Chapter 9

Inheritance and Polymorphism

Inheritance is a mechanism that allows one class to incorporate the state and behavior of another class. This makes it possible to build new classes out of existing classes thereby promoting code sharing and reusability. This chapter describes how inheritance is achieved in object-oriented programming. It also discusses an important related concept known as polymorphism which allows classes that inherit from a common class to be treated in a uniform manner.

9.1 Generalization Relationships and Inheritance

The `Account` class implements the concept of an account in a bank. In the real world, banks offer various types of accounts to customers. For example, customers can open accounts such as the following:

- Savings account: This type of account is generally intended for saving funds; interest is usually paid to the account at specified intervals (e.g., monthly).

- Chequing account: This type of account is generally intended for day-to-day transactions. The customer may be allowed to write a certain amount of free cheques in a certain period; any amount of cheques beyond this amount is charged a fee. No interest is payable on this type of account.

Each type of account has an account number and a balance. Each type of account also enables deposits and withdrawals. Thus, the concept of a savings account is really an extension of the concept of an account. Similarly, the concept of a chequing account is an extension of the concept of an account. Inheritance makes it possible to create a `SavingsAccount` class and a `ChequingAccount` class by extending the `Account` class. Before explaining how this is achieved, some terminology is presented.

Generalization is used to model a relationship between classes in which one class represents a more general concept (e.g., `Account`) and another class represents a more specialized concept (e.g., `SavingsAccount`). The more general class is called the *superclass* or *parent class* and the more *specialized* class is called the *subclass* or *child class*. Each subclass is said to inherit the features (attributes and operations) of its superclass. Generalization is sometimes called the "is-a" relationship because an instance of a subclass is an instance of the superclass as well. For example, an instance of `SavingsAccount` "is-a" instance of `Account`.

Generalization and specialization are two different viewpoints of the same relationship. Generalization refers to the fact that the superclass generalizes the subclasses while specialization refers to the fact that subclasses specialize the superclass. *Inheritance* refers to the mechanism of sharing attributes and operations (state and behavior) using the generalization relationship.

A subclass extends the capabilities of the superclass with additional methods and attributes. For example, consider the generalization relationships between `Account`, `SavingsAccount`, and `ChequingAccount`, shown in the UML diagram in Figure 9.1 below.

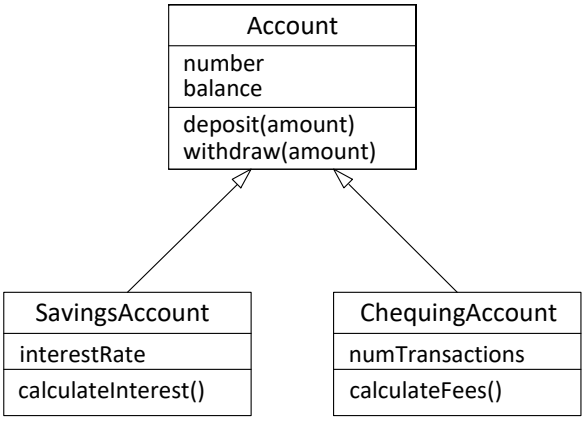


Figure 9.1: Generalization Relationships between `Account`, `SavingsAccount`, and `ChequingAccount`

`SavingsAccount` adds one attribute, `interestRate`, and one method, `calculateInterest()`, to the `Account` class. `ChequingAccount` also adds one attribute, `numTransactions`, and one method, `calculateFees()`, to the `Account` class. Using the terminology, `Account` is a generalization of `SavingsAccount` and `ChequingAccount`. Conversely, `SavingsAccount` is a

specialization of `Account`; `ChequingAccount` is also a specialization of `Account`.

It is possible to specialize the subclasses in Figure 9.1 even further. For example, suppose that there are three types of savings accounts in a bank. We can create three new classes, `A`, `B`, and `C`, which are subclasses of `SavingsAccount`. Suppose further that there are two types of chequing accounts in a bank. We can create two new classes, `D`, and `E`, which are subclasses of `ChequingAccount`. A UML diagram with the new classes is shown in Figure 9.2.

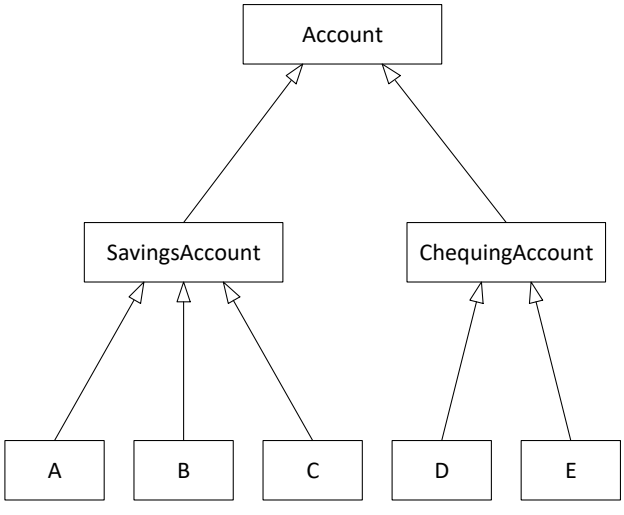


Figure 9.2: Specializing `SavingsAccount` and `ChequingAccount`

The UML diagrams in Figure 9.1 and Figure 9.2 are examples of an inheritance hierarchy. An *inheritance hierarchy* consists of a set of classes which are connected by generalization relationships. The *depth* of an inheritance hierarchy is the length of the path from the root to the deepest class in the hierarchy; it varies from application to application. The inheritance hierarchy in Figure 9.1 has a depth of one. The inheritance hierarchy in Figure 9.2 has a depth of two.

