`, you can pass a variable of its type to the method `walk`. With this version of method `walk`, you can pass it an object of the class `Vehicle` or any of its subclasses.

Q6-4. Select the correct statements:

- **a** Only an `abstract` class can be used as a base class to implement polymorphism with classes.
- **b** **Polymorphic methods are also called overridden methods.**
- **c** In polymorphism, depending on the exact type of object, the JVM executes the appropriate method at compile time.
- **d** None of the above.

Answer: b

Option (a) is incorrect. To implement polymorphism with classes, either an `abstract` class or a concrete class can be used as a base class.

Option (c) is incorrect. First of all, no code execution takes place at compile time. Code can only execute at runtime. In polymorphism, the determination of the exact method to execute is deferred until runtime and is determined by the exact type of the object on which a method needs to be called.

Q6-5. Given the following code, select the correct statements:

```
class Person {}
class Employee extends Person {}
class Doctor extends Person {}
```

- **a** The code exhibits polymorphism with classes.
- **b** The code exhibits polymorphism with interfaces.
- **c** The code exhibits polymorphism with classes and interfaces.
- **d** **None of the above.**

Answer: d

Explanation: The given code does not define any method in the class `Person` that is redefined or implemented in the classes `Employee` and `Doctor`. Though the classes `Employee` and `Doctor` extend the class `Person`, all these three polymorphism concepts or design principles are based on a method, which is missing in these classes.

Q6-6. Which of the following statements are true?

a **Inheritance enables you to reuse existing code.**
b **Inheritance saves you from having to modify common code in multiple classes.**
c Polymorphism passes special instructions to the compiler so that the code can run on multiple platforms.
d Polymorphic methods cannot throw exceptions.

Answer: a, b

Explanation: Option (a) is correct. Inheritance can allow you to reuse existing code by extending a class. In this way, the functionality that is already defined in the base class need not be defined in the derived class. The functionality offered by the base class can be accessed in the derived class as if it were defined in the derived class.

Option (b) is correct. Common code can be placed in the base class, which can be extended by all the derived classes. If any changes need to be made to this common code, it can be modified in the base class. The modified code will be accessible to all the derived classes.

Option (c) is incorrect. Polymorphism doesn't pass any special instructions to the compiler to make the Java code execute on multiple platforms. Java code can execute on multiple platforms because the Java compiler compiles to virtual machine code, which is platform-neutral. Different platforms implement this virtual machine.

Option (d) is incorrect. Polymorphic methods can throw exceptions.

Q6-7. Given the following code, which of the options are true?

```
class Satellite {
    void orbit() {}
}
class Moon extends Satellite {
    void orbit() {}
}
class ArtificialSatellite extends Satellite {
    void orbit() {}
}
```

a **The method `orbit` defined in the classes `Satellite`, `Moon`, and `ArtificialSatellite` is polymorphic.**
b Only the method `orbit` defined in the classes `Satellite` and `ArtificialSatellite` is polymorphic.
c Only the method `orbit` defined in the class `ArtificialSatellite` is polymorphic.
d None of the above.

Answer: a

Explanation: All of these options define classes. When methods with the same method signature are defined in classes that share an inheritance relationship, the methods are considered polymorphic.

Q6-8. Examine the following code:

```
class Programmer {
    void print() {
        System.out.println("Programmer - Mala Gupta");
    }
}
class Author extends Programmer {
    void print() {
        System.out.println("Author - Mala Gupta");
    }
}
class TestEJava {
    Programmer a = new Programmer();
    // INSERT CODE HERE
    a.print();
    b.print();
}
```

Which of the following lines of code can be individually inserted at `//INSERT CODE HERE` so that the output of the code is as follows:

```
Programmer - Mala Gupta
Author - Mala Gupta
```

a. `Programmer b = new Programmer();`
b. **`Programmer b = new Author();`**
c. **`Author b = new Author();`**
d. `Author b = new Programmer();`
e. `Programmer b = ((Author)new Programmer());`
f. `Author b = ((Author)new Programmer());`

Answer: b, c

Explanation: Option (a) is incorrect. This code will compile, but because both the reference variable and object are of type `Programmer`, calling `print` on this object will print `Programmer - Mala Gupta`, not `Author - Mala Gupta`.

Option (d) is incorrect. This code will not compile. You can't assign an object of a base class to a reference variable of a derived class.

Option (e) is incorrect. This line of code will compile successfully, but it will fail at runtime with a `ClassCastException`. An object of a base class can't be cast to an object of its derived class.

Option (f) is incorrect. The expression `((Author)new Programmer())` is evaluated before it can be assigned to a reference variable of type `Author`. This line of code also tries to cast an object of the base class—`Programmer`—to an object of its derived

class—Author. This code will also compile successfully but will fail at runtime with a ClassCastException. Using a reference variable of type Author won't make a difference here.

Q6-9. Given the following code, which of the options, when applied individually, will make it compile successfully?

```
Line1>    interface Employee {}

Line2>    interface Printable extends Employee {
Line3>        String print();
Line4>    }

Line5>    class Programmer {
Line6>        String print() { return("Programmer - Mala Gupta"); }
Line7>    }

Line8>     class Author extends Programmer implements Printable, Employee {
Line9>        String print() { return("Author - Mala Gupta"); }
Line10>    }
```

a Modify code on line 2 to: `interface Printable {`

b Modify code on line 3 to: public String print();

c **Define the accessibility of the `print` methods to `public` on lines 6 and 9.**

d Modify code on line 8 so that it implements only the interface Printable.

Answer: c

Explanation: The methods in an interface are implicitly public. A non-abstract class that implements an interface must implement all the methods defined in the interface. While overriding or implementing the methods, the accessibility of the implemented method must be public. An overriding method can't be assigned a weaker access privilege than public.

Option (a) is incorrect. There are no issues with the interface Printable extending the interface Employee and the class Author implementing both of these interfaces.

Option (b) is incorrect. Adding the access modifier to the method print on line 3 will not make any difference to the existing code. The methods defined in an interface are implicitly public.

Option (d) is incorrect. There are no issues with a class implementing two interfaces when one of the interfaces extends the other interface.

Q6-10. What is the output of the following code?

```
class Base {
    String var = "EJava";
    void printVar() {
        System.out.println(var);
    }
}
class Derived extends Base {
    String var = "Guru";
    void printVar() {
```

```
        System.out.println(var);
    }
}
class QReference {
    public static void main(String[] args) {
        Base base = new Base();
        Base derived = new Derived();

        System.out.println(base.var);
        System.out.println(derived.var);
        base.printVar();
        derived.printVar();
    }
}
```

a
```
EJava
EJava
EJava
Guru
```

b
```
EJava
Guru
EJava
Guru
```

c
```
EJava
EJava
EJava
EJava
```

d
```
EJava
Guru
Guru
Guru
```

Answer: a

Explanation: With inheritance, the instance variables bind at compile time and the methods bind at runtime. The following line of code refers to an object of the class `Base`, using a reference variable of type `Base`. Hence, both of the following lines of code print `EJava`:

```
System.out.println(base.var);
base.printVar();
```

But the following line of code refers to an object of the class `Derived` using a reference variable of type `Base`:

```
Base derived = new Derived();
```

Because the instance variables bind at compile time, the following line of code accesses and prints the value of the instance variable defined in the class `Base`:

```
System.out.println(derived.var);    // prints EJava
```

In `derived.printVar()`, even though the method `printVar` is called using a reference of type `Base`, the JVM is aware that the method is invoked on a `Derived` object and so executes the overridden `printVar` method in the class `Derived`.