Table of Contents

Table of Contents

Table of Contents

Table of Contents

CHAPTER 50 Inheritance

Inheritance enables you to define a new class based on an existing class definition. The new class will be similar to the existing class, but will have some new characteristics. This makes programming easier, because you can build upon your previous work instead of starting out from scratch.

Inheritance (and other features of object oriented languages) is responsible for the recent enormous increase in the features and complexity of modern software. Programmers are able to build upon previous work and continuously improve and upgrade software. Graphical user interfaces define each visual component (windows, buttons, sliders, and icons) by using inheritance with a "toolkit" of basic components.

This chapter discusses the syntax and semantics of inheritance using some simple examples. The next chapter continues the discussion using some larger examples.

Chapter Topics:

- *Single Inheritance*
- *IS−A Relationship*
- *Class Hierarchies*
- *Syntax of Java Inheritance*
- *The* `super` *Reference*
- *Method Overriding*

QUESTION 1:

Couldn't you get the features of inheritance by just copying code and modifying it?

A good answer might be:

Yes. But if the original code is still in use you now have redundant methods in the original and in the copy. This leads to confusion.

Single Inheritance

The class that is used to define a new class is called a **parent** class (or superclass or base class.) The class based on the parent class is called a **child** class (or subclass or derived class.)

In Java, (unlike with humans) children inherit characteristics from *just one* parent. This is called **single inheritance**. Some languages allow a child to inherit from more than one parent. This is called **multiple inheritance**. With multiple inheritance, it is sometimes hard to tell which parent contributed what characteristics to the child (as with humans.) Java avoids these problems by using single inheritance.

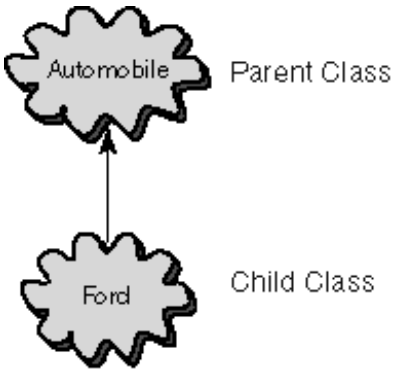
QUESTION 2:

(Thought question:) Can a parent class have more than one child class? (Hint: think about humans.)

A good answer might be:

Yes. A child can have only one parent, but a parent can have any number of children.

Is−a Relationship



The picture shows a parent class and a child class. The line between them shows the "is−a" relationship. The arrow points **to** the parent class **from** the child class. The picture can be read as "a Ford is−a automobile."

The clouds represent classes. That is, the picture does not show any particular automobile or any particular Ford, but shows that the class Ford is a child of the class Automobile.

> **Inheritance is between *classes*, not between objects.**

A parent class is a blueprint that is followed when an object is constructed. A child class of the parent is another blueprint (that looks much like the original), but with added features. The child class is used to construct objects that look like the parent's objects, but with added features.

QUESTION 3:

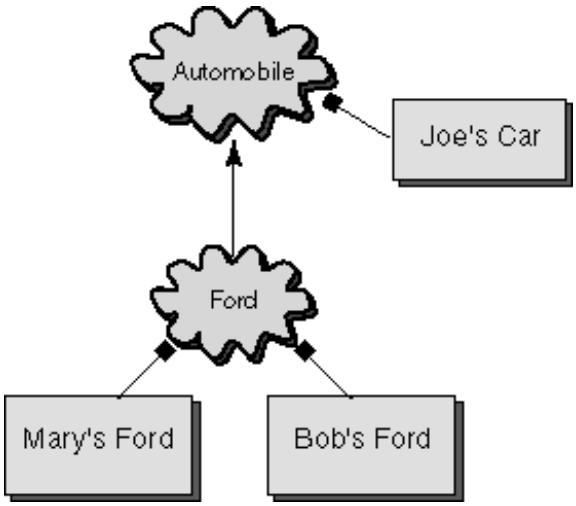
Next

How many objects of a given class can be constructed? (Hint: how many houses can be build from one blueprint?)

A good answer might be:

As many as required by the program. A class is a pattern that can be followed an unlimited number of times.

Is−a Relationship



The picture shows a parent class and a child class, and some objects that have been constructed from each. These objects are shown as rectangles (to convey the idea that they are more real than the classes, which are only designs.)

In the picture, "Joe's car," "Mary's Ford," and "Bob's Ford" represent actual objects. The cloudy classes represent designs that were used to construct the objects.

QUESTION 4:

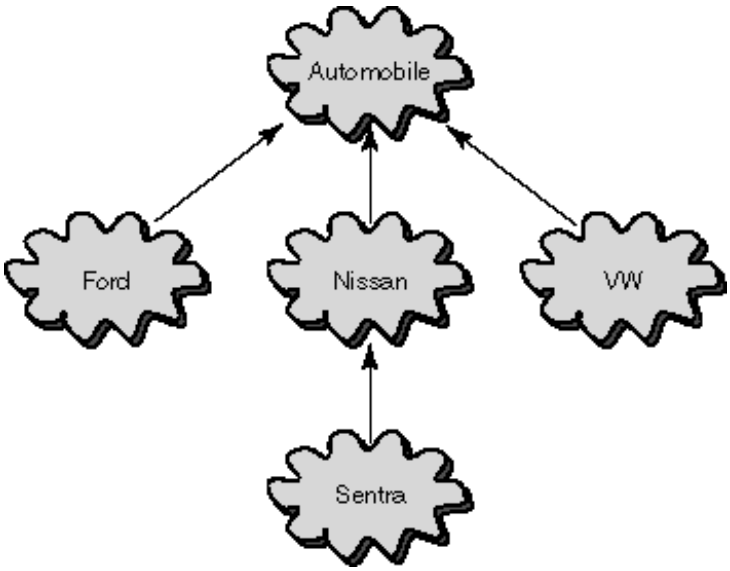
Next

- How many *types* of automobile are there in the diagram?
- How many *automobiles* are there in the diagram?

A good answer might be:

- How many *types* of automobile are there in the diagram?
  - ♦ Two the general type "Automobile" and a specific type, "Ford."
- How many *automobiles* are there in the diagram?
  - ♦ Three.

Hierarchies



This picture shows a *hierarchy* of classes. It shows that "Ford is−a automobile," "Nissan is−a automobile," and that "VW is−a automobile." It also shows that "Sentra is−a Nissan."