Since `SavingsAccount` is a child class of `Account`, it is said to be a *direct subclass* of `Account`. Now, class `A` is a direct subclass of `SavingsAccount` since it is a child class of `SavingsAccount`. However, `A` has all the features of `Account` since `SavingsAccount` inherits all the features of `Account`. Thus, `A` is also a subclass of `Account`, even though there is an intervening class between `A` and `Account` in the inheritance hierarchy. In this situation, `A` is said to be an *indirect subclass* of `Account`. Note that since an inheritance hierarchy

can be as deep as required, there may be several intervening classes between `Account` and one of its indirect subclasses.

9.2 Creating and Manipulating Child Classes

This section explains how to write the code for a child class. In particular, it shows how to write the code for the two child classes, `SavingsAccount` and `ChequingAccount`, shown in Figure 9.1. It also explains how instances of a child class can be manipulated by clients.

9.2.1 Writing a Child Class

To create a subclass of `Account`, the `extends` keyword is used to indicate that a subclass/superclass relationship is being established. The attributes and methods that belong to the subclass are written the same way as in any other class. The `SavingsAccount` subclass is written as follows:

```
public class SavingsAccount extends Account {

   private double interestRate;      // new attribute of subclass

   public SavingsAccount(int number, double balance,
      double interestRate) {

      // code for constructor
   }

   public void calculateInterest() {
      // code for calculateInterest()
   }
}
```

Similarly, the `ChequingAccount` class is written as follows:

```
public class ChequingAccount extends Account {

   private int numTransactions;

   public ChequingAccount(int number, double balance) {
      // code for constructor
   }

   public void calculateFees() {
      // code for calculateFees()
```

```
    }
}
```

Note that the instance variables of `Account` are not re-declared in `SavingsAccount` or `ChequingAccount` since a subclass inherits all the instance variables of its superclass. Also, the methods of `Account` are not rewritten since each subclass inherits the methods of its superclass as well. This is what inheritance is all about. A subclass automatically acquires the attributes and methods of its superclass.

9.2.2 Inheriting Attributes and Methods from Parent Class

Consider the `calculateInterest()` method of the `SavingsAccount` class. This method should calculate the monthly interest based on the current balance in the account. Since `balance` is an attribute of `Account` which is inherited by `SavingsAccount`, the following seems to be a reasonable implementation of the `calculateInterest()` method:

```
public void calculateInterest() {
   double interest;

   interest = balance * (interestRate / 12.0);
   deposit(interest); // method inherited from Account superclass
}
```

However, the above code will not compile. Although `balance` is an attribute of `SavingsAccount` by inheritance, it has been declared as `private` in its parent class, `Account`. So, the `balance` attribute is only visible within the `Account` class. To make the `balance` attribute visible to subclasses, it should be declared as `protected` in `Account`. Of course, declaring the `balance` instance variable as `public` makes it accessible to any other class, including subclasses of `Account`.

Note that even though the `SavingsAccount` class does not have a `deposit()` method, the following statement is valid since the `deposit()` method is inherited from `Account`:

```
deposit(interest);
```

Inheritance of methods provides another way to fix the `calculateInterest()` method instead of changing the access modifier for `balance` in `Account`. In this case, the accessor for the `balance` instance

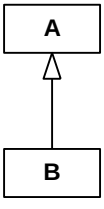
variable in `Account` is called from the `calculateInterest()` method as follows:

```
public void calculateInterest() {
   double interest;

   interest = getBalance() * (interestRate / 12.0);
   deposit(interest);
}
```

The `getBalance()` accessor from `Account` is inherited by `SavingsAccount` so it can be used by the `calculateInterest()` method, just like any other method in the `SavingsAccount` class.

9.2.3 Initialising Instance Variables



If `B` is a subclass of `A`, the constructor of `A` can be called in the constructor of `B` to initialise the instance variables declared in `A`. This is useful when the instance variables of `A` are declared as `private` since, as mentioned before, they cannot be directly accessed in `B`. The superclass constructor can be called using the special method `super()`, followed by the list of arguments required by the superclass constructor.

For example, the constructor of the `ChequingAccount` class can be written as follows:

```
public ChequingAccount(int number, double balance) {
   super(number, balance);
                     // call superclass constructor

   numTransactions = 0;
                     // set attributes declared in ChequingAccount
}
```

The call to `super()` initializes the `number` and `balance` attributes using the superclass constructor. Note that the call to `super()` must be the first statement in the constructor of `ChequingAccount`.

Suppose that the call to `super(number, balance)` is omitted from the `ChequingAccount` constructor. The compiler attempts to insert code which invokes the default no-argument constructor in the superclass, `Account`. It will generate an error if the superclass does not contain a no-argument constructor

(either explicitly written in the superclass or provided as a default constructor by the compiler).

If there is no constructor in the subclass, the compiler attempts to provide the default no-argument constructor. This no-argument constructor in the subclass needs to invoke the no-argument constructor in the superclass to initialize the instance variables of the superclass. Again, the compiler will generate an error if the superclass does not contain a no-argument constructor.

9.2.4 Manipulating Child Classes

A child class can be manipulated like any other class. A client object can create an instance of the child class using the `new` keyword and then proceed to invoke methods on the child class using the object variable. For example, consider a client object performing the following operations on the `ChequingAccount` child class:

Line 1:
```
ChequingAccount ca = new ChequingAccount(10, 2000.00);
```

Line 2:
```
ca.withdraw(500.00);
```

Line 3:
```
System.out.println(ca.getBalance());
```

Line 4:
```
ca.calculateFees();
```

Line 1 creates an instance of `ChequingAccount` and assigns it to a variable of type `ChequingAccount`. Line 2 invokes the `withdraw()` method on the instance of `ChequingAccount`. Even though the `ChequingAccount` class does not have a `withdraw()` method defined, this is valid since the method is inherited from `Account`. Similarly, the `getBalance()` method is inherited from `Account` in Line 3 and can be used as if it were defined in `ChequingAccount`. Since the `calculateFees()` method is defined in `ChequingAccount` it is legal to invoke this method on an instance of `ChequingAccount` so Line 4 is also valid.

It is helpful to visualize the superclass and subclass as a fused structure that contains all the methods and attributes of the superclass as well as all the methods and attributes of the subclass. The subclass contains all the methods and attributes of the fused structure and thus an instance of the subclass contains all the attributes of the fused structure. All the methods of the fused

structure can be invoked on an instance of the subclass. This fused structure is shown in Figure 9.3.

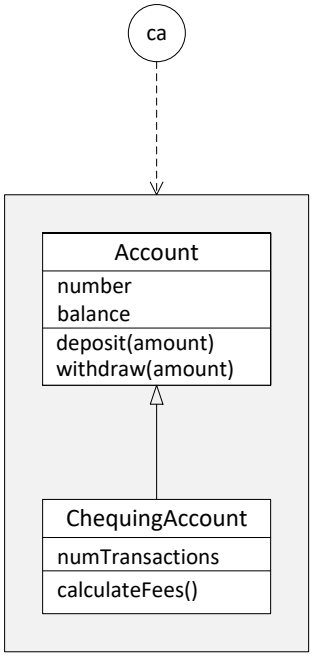
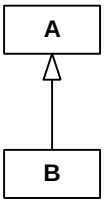


Figure 9.3: Fusing of Superclass and Subclass

9.3 Method Refinement and Replacement

So far we have assumed that the attributes and methods added by a subclass to those inherited from its superclass are always distinct. However, it is possible for a child class to define a method with the same signature (method name and parameter list) as a method in the parent class. The method in the child class is said to *override* the method in the parent class and it effectively hides the method in the parent class.



Suppose that `B` is a subclass of `A`. A method defined in `B` can override a method inherited from `A` in one of two ways: method refinement and method replacement. In *method refinement*, the method inherited from the parent class is executed as part of the behavior of the child class method. Thus, the behavior of the parent class is preserved and augmented with method refinement. In *method replacement*, the code of the parent class method is never

executed when instances of the child class are manipulated. This requires the overridden method to be completely rewritten in `B`.

As an example of method refinement, consider the `toString()` method of the `Account` class:

```
public String toString() {
   String s;
   s = "Account number: " + number + " Balance: " + balance;
   return s;
}
```

A reasonable attempt to refine the `toString()` method in `ChequingAccount` is as follows:

Line 1:
```
public String toString() {
```

Line 2:
```
   String s;
```

Line 3:
```
   s = toString() + " Transaction Count: " + numTransactions;
```

Line 4:
```
   return s;
```

Line 5:
```
}
```

In Line 3, an attempt is made to call the `toString()` method of the parent and augment the string obtained with the `numTransactions` attribute of `ChequingAccount`. However, this results in a recursive call to the `toString()` method being defined. This problem can be solved by preceding the call to the `toString()` method in Line 3 with the keyword `super` as follows:

```
s = super.toString() + " Transaction Count: " + numTransactions;
```

In general, when a superclass method is being refined in a subclass, the keyword `super` should be used to distinguish the method in the superclass from the method being defined in the subclass.

As an example of method replacement, suppose that the `SavingsAccount` class wishes to override the `toString()` method of `Account` using replacement. The `toString()` method must be written without using the services of the `toString()` method from the parent class, `Account`. For example, it can be written as follows:

```
public String toString() {
   String s;

   s =   "Savings Account ->" + " Number: " + getNumber() +
         " Interest rate: " + interestRate;

   return s;
}
```

In the example, the `toString()` method uses the `getNumber()` method to find the value of the `number` attribute of the `SavingsAccount` object since it is a `private` instance variable in `Account`.

Finally, it should be mentioned that if a method is re-defined in a subclass `B` with a different signature from the superclass `A`, this is really *method overloading* which is not related to inheritance and polymorphism.

9.4 The `Object` Class

All Java classes are automatically subclasses of the `Object` class. A class that does not explicitly inherit from another class is a direct subclass of `Object`. For example, although it wasn't mentioned before, the `Account` class is really a direct subclass of the `Object` class. Consequently, every class in Java is either a direct or an indirect subclass of `Object`. Thus, the methods of the `Object` class are available to every class in Java.

Three methods of the `Object` class that are frequently used in this book are listed in Table 9.1.

Method	*Description*
public String toString()	Returns a string representation of the state of the object.
public boolean equals(Object obj)	Returns `true` if the current object is equal to `obj` and `false`, otherwise.
public int hashCode()	Returns a hash code for use in data structures that employ hash tables.

Table 9.1: Commonly Used Methods of the `Object` Class

The methods of the `Object` class are meant to be overridden in user-defined classes. If they are not overridden in a class, they return the values described in Table 9.2.