In a hierarchy, each class has at most one parent but might have several children classes. The class at the top of the hierarchy has no parent. This class is sometimes called the **root** of the hierarchy.

QUESTION 5:

Next

What is the parent class of the class Sentra?

A good answer might be:

Nissan.

Relative Terms

The terms "parent class" and "child class" are not absolute. A class is the parent for a child class derived from it and is the child of the class it was derived from. Just as with human relationships an individual is a child to some relatives and a parent to others. The syntax for deriving a child class from a parent class is:

```
class childClass extends parentClass
{
   // new characteristics of the child class go here

}
```

You have already seen this in creating applets:

```
class yourApplet extends Applet
{
  // new characteristics of your applet go here

}
```

Many of the characteristics of the applets you have written were inherited from the base class *Applet*. The base class has code to handle all the details of interfacing with the web browsers, with setting up graphics, and other things. You get all that automatically by inheritance. All you need to do is add the additional characteristics you want.

QUESTION 6:

(Thought question:) Can a parent class be one of your own classes (not a predefined class like Applet)?

A good answer might be:

Yes. Programming in Java consists mostly of creating class hierarchies and instantiating objects from them.

Tape Store Example

Here is a program that uses a class VideoTape to represent tapes available at a video tape rental store. Inheritance is not explicitly used in this program (so far).

```
class VideoTape
{
  String  title;    // name of the item
  int     length;   // number of minutes
  boolean avail;    // is the tape in the store?

  // constructor
  public VideoTape( String ttl )
  {
    title = ttl; length = 90; avail = true;
  }

  // constructor
  public VideoTape( String ttl, int lngth )
  {
    title = ttl; length = lngth; avail = true;
  }

  public void show()
  {
    System.out.println( title + ", " + length + " min. available:" + avail );
  }

}

class TapeStore
{
  public static void main ( String args[] )
  {
    VideoTape item1 = new VideoTape("Jaws", 120 );
    VideoTape item2 = new VideoTape("Star Wars" );

    item1.show();
    item2.show();
  }
}
```

You can copy this program to an editor, save it, and run it.

QUESTION 7:

What will this program print?

A good answer might be:

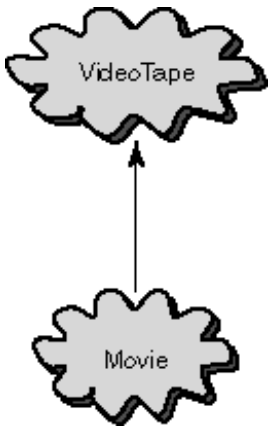
```
Jaws, 120 min. available: true
Star Wars, 90 min. available: true
```

Using Inheritance

The VideoTape class has basic information in it, and would be OK for documentaries and instructional tapes. But movies need more information. Let us make a class that is similar to VideoTape, but has the name of the director and a rating.

```
class VideoTape
{
  String  title;    // name of the item
  int     length;   // number of minutes
  boolean avail;    // is the tape in the store?

  // constructor
  public VideoTape( String ttl )
  {
    title = ttl; length = 90; avail = true;
  }

  // constructor
  public VideoTape( String ttl, int lngth )
  {
    title = ttl; length = lngth; avail = true;
  }

  public void show()
  {
    System.out.println( title + ", " + length + " min. available:" + avail );
  }

}

class Movie extends VideoTape
{
  String  director;     // name of the director
  String  rating;       // G, PG, R, or X

  // constructor
  public Movie( String ttl, int lngth, String dir, String rtng )
  {
    super( ttl, lngth );                // use the super class's constuctor
    director = dir;  rating = rtng;     // initialize what's new to Movie
  }

}
```

The class `Movie` is a subclass of `VideoTape`. An object of type `Movie` has the following members in it:



member	
title	inherited from VideoTape
length	inherited from VideoTape
avail	inherited from VideoTape
show()	inherited from VideoTape
director	defined in Movie
rating	defined in Movie

Both classes are defined: the `VideoTape` class can be used to construct objects of that type, and now the `Movie` class can be used to construct objects of the `Movie` type.

QUESTION 8:

Are both variables and methods inherited?

A good answer might be:

Yes the method `show` in the superclass `VideoTape` is inherited in the subclass `Movie`.

Using a Super Class's Constructor

The class definition for `VideoTape` has a constructor that initializes the member data of `VideoTape` objects. The class `Movie` has a constructor that initializes the data of `Movie` objects. The constructor for class `Movie` looks like this:

```
// constructor
public Movie( String ttl, int lngth, String dir, String rtng )
{
  super(ttl, lngth);                // use the super class's constuctor
  director = dir;  rating = rtng;   // initialize the members new to Movie
}
```

The statement `super(ttl, lngth)` invokes a constructor of the parent to initialize some of the data. It is called `super()` because the parent of a class is sometimes called its superclass. There are two constructors in the parent. The one that is invoked is the one that matches the argument list in `super(ttl, lngth)`. Then the next two statements (on one line) initialize the members that only `Movie` has.

If it is used, `super()` (both with and without arguments) must be the first statement in the subclass's constructor.

QUESTION 9:

Can `super()` be used twice in a constructor?

A good answer might be:

No, because the second use would not be the first statement. (This is a syntactical answer. You could also answer that semantically using a constructor twice for one object makes no sense.)

The Constructor Invoked by `super( )`

A constructor for a child class always starts with an invocation of one of the constuctors in the parent class. If the parent class has several constructors then the one which is invoked is determined by matching argument lists.

For example, we could define a second constructor for `Movie` that does not include an argument for length. It starts out by invoking the parent constructor that does not have an argument for length:

```
// alternate constructor
public Movie( String ttl, String dir, String rtng )
{
  super( ttl );    // invoke the matching parent class constructor
  director = dir;  rating = rtng;     // initialize members unique to Movie
}
```

QUESTION 10:

Does a child constructor always invoke a parent constructor?

A good answer might be:

Yes.