Method	*Value Returned When Method is not Overridden*
public String toString()	Returns a string with the memory address of the object, prefixed with the name of the class and the '`@`' symbol, e.g., `Account@19821f`.
public boolean equals(Object obj)	Returns `true` if the current object is stored at the same memory address as `obj` and `false`, otherwise. In other words, the `equals()` method in `Object` behaves as "`==`".
public int hashCode()	Generates a hash code which is based on the memory address of the object.

Table 9.2: Not Overriding Methods of `Object` Class

9.5 Substitutability and Polymorphism

This section describes the concept of substitutability which enables an object variable to refer to different types of objects in an inheritance hierarchy. It also discusses an important concept in object-oriented programming known as polymorphism.

9.5.1 The Principle of Substitutability

The Principle of Substitutability says that if we have two classes `A` and `B`, such that `B` is a subclass of `A` (even indirectly), it should be possible to substitute instances of class `B` for instances of class `A` in *any situation* with *no observable effect*.

Subtype refers to a subclass relationship in which the Principle of Substitutability is maintained. This is distinguished from the general *subclass* relationship, which may or may not satisfy this principle.

Consider the following declarations:

```
Account account;
ChequingAccount ca;
```

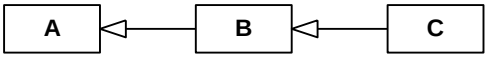
By the Principle of Substitutability, it is possible to create an instance of `ChequingAccount` and assign it to a variable of type `Account`:

```
ca = new ChequingAccount(10, 5000.00);
account = ca;
```

The `ChequingAccount` instance has all the features of an `Account` plus additional features of its own (this is what specialization is all about). It is safe to refer to the `ChequingAccount` instance using a variable of type `Account` since all the `Account` features are present in `ChequingAccount`. However, the converse is not true. It is not possible to create an instance of `Account` and assign it to a variable of type `ChequingAccount`:

```
account = new Account(10, 5000.00);
ca = account;
```

Suppose it is possible to refer to an `Account` instance using a variable of type `ChequingAccount`. An `Account` instance has less features than a `ChequingAccount` instance (it does not have the `numTransactions` attribute neither does it have the `calculateFees()` method). If a `ChequingAccount` feature is requested which is not present in `Account` (e.g., `calculateFees()`), the request would fail. For this reason, substituting an instance of the parent class where an instance of the child class is expected is not allowed.



It should be noted that substitutability works across an inheritance hierarchy. For example, suppose that `B` is a direct subclass of `A` and `C` is a direct subclass of `B`. Since `C` is an indirect subclass of `A`, it is still possible to substitute an instance of `C` whenever an instance of `A` is expected. So, if we have an inheritance hierarchy rooted at `A`, an object variable of type `A` can be used to refer to an instance of any class in the hierarchy.

9.5.2 Polymorphism

Before discussing polymorphism, the terms static type and dynamic type will now be defined. *Static type* is the type assigned to an object variable by means of a declaration statement. Consider the following declarations:

```
Account account;
SavingsAccount sa;
```

Based on the definition, the static type of `account` is `Account` and the static type of `sa` is `SavingsAccount`.

Dynamic type is the actual type associated with an object variable at run-time. Consider the following code which uses the Principle of Substitutability:

```
sa = new SavingsAccount(10, 5000.00, 5.50);
account = sa;
```

The dynamic type of the `account` object variable is `SavingsAccount` since it is actually referring to an instance of `SavingsAccount` at run-time. The `account` variable is said to be *polymorphic* since its dynamic type does not match its static type.

Now, recall that `SavingsAccount` defines a method of its own, `calculateInterest()`, which is declared as follows:

```
public void calculateInterest() {
   :
}
```

It is instructive to consider what happens when the following call is made:

```
account.calculateInterest();
```

This line of code generates a compilation error. Even though the `account` variable is actually referring to a `SavingsAccount` instance at run-time (which has the `calculateInterest()` method defined), the compiler does not know this (because it happens at run-time). Thus, the compiler uses the static type of an object variable to determine if a method call is legal. Since the static type of `account` is `Account` and there is no `calculateInterest()` method in `Account`, the compiler generates an error to the effect that there is no such method in the `Account` class. Thus, it is only possible to invoke methods defined in the class corresponding to the static type of an object variable, even

though there may be additional methods in the class corresponding to the dynamic type.

Figure 9.4 depicts this situation using the fused superclass/subclass diagram introduced earlier. The features of the subclass have been shaded to indicate that, while they are actually present in the object being referred to, they cannot be accessed. Only the features defined in the static type (superclass) can be accessed.

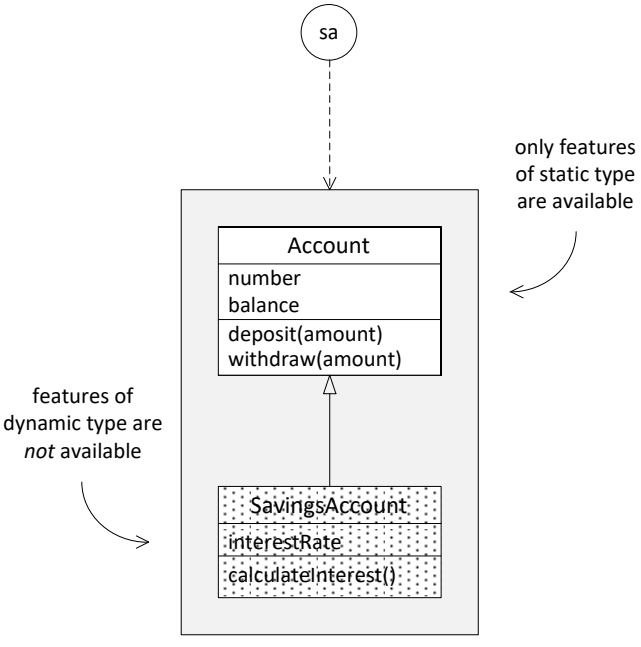


Figure 9.4: Polymorphic Variables Can Only Access Features of Static Type

Substitutability and polymorphism make it possible to treat instances of a parent class and instances of child classes in a uniform manner. For example, consider the following code:

```
Account[] accounts = new Account[3];
accounts[0] = new Account(10, 3500.00);
accounts[1] = new ChequingAccount(20, 5000.00);
accounts[2] = new SavingsAccount(30, 1500.00, 5.50);

for (int i=0; i<accounts.length; i++)
   System.out.println(accounts[i].toString());
```

In this example, instances of `Account`, `ChequingAccount`, and `SavingsAccount` are created and stored in an array of type `Account` since they are all `Account` instances based on the generalization hierarchy. The `for` loop then processes the objects in the array as if they were all `Account` instances.

Recall that the `toString()` method of the `Account` class implemented as follows:

```
public String toString() {
   String s;
   s = "Account number: " + number + " Balance: " + balance;
   return s;
}
```

Given this implementation, it is reasonable to assume that the following output will be generated from the code above:

```
Account number: 10 Balance: 3500.00
Account number: 20 Balance: 5000.00
Account number: 30 Balance: 1500.00
```

However, it may come as a surprise to find out that the actual output generated is the following:

```
Account number: 10 Balance: 3500.00
Account number: 20 Balance: 5000.00 Transaction Count: 0
Account number: 30 Balance: 1500.00
```

Somehow, it seems that the `toString()` method of `ChequingAccount` was used to generate the second line of output, even though the static type of the object is `Account`. To understand why this happened, the concept of *method binding* will now be discussed.

9.5.3 Method Binding

Consider the `withdraw()` method of the `Account` class. This method is declared as follows:

```
public void withdraw(double amount) {
   :
}
```

Suppose the `withdraw()` method is overridden in `ChequingAccount` using refinement:

```
public void withdraw(double amount) {
   super.withdraw(amount);
   numTransactions++;
}
```

Let's re-examine the following client code which was given in a previous section:

```
Account account;
ChequingAccount ca;
ca = new ChequingAccount(10, 5000.00);
account = ca;
```

After this code is executed, the `account` object variable refers to an instance of `ChequingAccount`. Consider now what happens when the `withdraw()` method is invoked on the `account` object variable:

```
account.withdraw(200.00);
```

Which `withdraw()` method is invoked at run-time? Will it be the one in `Account` or the one in `ChequingAccount`? In Java, method binding (i.e., choosing which method to execute) is based on the dynamic type of the object variable. This means that although the static type of `account` is `Account`, the `withdraw()` method that will be invoked is the one defined in the dynamic type, `ChequingAccount`. This makes sense since the `account` object variable is currently referring to a `ChequingAccount` instance.

If there is no `withdraw()` method in the dynamic type, the search for a `withdraw()` method continues upwards along the generalization hierarchy. The first `withdraw()` method found with the matching signature is selected. Note that a `withdraw()` method is guaranteed to be found somewhere along the generalization hierarchy; otherwise, the compiler would have generated an error.

9.5.4 The Reverse Polymorphism Problem

Consider the `accounts` array which was previously introduced to store `Account`, `ChequingAccount`, and `SavingsAccount` objects:

```
Account[] accounts = new Account[3];
```

Using substitutability, the array was populated with an instance of `Account`, an instance of `ChequingAccount`, and an instance of `SavingsAccount`.

Suppose we are given the `accounts` array and we don't know what is stored in the array. Certainly, all the objects can be considered to be of type `Account`. But, can we tell if an object is really an `Account`, a `ChequingAccount`, or a `SavingsAccount`? Also, if an object is indeed a `ChequingAccount` or a `SavingsAccount`, how can we get back the original object that was placed in the array? This problem is called the *reverse polymorphism problem*.

The reverse polymorphism problem is solved in Java in two stages:

- Finding out what is the type of an object using the `instanceof` operator[1]
- Casting an object whose type is higher in a generalization hierarchy to a type lower in the hierarchy (i.e., the reverse of substitutability).

Suppose that `o` is a reference to an object whose type is not known. The `instanceof` operator can be used as follows to determine the actual type of `o`:

```
if (o instanceof Account)
   // A: do something based on Account
else
if (o instanceof ChequingAccount)
   // B: do something based on ChequingAccount
```

The intention behind the code is that if `o` is an `Account` object, the code denoted by `A` in the example will be executed; and, if `o` is a `ChequingAccount` object, the code denoted by `B` will be executed. However, this does not happen. If `o` is an `Account` object *or* a `ChequingAccount` object, only the code denoted by `A` will be executed. This is because `ChequingAccount` inherits from `Account` so an instance of `ChequingAccount` can be considered to be an instance of `Account`. The same thing happens if `o` is replaced by the polymorphic variable `account` discussed in the previous section.