No−argument Constructor

Examine the following proposed constructor for `Movie`:

```
// proposed constructor
public Movie( String ttl, int lngth, String dir, String rtng )
{
  title = ttl;       // do what the parent's constuctor does.
  length = lngth;
  avail = true;

  director = dir; rating = rtng;
}
```

It looks like there is no need to invoke the parent class's constructor since all variables are initialized in this one. However a constructor from the parent class is always invoked even if you don't explicity ask for it. The Java compiler regards the above code as "shorthand" for this:

```
// proposed constructor
public Movie( String ttl, int lngth, String dir, String rtng )
{
  super();           // invoke the parent's no−argument constructor
  title = ttl;       // do what the parent's constuctor does.
  length = lngth;
  avail = true;

  director = dir; rating = rtng;
}
```

As always, `super()` comes first, even if you don't write it in. If the parent does not have a no−argument constructor, then using this "shorthand" causes a syntax error.

In our program the class definition for `VideoTape` (the superclass) lacks a no−argument constructor. The proposed constructor (above) calls for such a constructor so it would cause a syntax error.

QUESTION 11:

(Review:) When does a class have a no−argument constructor?

A good answer might be:

1. The programer can write a no−argument constructor for a class.
2. If the programmer does not write any constructors for a class, then a no−argument constructor (called the default constructor) is automatically supplied.
3. If the programmer writes even one constructor for a class then the default constructor is <u>not</u> supplied.

Instantiating `Movie`

In the example program, the class definition for `VideoTape` includes a constructor, so the default constructor is not present. So the constructor proposed for `Movie` causes a syntax error. Let us not use it.

Here is an example program that makes use of the two classes:

```
class TapeStore
{
  public static void main ( String args[] )
  {
    VideoTape item1 = new VideoTape("Microcosmos", 90 );
    Movie     item2 = new Movie("Jaws", 120, "Spielberg", "PG" );
    item1.show();
    item2.show();
  }
}
```

(You can run this program by using NotePad and copying and pasting the class definitions from the previous pages.) When you run it, you will get the following output:

```
Microcosmos, 90 min. available:true
Jaws, 120 min. available:true
```

The statement `item2.show()` calls the `show()` method of `item2`. This method was inherited *without change* from the class `VideoTape`. This is what it looks like:

```
  public void show()
  {
    System.out.println( title + ", " + length + " min. available:" + avail );
  }
```

It does not mention the new variables that have been added to objects of type `Movie`, so nothing new is printed out.

QUESTION 12:

Why not change `show()` in `VideoTape` to include the line

```
    System.out.println( "dir: " + director + rating );
```

A good answer might be:

Because the class `VideoTape`, does not have the variables `director` nor `rating`, so its `show()` method can't use them.

Overriding Methods

We need a new `show()` method in the class `Movie`:

```
  // added to class Movie
  public void show()
  {
    System.out.println( title + ", " + length + " min. available:" + avail );
    System.out.println( "dir: " + director + "  " + rating );
  }
```

Now, even though the parent has a `show()` method the new definition of `show()` in the child class will **override** the parent's version.

A child's method **overrides** a parent's method when it has the same signature as a parent method. Now the parent has its method, and the child has its own method with the same name.

An object of the parent type will include the method given in the parent's definition. An an object of the child type will include the method given in the child's definition.

With the change in the class `Movie` the following program will print out the full information for both items.

```
class TapeStore
{
  public static void main ( String args[] )
  {
    VideoTape item1 = new VideoTape("Microcosmos", 90 );
    Movie     item2 = new Movie("Jaws", 120, "Spielberg", "PG" );
    item1.show();
    item2.show();
  }
}
```

The line `item1.show()` calls the `show()` method defined in `VideoTape`, and the line `item2.show()` calls the `show()` method defined in `Movie`.

```
Microcosmos, 90 min. available:true
Jaws, 120 min. available:true
dir: Spielberg PG
```

QUESTION 13:

Does the definition of `show()` in the `Movie` class include some code that is already written?

A good answer might be:

Yes looks like we wrote the same code twice.

Using super in a Child's Method

Sometimes (as in the example) you want a child class to have its own method, but that method includes everything the parent's method does. You can use the `super` reference in this situation. For example, here is `VideoTape's` method:

```
public void show()
{
  System.out.println( title + ", " + length + " min. available:" + avail );
}
```

Here is `Movie's` method without using `super:`

```
public void show()
{
  System.out.println( title + ", " + length + " min. available:" + avail );
  System.out.println( "dir: " + director + "  " + rating );
}
```

`Movie`'s method would better be written using `super:`

```
public void show()
{
  super.show();
  System.out.println( "dir: " + director + "  " + rating );
}
```

Unlike the case when `super` is used in a constructor, inside a method `super` does not have to be used in the first statement.

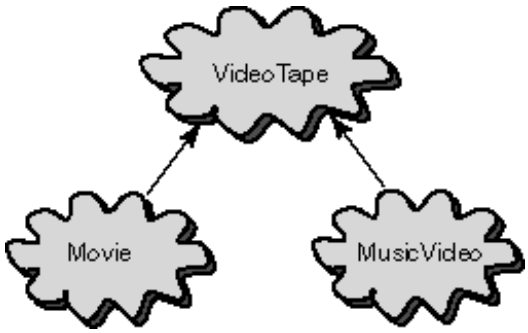
QUESTION 14:

Think of *two* reasons why using `super` in this way is a good thing to do.

A good answer might be:

1. You should not have to write the same code more than once.
2. A change made to the method in the parent class is inherited by the child class.

Music Video Class



So far the video rental application has two classes: `VideoTape` and `Movie`. Say that you wanted to create a new class, `MusicVideo` that will be like `VideoTape` but will have two new instance variables: `artist` (the name of the performer) and `category` ("R&B", "Pop", "Classical", "Other"). Both of these will be Strings. The `MusicVideo` class will need its own constructor and its own `show()` method. Here is the parent class:

```
class VideoTape
{
  String  title;    // name of the item
  int     length;   // number of minutes
  boolean avail;    // is the tape in the store?

  // constructor
  public VideoTape( String ttl, int lngth )
  {
    title = ttl; length = lngth; avail = true;
  }

  public void show()
  {
    System.out.println( title + ", " + length + " min. available:" + avail );
  }

}
```

The new class looks like this:

```
class _______________ extends ___________________
{
  String  _______________;     //  the artist
  String  _______________;     //  the music category

  // constructor will go here (skip for now)

  // show method will go here (skip for now)

}
```

QUESTION 15:

Next

Fill in the blanks. For now, leave out the constructor and the `show()` method.