To handle situations such as these, the check for the more specific type (in this case, `ChequingAccount`) should be placed *before* the check for the more general type (`Account`). So, the code above should be written as follows:

```
if (o instanceof ChequingAccount)
   // B: do something based on ChequingAccount
else
if (o instanceof Account)
```

[1] It is also possible to use `o.getClass().getName()` to find out the class name of an object, `o`, in Java. However, this approach is not discussed in this chapter.

```
   // A: do something based on Account
```

Once we know the type of an object, we can cast it to this type. The following code shows how the `instanceof` operator together with casting can be used to solve the reverse polymorphism problem:

```
if (o instanceof ChequingAccount) {
   ChequingAccount ca = (ChequingAccount) o;
               // cast is perfectly safe

   // perform ChequingAccount operations with ca
}
else
if (o instanceof Account) {
   Account a = (Account) o;
               // cast is perfectly safe

   // perform Account operations with a
}
```

However, if the client code contains a number of `instanceof` checks with classes in a generalization hierarchy (such as `Account` and `ChequingAccount` above) it is likely that the code can be optimized using method overriding and polymorphism. This requires an examination of the specific behavior that is taking place inside the `if` statements. This behavior can be incorporated into a method. Using method overriding, different versions of the method can be written for each class in the hierarchy. Once this is done, polymorphic behavior is used to select the correct method to execute at run-time.

9.5.5 Parameter Passing and Return Types of Methods

We know that `SavingsAccount` is a subclass of `Account`. Suppose that a class `Bank` has a method `method1()` which accepts a parameter of type `Account` and returns nothing:

```
public void method1 (Account a) {
   // code for method1()
}
```

Suppose also that `b` is an instance of `Bank`. It is possible to invoke `method1()` on `b` using an instance of `SavingsAccount` rather than an instance of `Account`. This is the Principle of Substitutability at work:

```
SavingsAccount sa = new SavingsAccount(10, 1500.00, 5.50);
```

```
b.method1(sa);
```

`method1()` can make no assumptions about the parameter `a` and must treat it as an instance of `Account`. Of course, it can use the `instanceof` operator to find out the type of `a`.

Suppose that `Bank` has a method `method2()` which accepts a parameter of type `int` and returns an `Account` instance:

```
public Account method2 (int x) {
   // code for method2()
}
```

Using substitutability, `method2()` can return an instance of `Account` or an instance of any subclass of `Account` such as `SavingsAccount` (even indirectly). If the client code that invokes `method2()` wishes to use the return value, it must use an object variable of type `Account`. Again type checking with casting can be used to obtain the original type of the object:

```
Account a;
SavingsAccount sa;

a = b.method2(0);
if (a instanceof SavingsAccount) {
   sa = (SavingsAccount) a;  // cast a to SavingsAccount instance

   // perform SavingsAccount operations with sa
}
```

9.6 Preventing Inheritance

In Java it is possible to prevent a method from being overridden in a subclass by using the keyword `final` when defining the method. For example, the following method `deposit()` from the `Account` class cannot be overridden in a subclass:

```
public final void deposit(double amount) {
   balance = balance + amount;
}
```

Note that other methods of the parent class (or its ancestors) can be overridden as long as they are not declared as `final`.

It is also possible to prevent a class from having subclasses by using the keyword `final`. For example, to prevent subclasses of `Account` from being defined, the keyword `final` can be used as follows:

```
public final class Account {
   // body of Account class
}
```

It is not possible to define a class that inherits from a `final` class. So, the following cannot be done if `Account` is a `final` class:

```
public class SavingsAccount extends Account {
   // body of SavingsAccount class
}
```

9.7 Abstract Classes and Abstract Methods

For various reasons, it is sometimes necessary for client objects *not* to be able to create instances of a class. To prevent instances of a class from being created, it can be made *abstract*. For example, the `Account` class can be made abstract as follows:

```
public abstract class Account {
   // method body is the same as before
}
```

The `abstract` keyword prevents instances of `Account` from being created. However, the `Account` class can have attributes and methods such as those already discussed in this chapter.

An abstract class can be useful in situations where we want to factor out common attributes and behavior from several classes. An abstract class can be used to represent the factored-out attributes and behavior. The original classes can then inherit from the abstract class simplifying their design, development, and maintenance.

For example, suppose we didn't have an `Account` class but we had a `ChequingAccount` class and a `SavingsAccount` class, each with their own attributes and behavior. After factoring out the attributes and behavior in a new class `Account`, `ChequingAccount` and `SavingsAccount` can be created as subclasses of `Account`. All the attributes and methods of the `Account` class will be inherited by `ChequingAccount` and `SavingsAccount` in the way previously described.

Since it is not possible to create instances of an abstract class, we will use the term *concrete class* to describe a class from which instances can be created. The term *concrete method* will be used to denote the implementation of a method in a class (which is done by writing statements in the body of the method enclosed with braces). A concrete method is different from an *abstract method* which is simply a method declaration without a method body (or implementation). The following is an abstract method `withdraw()` in the `Account` class:

```
public abstract void withdraw (double amount);
```

Suppose the `withdraw()` method is present in the abstract class `Account` and that `ChequingAccount` inherits from `Account`. Since the `withdraw()` method is abstract in the `Account` class, it must be implemented in `ChequingAccount` if `ChequingAccount` is a concrete subclass of `Account`. If `ChequingAccount` does not implement the `withdraw()` method (by writing code in the method body), it must itself be declared as abstract.

An abstract method in a superclass enforces a protocol in all its subclasses without specifying how the protocol is implemented. Thus, the `withdraw()` method above forces all the subclasses of `Account` to have a `withdraw()` method which accepts a `double` parameter and returns nothing. However, each subclass is free to implement the `withdraw()` method according to its needs.

It should be noted that the name of an abstract class is written in italics in the UML. Abstract methods are also written in italics. Figure 9.5 shows how to specify in the UML that the `Account` class is abstract and that it contains an abstract `withdraw()` method.

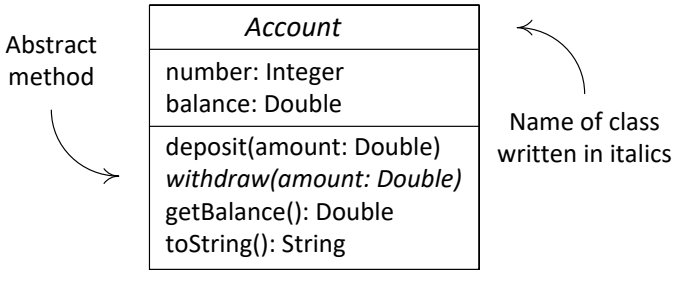


Figure 9.5: Abstract Class with an Abstract Method

Using substitutability, an abstract class can also be used as a type, simplifying the manipulation of child classes. For example, the following is possible in a client class:

```
Account[] accounts = new Account[2];
accounts[0] = new ChequingAccount(10, 5000.00);
accounts[1] = new SavingsAccount(20, 3500.00, 5.25);

for (int i=0; i<accounts.length; i++)
   System.out.println(accounts[i].toString());
```

In this example, the object references of the `ChequingAccount` instance and the `SavingsAccount` instance are of type `Account`. The object references can therefore be stored in an array of type `Account` and can be processed based on this general type instead of based on their specific types.

An abstract class can have zero or more concrete methods. It can also have zero or more abstract methods. A concrete class cannot contain an abstract method since if an instance is created, there will be no method implementation of that method. Finally, it should be mentioned that an abstract class can be a subclass of a concrete class.

9.8 Forms of Inheritance

Inheritance can be used in number of different ways. This section gives a short summary of the ways in which inheritance can be used (Budd, 1998).

- *Specialization.* The new class is a specialized form of the parent class but satisfies the specifications of the parent class in all relevant aspects. Thus, the Principle of Substitutability is explicitly upheld and the new class is a subtype of the parent class. This is the most ideal form of inheritance.

- *Specification.* The parent class defines behavior that is implemented in its child classes but not in the parent class. This guarantees that the child classes maintain a common interface. In Java, this is accomplished by using abstract methods in the parent class.

- *Construction.* The child class makes use of the behavior provided by the parent class but is not a subtype of the parent class. It is often a fast and easy route to developing new classes from existing ones.

- *Extension.* The child class adds new functionality to the parent class, but does not change any inherited behavior. Since the functionality of the parent is not modified, inheritance for extension does not violate the Principle of Substitutability and child classes are always subtypes.

- *Combination*. The child class inherits features from more than one parent class. This is known as *multiple inheritance* and is discussed in a later section of this chapter.

9.9 Benefits and Drawbacks of Inheritance

There are many benefits that can be gained by using inheritance when developing an object-oriented application. A major benefit is that of software reusability. When a class inherits behavior from another class, the code that provides the behavior does not have to be rewritten. Reusable code is also more reliable since there are more opportunities for discovering errors.

Another important benefit of inheritance is that of code sharing when two or more classes inherit from a single parent class. The code in the parent class only has to be written once resulting in more reliable code that requires less maintenance. Also, when two or more classes inherit from the same parent class, the behavior they inherit will be the same in all cases. This makes it easier to guarantee that interfaces to similar objects are indeed similar.

There are various problems associated with using inheritance in an object-oriented application. These include the following:

- Inheritance introduces inheritance coupling; instead of features being encapsulated in one class, they are scattered throughout classes in the inheritance hierarchy, increasing the coupling between classes.

- In medium-sized and large inheritance hierarchies, it is often difficult to understand what features are present in the classes that are lower in the hierarchy.

- Modifications to ancestor classes may cause unexpected effects on descendant classes thereby reducing reliability.

9.10 Multiple Inheritance

Multiple inheritance permits a class to have more than one parent class and to inherit features from all its parents. This permits mixing of information from two or more sources. A class with two or more parent classes is called a *join class*. Figure 9.6 is a UML diagram showing how a `VestedHourlyEmployee` class inherits from both an `HourlyEmployee` class and a `VestedEmployee` class (Rumbaugh et al, 1991). `VestedHourlyEmployee` is therefore a join class.

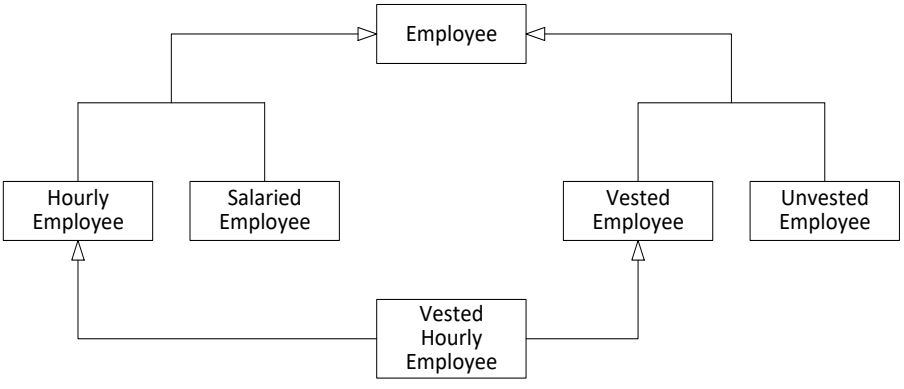


Figure 9.6: Multiple Inheritance

Note that a *salaried employee* is one who is paid at an annual or monthly rate. An *hourly employee* is one who is paid by the hour. A *vested employee* is one who has worked for an organization for a certain period of time and becomes entitled to full pension benefits. An *unvested employee* is one who is not yet entitled to full pension benefits.