A good answer might be:

The new class definition is (in part) below.

Adding a Constructor

The `MusicVideo` class is a subclass of `VideoTape`. A skeleton of the definition is below. The `Movie` class is not shown, but is also a subclass of `VideoTape`. Remember that a class can have several subclasses.

```
class VideoTape
{
  String  title;    // name of the item
  int     length;   // number of minutes
  boolean avail;    // is the tape in the store?

  // constructor
  public VideoTape( String ttl, int lngth )
  {
    title = ttl; length = lngth; avail = true;
  }

  public void show()
  {
    System.out.println( title + ", " + length + " min. available:" + avail );
  }

}

class MusicVideo extends VideoTape
{
  String artist;
  String category;

  // The constructor will go here

  // The show() method will go here

}
```

QUESTION 16:

What instance variables are part of the class `MusicVideo`?

A good answer might be:

`MusicVideo` inherits `title`, `length`, and `avail` from its superclass and adds `artist` and `category`.

Writing the Constructor

Notice that `MusicVideo` inherits only from its superclass `VideoTape`. It does not inherit anything from its sibling class `Movie`. Here is the definition so far:

```
class VideoTape
{
  String  title;    // name of the item
  int     length;   // number of minutes
  boolean avail;    // is the tape in the store?

  // constructor
  public VideoTape( String ttl, int lngth )
  {
    title = ttl; length = lngth; avail = true;
  }

  public void show()
  {
    System.out.println( title + ", " + length + " min. available:" + avail );
  }

}

class MusicVideo extends VideoTape
{
  String artist;
  String category;

  // The constructor will go here

  // The show() method will go here

}
```

A constructor is needed for `MusicVideo`. The constructor will use four parameters to initialize `title`, `length`, `artist` and `category`. Initialize `avail` to true without using a parameter.

QUESTION 17:

Write a constructor for `MusicVideo`. Use the `super` reference to the constructor in `VideoTape`.

A good answer might be:

The new constructor is seen below.

A whole new show( )

Here are the class definitions so far:

```
class VideoTape
{
  String  title;    // name of the item
  int     length;   // number of minutes
  boolean avail;    // is the tape in the store?

  // constructor
  public VideoTape( String ttl, int lngth )
  {
    title = ttl; length = lngth; avail = true;
  }

  public void show()
  {
    System.out.println( title + ", " + length + " min. available:" + avail );
  }

}

class MusicVideo extends VideoTape
{
  String artist;
  String category;

  // constructor
  public MusicVideo ( String ttl, int len, String art, String cat )
  {
    super( ttl, len );
    artist   = art;
    category = cat;
  }

  // The show() method will go here

}
```

Remember that the `super` reference (if it is used) must be in the first statement of the constructor.

To finish the `MusicVideo` class, write a `show()` method for it. Use a `super` reference to do the things already done by the superclass.

QUESTION 18:

Write the `show()` method.

A good answer might be:

The finished class is given below.

Complete `MusicVideo` Class

The subclass (MusicVideo) can use `super` anywhere within its own show() method. It is used in the first statement here because that is where it makes the most sense.

```
class VideoTape
{
  // stuff ommitted here
  public void show()
  {
    System.out.println( title + ", " + length + " min. available:" + avail );
  }

}

class MusicVideo extends VideoTape
{
  String artist;
  String category;

  // constructor
  public MusicVideo ( String ttl, int len, String art, String cat )
  {
    super( ttl, len );
    artist   = art;
    category = cat;
  }

  public void show()
  {
    super.show();
    System.out.println( "artist:" + artist + " style: " + category );
  }
}
```

QUESTION 19:

You may have noticed a major design flaw in the definitions of these classes none has a rental price! Say that it was your job to fix this problem. What changes will you make?

A good answer might be:

All video tapes will have a rental price, so the fixes should be made to the parent class `VideoTape`. If a new member varible `rent` is added to the base class, and if its constructor and its show() method are modified, the two child classes will inherit these changes. Fixing the parent class fixes all of its children.

Class `Object`

Remember that rule, that every constructor starts out by invoking a `super(...)` constructor even if the call is not explicitly written. Now look at the definition of `VideoTape`:

```
class VideoTape
{
  String  title;    // name of the item
  int     length;   // number of minutes
  boolean avail;    // is the tape in the store?

  // constructor
  public VideoTape( String ttl, int lngth )
  {
    title = ttl; length = lngth; avail = true;
  }

  public void show()
  {
    System.out.println( title + ", " + length + " min. available:" + avail );
  }
}
```

According to the rule the constructor implicitly does this:

```
  // constructor
  public VideoTape( String ttl, int lngth )
  {
    super();     // use the super class's constuctor
    title = ttl; length = lngth; avail = true;
  }
```

This is correct. All classes have a parent class (a super class) except one. The class at the top of the Java class hierarchy is called **`Object`**. If a class definition does not specify a parent class then it automatically has `Object` as a parent class. The compiler automatically assumes, for example:

```
class VideoTape extends Object
{
. . .
}
```

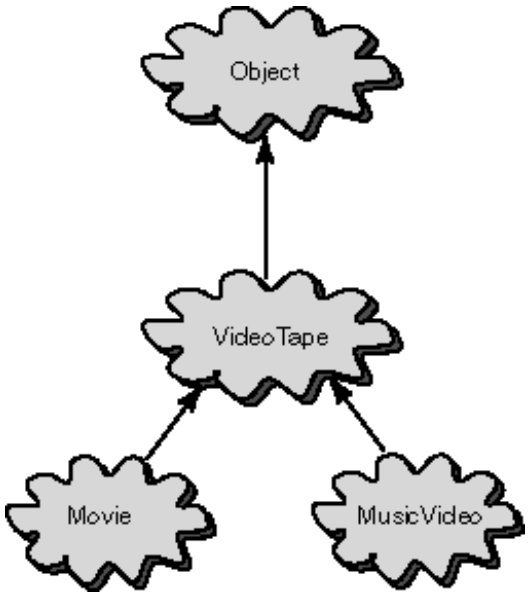
QUESTION 20:

Is it possible to write a class definition that does not have `Object` as its ultimate ancestor?

A good answer might be:

No.

Only One Class Hierarchy



Think about the answer a bit. If you define a class that does not explicitly extend another class, then it automatically extends `Object`. If you define the class so that it extends a class other than `Object`, then that class either extends another class or extends `Object`. Now that class in turn must either extend another class or extend `Object`. There is no way to end the chain except by (ultimately) extending `Object`.

You might cleverly try to have class A extend class B, and have class B extend class A. But this (and other loops) is not allowed.

Ultimately, every Java object inherits its object−like behavior from the class `Object`. When an object is constructed a chain of constructors is invoked starting with the one in `Object` and ending with the one in the requested class. The fundamental parts of the object are put together by the constructor in `Object`, then additional parts are added down the chain until the requested class is reached.

Construction starts with the constructor in `Object` because each constructor (except `Object`'s constructor) starts by invoking its `super()` constructor (with or without arguments).

QUESTION 21:

Next

Does a parent class have to be recompiled each time a new child class is derived from it?

A good answer might be:

No.

End of Chapter

You have reached the end of the chapter. You may wish to review the following. Click on a blue subject that interests you to go to where it was discussed. To get back here, click on the "back arrow" button of your browser.

- Single Inheritance.
- Is−a Relationship
- Class hierarchies
- The `super` Reference in a constructor.
- Method Overriding
- The `super` Reference in a child method.

Back to the main menu.

You have reached the end of the chapter.

creation: 2/14/99; revisions: 2/25/99, 7/17/99, 01/24/00, 07/28/2002

CHAPTER 51 Abstract Classes and Polymorphism

Classes in a hierarchy are related by the "is−a" relationship. For example, a Nissan is−a Automobile, and a Sentra is−a Nissan. This chapter discusses how reference variables are used for objects from different classes within a hierarchy. Another subject is the idea of an **abstract class** a class that cannot be instantiated but that can be the parent of other classes.

Chapter Topics:

- *Abstract Classes.*
- *Abstract Methods.*
- *Polymorphism.*

QUESTION 1:

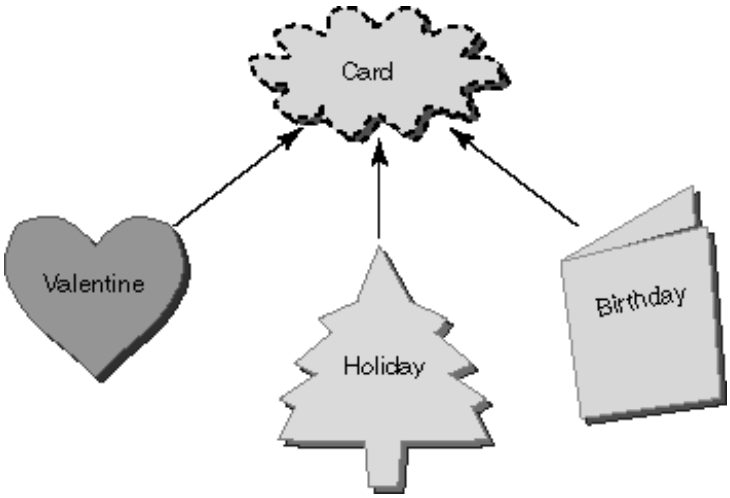
How many Valentine cards did you get this year?

A good answer might be:

As many as I sent.

Greeting Card Hierarchy

The example of this chapter is a hierarchy of greeting cards types. The parent class is `Card`, and its children classes are `Valentine`, `Holiday`, and `Birthday`.



A card object will have a `greeting()` method that writes out a greeting. Each type of card contains an appropriate greeting. The Holiday card says "Season's Greetings." The Birthday card says "Happy Birthday." The Valentine cards says "Love and Kisses."

In this example, an object must be an instance of one of the three child types: `Valentine`, `Holiday`, and `Birthday`. There will be no such thing as an object that is merely a "Card." The `Card` class represents the abstract concept of "Card." All actual card objects must be more specific.

QUESTION 2:

Next

If you had a shoe box full of greeting cards you had received over the years Valentine cards, birthday cards, holiday cards, and so on what would be a good label to write on the box?

A good answer might be:

"Cards" even though everything in the box is a specific type of card the box would be labeled with the general idea "Cards."

Abstract Class

For this example, all card objects are one of the three types (`Valentine`, `Holiday`, or `Birthday`) and the parent class `Card` is used only to group them into a hierarchy. There will never be an object that is just a card. This is useful to do, just as a store has all its various greeting cards in one display, arranged into several categories.

> An **abstract class** in Java is a class that is never instantiated. Its purpose is to be a parent to several related classes. The child classes inherit from the abstract parent class.

In hierarchy drawings (such as on the previous page), abstract classes are drawn with dotted lines. An abstract class is defined like this:

```
abstract class ClassName
{

   . . . . .  // definitions of methods and variables

}
```

Access modifiers such as `public` can be placed before `abstract.` Even though it can not be instantiated, an abstract class can define methods and variables that children classes inherit.

QUESTION 3:

Which of the card types has a `greeting()` method?

A good answer might be:

All of them.

Abstract Methods

In this example, each card class has its own version of the `greeting()` method. Each class has a `greeting()`, but each one is implemented differently. It is useful to put an abstract `greeting()` method in the parent class. This says that each child inherits the "idea" of `greeting()`, but each implementation is different. Here is the class definition of the abstract class `Card`:

```
abstract class Card
{
  String recipient;                 // name of who gets the card
  public abstract void greeting();  // abstract greeting() method
}
```

This is the complete definition of this abstract class. Notice the **abstract method**. Abstract classes can (but don't have to) contain abstract methods. Also, an abstract class can contain non−abstract methods, which will be inherited by the children.

> An **abstract method** has no body. (It has no statements.) It declares an access modifier, return type, and method signature followed by a semicolon. A non−abstract child class inherits the abstract method and must define a non−abstract method that matches the abstract method.

An `abstract` child of an `abstract` parent does not have to define non−abstract methods for the abstract signatures it inherits. This means that there may be several steps from an abstract base class to a child class that is completely non−abstract.

Since no constructor is defined in `Card` the default no−argument constructor is automatically supplied by the compiler. However, this constructor cannot be used directly because no `Card` object can be constructed.

QUESTION 4:

Are you feeling a bit abstract right now?

A good answer might be:

Not surprising. You must have got it from your abstract parents.