Multiple inheritance can be useful in practical programming situations. However, it can be very complex to implement in an object-oriented programming language. Two major challenges in implementing multiple inheritance are:

1. How to deal with name clashes where features with the same name (e.g., instance variables and methods) are inherited from different parent classes. The solution usually involves a combination of *renaming* and *redefinition* (Budd, 1998).

2. How to inherit from two classes that themselves inherit from a common parent class. The major issue here is the sharing of data from the common parent: do we want two copies of the data or only one copy?

The designers of Java decided that the complexity of multiple inheritance outweighed the benefits so Java only supports single inheritance. Two workarounds will now be presented for achieving the effects of multiple inheritance in Java. These workarounds make use of *delegation*, an implementation mechanism by which an object forwards an operation to another object for execution.

The first workaround is *delegation using aggregation of roles* (Rumbaugh et al, 1991). A superclass in a generalization hierarchy with multiple inheritance is converted to an aggregate in which each component replaces a generalization relationship. Figure 9.7 shows how the model of Figure 9.6 can be restructured using delegation.

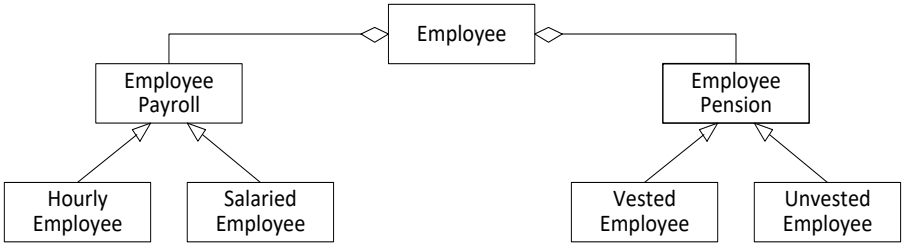


Figure 9.7: Multiple Inheritance Using Delegation

Two new parent classes are created, `EmployeePayroll` and `EmployeePension`. `EmployeePayroll` becomes the parent class of `HourlyEmployee` and `SalariedEmployee`. `EmployeePension` becomes the parent class of `VestedEmployee` and `UnvestedEmployee`. `Employee` can now be modeled as an aggregation of `EmployeePayroll` and `EmployeePension`.

Note that the join class, `VestedHourlyEmployee` is not explicitly created. Its functionality is obtained by combining the functionality of `HourlyEmployee` with the functionality of `VestedEmployee`, both of which are stored as instances in the `Employee` class. Behaviors from the `HourlyEmployee` class are delegated to the `EmployeePayroll` component by the `Employee` class. Similarly, behaviors from the `VestedEmployee` class are delegated to the `EmployeePension` component.

The second workaround is to *inherit from the most important class in the hierarchy and delegate the rest through aggregation* (Rumbaugh et al, 1991). This is illustrated in Figure 9.8 where the most important superclass is `Employee`. `Employee` is also an aggregation of `EmployeePension` just like in the first workaround. The `HourlyEmployee` class inherits from `Employee` as in Figure 9.6. The functionality of the join class, `VestedHourlyEmployee`, is achieved by combining the functionality of `HourlyEmployee` with delegation to the `VestedEmployee` aggregate.

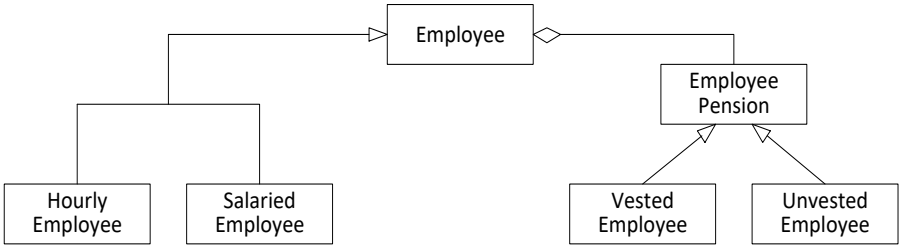


Figure 9.8: Multiple Inheritance Using Inheritance and Delegation

It should be mentioned that the semantics of substitutability are lost when delegation is used. For example, it is expected that an instance of `VestedHourlyEmployee` can take the place of an instance of `HourlyEmployee` or an instance of `VestedEmployee`. Using the workarounds above, there is no explicit `VestedHourlyEmployee` class so substitutability is impossible.

Exercises

1. Explain what is meant by an inheritance hierarchy. How is generalization different from specialization?

2. Distinguish between a direct subclass of a class `A` and an indirect subclass of a class `A`.

3. Suppose that `B` is a subclass of `A`. An instance of `B` is stored in a polymorphic variable of type `A`. With respect to the polymorphic variable, explain how method binding takes place at run-time.

4. Describe the reverse polymorphism problem and explain how it is usually solved in an object-oriented program.

5. Suppose we would like to create a subclass `C` which inherits from two classes `A` and `B`. Since multiple inheritance is not possible in Java, describe a workaround that can provide almost the same functionality.

6. Discuss two benefits and two drawbacks of inheritance.

7. `SavingsAccount` is a subclass of Account. Suppose a method of a `Bank` class has a parameter of type `Account`. Can a `SavingsAccount` object be supplied as an argument to this method? Should `Bank` be allowed to have another method that takes a parameter of type `SavingsAccount`? Investigate how Java deals with this issue.

8. If `X` is an immutable class, it is possible for a class to inherit from `X` and add attributes which are mutable. Explain how this can be prevented.