Holiday

This stuff takes some thought, and some practice, and then some thought followed by practice. It took years and years and years for computer scientists to agree on this idea, and they have not stopped arguing about it. If you are confused, you might wish to continue on to Ph. D. work.

Abstract classes are used to organize the "concept" of something that has several different versions in the children. The abstract class can include abstract methods and non−abstract methods.

Here is a class definition for class `Holiday`. It is a non−abstract child of an abstract parent:

```
class Holiday extends Card
{
  public Holiday( String r )
  {
    recipient = r;
  }

  public void greeting()
  {
    System.out.println("Dear " + recipient + ",\n");
    System.out.println("Season's Greetings!\n\n");
  }
}
```

The class `Holiday` is not an abstract class. Objects can be instantiated from it. Its constructor implicitly calls the no−argument constructor in its parent, `Card`, which calls the constructor in `Object`. So even though it has an abstract parent, `Holiday` objects are as much objects as any other.

- `Holiday` inherites the abstract method `greeting()` from its parent.
- `Holiday` must define a `greeting()` method that includes a method body (statements between braces).
- The definition of `greeting()` must match the signature given in the parent.
- If `Holiday` did not define `greeting()`, then `Holiday` would be declared an abstract class.
    - ♦ This would make it an abstract child of an abstract parent.

QUESTION 5:

Will each of the classes that inherit from `Card` have a `greeting()` method?

A good answer might be:

Yes by using an abstract class a programmer can make all children of that class look alike in important ways.

Not Everything in an abstract Class is abstract

Not everything defined in an abstract classes needs to be abstract. The variable `recipient` is defined in `Card` and inherited in the usual way. However, if a class contains even one abstract method, then the class itself has to be declared to be abstract. Here is a program to test the two classes.

```
abstract class Card
{
  String recipient;
  public abstract void greeting();
}

class Holiday extends Card
{
  public Holiday( String r )
  {
    recipient = r;
  }

  public void greeting()
  {
    System.out.println("Dear " + recipient + ",\n");
    System.out.println("Season's Greetings!\n\n");
  }
}

public class CardTester
{
  public static void main ( String[] args )
  {
    Holiday hol = new Holiday("Santa");
    hol.greeting();
  }
}
```

This is a complete, runable program. The urge to copy it to NotePad and get it running must be irresistable. Remember to call the file "CardTester.java" .

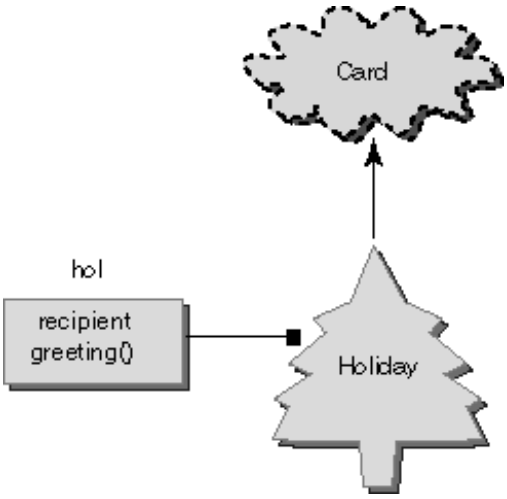
QUESTION 6:

Could you write this program without using an abstract class?

A good answer might be:

Yes.

Advantage of Abstract Classes



Abstract classes are a way of organizing a program. You can get the same thing done without using this way to organize. This is a matter of program design, which is not easy at all.

Here is a sample run of the program:

```
Dear Santa,

Season's Greetings!
```

The advantage of using an abstract class is that you can group several related classes together as siblings. Grouping classes together is important in keeping a program organized and understandable. The picture shows this program after its object has been constructed.

It would be nice to deal some other cards. Here is a skeleton for the `Birthday` class:

```
class Birthday extends _______________
{
  int age;

  public _______________ ( String r, int years )
  {
    recipient = r;
    age = years;
  }

  public void greeting()
  {
    System.out.println("Dear " + recipient + ",\n");
    System.out.println("Happy " + _______ + "th Birthday\n\n");
  }
}
```

QUESTION 7:

Next

Fill in the missing parts.

A good answer might be:

The missing parts have been filled in, below.

Valentine Card

Here is the complete birthday card:

```
class Birthday extends Card
{
  int age;

  public Birthday ( String r, int years )
  {
    recipient = r;
    age = years;
  }

  public void greeting()
  {
    System.out.println("Dear " + recipient + ",\n");
    System.out.println("Happy " + age + "th Birthday\n\n");
  }
}
```

The `Valentine` class is much the same, except for some added passion:

```
class Valentine extends Card
{
  int kisses;

  public Valentine ( String r, int k )
  {
    recipient = r;
    kisses    = k;
  }

  public void greeting()
  {
    System.out.println("Dear " + recipient + ",\n");
    System.out.println("Love and Kisses,\n");
    for ( int j=0; j<kisses; j++ )
      System.out.print("X");
    System.out.println("\n\n");
  }
}
```

QUESTION 8:

Each `message()` method from each of the sibling classes is different. Do they all meet the requirement of the abstract parent class, `Card`?

A good answer might be:

Yes. All that `Card` asked for was:

```
public abstract void greeting();
```

Each sibling did that in its own way.

Complete Program

Here is a complete program with all three card classes and objects of each type. If you were deficient in greeting cards this year, you might wish to copy this to NotePad and run it a few times.

```
import java.io.*;

abstract class Card
{
  String recipient;
  public abstract void greeting();
}

class Holiday extends Card
{
  public Holiday( String r )
  {
    recipient = r;
  }

  public void greeting()
  {
    System.out.println("Dear " + recipient + ",\n");
    System.out.println("Season's Greetings!\n\n");
  }
}

class Birthday extends Card
{
  int age;

  public Birthday ( String r, int years )
  {
    recipient = r;
    age = years;
  }

  public void greeting()
  {
    System.out.println("Dear " + recipient + ",\n");
    System.out.println("Happy " + age + "th Birthday\n\n");
  }
}

class Valentine extends Card
{
  int kisses;

  public Valentine ( String r, int k )
  {
    recipient = r;
    kisses    = k;
  }

  public void greeting()
  {
    System.out.println("Dear " + recipient + ",\n");
    System.out.println("Love and Kisses,\n");
    for ( int j=0; j < kisses; j++ )
      System.out.print("X");
    System.out.println("\n\n");
  }
}
```

```
public class CardTester
{
  public static void main ( String[] args ) throws IOException
  {
    String me;
    BufferedReader input = new BufferedReader( new InputStreamReader(System.in) );
    System.out.println("Your name:");
    me = input.readLine();

    Holiday   hol = new Holiday( me );
    hol.greeting();

    Birthday  bd  = new Birthday( me, 21 );
    bd.greeting();

    Valentine val = new Valentine( me, 7 );
    val.greeting();


  }
}
```

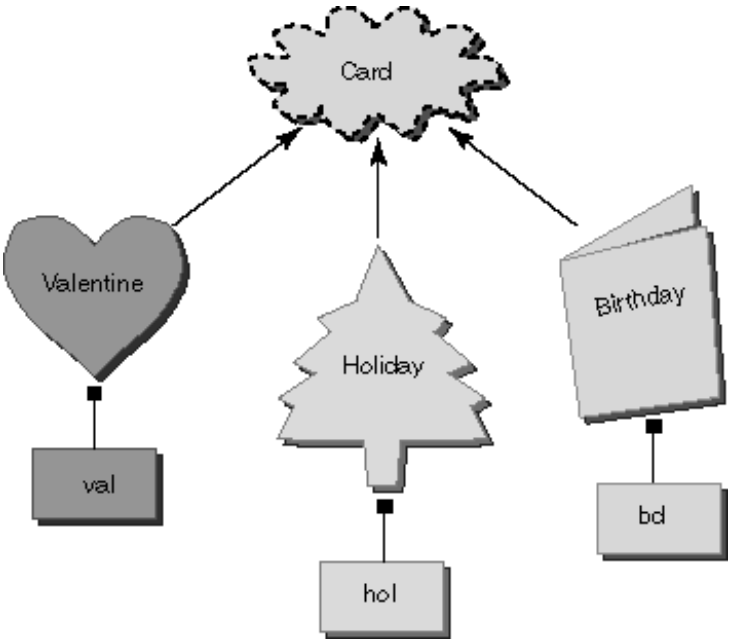
QUESTION 9:

This is a fairly long program 80 lines! Do you think that you could design a few more card classes without any problems?

A good answer might be:

Yes. The abstract `Card` class can be used as a model of what each new type of card should look like.

Objects of each Class



By using hierarchical organization and inheritance it is easy to add many more card classes and to create a well organized program. This was unthinkable not too many years ago. Here is a sample run of this program:

```
Your name:
Sue
Dear Sue,

Season's Greetings!

Dear Sue,

Happy 21th Birthday

Dear Sue,

Love and Kisses,

XXXXXXX
```

When the `main()` method has constructed its three objects, the situation is as in the picture. There are three classes that can be instantiated, and one object of each class has been instantiated. Of course, as many objects as you need of each type (except `Card`) can be instantiated.

QUESTION 10:

Next

There are 4 classes. Why can only 3 be instantiated?

A good answer might be:

The parent class `Card` is an abstract class and therefore cannot be instantiated.

Can't Instantiate an Abstract Class

You can't do the following:

```
   . . . .

  public static void main ( String[] args ) throws IOException
  {
     . . . .

    Card card = new Card() ;        // can't instantiate abstract class
    card.greeting() ;

     . . . .
  }
```

Because `Card` is an abstract class, the compiler flags this as a syntax error. `Card` does have a constructor that is (implicitly) invoked by its children, but it cannot be invoked directly. However, the following *is* OK:

```
   . . . .

  public static void main ( String[] args ) throws IOException
  {
     . . . .

    Card card = new Valentine( "Joe", 14 ) ;      // a Valentine is-a Card
    card.greeting() ;

     . . . .
  }
```

It is OK to save a reference to a `Valentine` object in a reference variable of type `Card` because `Valentine` **is-a** `Card`. You can think of the reference variable `Card` as being a card rack designed to hold any type of a `Card`.

QUESTION 11:

Do you think it would be OK to do the following?

```
Card card2 = new Holiday( "Bob" ) ;
Card card3 = new Birthday( "Emily" ) ;
```

A good answer might be:

```
Card card2 = new Holiday( "Bob" ) ;
Card card3 = new Birthday( "Emily" ) ;
```

Yes, both are correct, since `Holiday` is-a `Card` and `Birthday` is-a `Card`.

Using Parent Class Reference Variables

A reference variable of class "C" can be used with any object that is related by inheritance to class "C". For example, a `Card` reference variable `card2` can hold a reference to a `Holiday` object, a `Valentine` object, or a `Birthday` object. Usually, references to a parent class are used to hold children objects.

Important Point:

> When a method is invoked, it is the class of the object (not of the variable) that determines which method is run.

This is what you would expect. The method that is run is part of the object (at least conceptually). For example:

```
Card card = new Valentine( "Joe", 14 ) ;
card.greeting();

Card card2 = new Holiday( "Bob" ) ;
card2.greeting();

Card card3 = new Birthday( "Emily", 12 ) ;
card3.greeting();
```

This will run the `greeting()` method for a `Valentine`, then it will run the `greeting()` method for a `Holiday`, then it will run the `greeting()` method for a `Birthday`. The type of the object in each case determines which version of the method is run.

QUESTION 12:

Is it necessary to use three different variables for this program fragment?

A good answer might be:

No. Just one could have be used for each object in succession (assuming that the program does no more than is shown.) See below.

Polymorphism

Polymorphism means "having many forms." In Java, it means that a single variable might be used with several objects of related classes at different times in a program. When the variable is used with "dot notation" `variable.method()` to invoke a method, exactly which method is run depends on the object that the variable currently refers to. Here is an example:

```
 . . . .                          // class definitions as before

public class CardTester
{
  public static void main ( String[] args ) throws IOException
  {

    Card card = new Holiday( "Amy" );
    card.greeting();                      //Invoke a Holiday greeting()

    card = new Valentine( "Bob", 3 );
    card.greeting();                      //Invoke a Valentine greeting()

    card = new Birthday( "Cindy", 17 );
    card.greeting();                      //Invoke a Birthday greeting()

  }
}
```

QUESTION 13:

Next

What will the program write?

A good answer might be:

```
Dear Amy,

Season's Greetings!

Dear Bob,

Love and Kisses,

XXX

Dear Cindy,

Happy 17th Birthday
```

Holding References but not Using them

What types of objects a reference variable can refer to:

> A variable can hold a reference to an object whose class is a descendent of the class of the variable.

The class of the object must be *a descendant* of the class of the variable that holds a reference to that object. *A descendant* of a class is a child of that class, or a child of a child of that class, and so on. Siblings are not descendants of each other (you did not inherit anything from your brother, thank goodness.)

Note: there are some complications to this rule which are skipped in these notes, since you will probably never need to know them. Here are some variable declarations:

```
Card      c;
Valentine v;
Birthday  b;
Holiday   h
```

QUESTION 14:

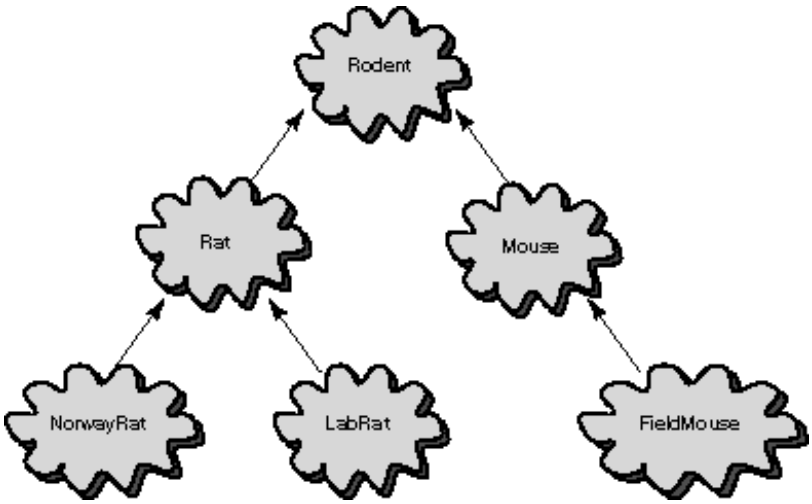
Which of the following are correct?

```
c = new Valentine("Debby", 8);
b = new Valentine("Elroy", 3);
v = new Valentine("Fiona", 3);
h = new Birthday ("Greg",  35);
```

A good answer might be:

```
Card      c;
Valentine v;
Birthday  b;
Holiday   h

c = new Valentine("Debby", 8);    //OK
b = new Valentine("Elroy", 3);    //WRONG
v = new Valentine("Fiona", 3);    //OK
h = new Birthday ("Greg",  35);   //WRONG
```

Another Hierarchy



Here is a picture of another hierarchy: There are no abstract classes in this hierarchy, so each class can be instantiated. As seen previously, the following is OK:

```
parentReference = childReference
```

A parent reference variable can hold a reference to an object of one of its child types (or a reference to one of their child types, and so on.) However, the opposite direction

```
// don't do this
childReference = parentReference
```

can not be done, unless the *parentReference* actually refers to a child object (this is rare.)

Here are some variables:

```
Rodent     rod;
Rat        rat;
Mouse      mou;
```

Look at the table and decide if each section of code is correct or not.

code section	OK or Not?	code section	OK or Not?
`rod = new Rat();`		`rod = new FieldMouse();`	
`mou = new Rat();`		`mou = new Rodent();`	
`rat = new Rodent();`		`rat = new LabRat();`	
`rat = new FieldMouse();`		`rat = new Mouse();`	

QUESTION 15:

Next

There is still much more to learn about inheritance and polymorphism.

A good answer might be:

Rats.

End of Chapter

You may wish to review the following. Click on a blue subject that interests you to go to where it was discussed. To get back here, click on the "back arrow" button of your browser.

- Abstract class.
- Abstract method.
- When a class must be declared to be abstract.
- Advantage of using an abstract class.
- Can't instantiate an abstract class.
- Using a variable of a parent type.
- Polymorphism
- What types of objects a variable can refer to.

The next chapter will discuss more about polymorphism.

Back to the main menu. You have reached the end of the chapter. creation: 2/20/99; revisions: 7/16/99, 01/24/00, 07/22/02

CHAPTER 52 More About Polymorphism

This chapter continues the discussion of polymorphism begun in the previous chapter. Recall that *polymorphism* is when a reference variable may at different times in a program refer to objects of different (but related) classes.

Chapter Topics:

- *Signature of a method.*
- *Overriding abstract methods.*
- *Arrays of related objects.*
- *The Object class.*
- *The instanceof operator.*

QUESTION 1:

(Review:) What is a method's *signature*?

A good answer might be:

The *signature* of a method is the name of the method and types in its parameter list, something like this:

```
someMethod( int, double, String )
```

The return type is not part of the signature. The names of the parameters do not matter, just their types.

Signature

The name of a method is not enough to uniquely identify it because there may be several versions of a method, each with different parameters. Using the types in the parameter list says more carefully which method you want. The return type can't be used because of the following situation. Say that there are two `someMethod()` methods: one defined in a parent class like this:

```
int someMethod( int x, double y, String z )
{
  //method body
}
```

and another method defined in a child class like this:

```
short someMethod( int a, double b, String c )
{
  // completely different method body
}
```

Somewhere else in the program `someMethod()` is called:

```
long w = child.someMethod( 12, 34.9, "Which one?" );
```

The value on the right of the `=` could be either `int` or `short`. Both can be converted to a `long` value needed for the variable `w`. You can not tell which `someMethod()` matches the statement. In other words, the signatures of the two methods are not unique.

QUESTION 2:

Do these two methods have different signatures?

```
double aMethod( int x, String y)
{. . .}

short  aMethod( int a, double b)
{. . .}
```

A good answer might be:

Yes.

Overriding abstract Methods

An abstract class will usually contain abstract methods. An abstract method definition consists of:

- optional access modifier (`public, private,` and others),
- the reserved word `abstract`,
- the class of the return value,
- a method signature,
- a semi−colon.

No curly braces or method body follow the signature. Here is the abstract class `Parent` including the abstract `compute()` method:

```
abstract class Parent
{
  public abstract int compute( int x, String j);
}
```

If a class has one or more abstract methods it must be declared to be `abstract`. An abstract class may have methods that are not abstract (the usual sort of method). These methods are inherited by children classes in the usual way. A non−abstract child class of an abstract parent class must *override* each of the abstract methods of its parent.

- A non−abstract child must **override** each abstract method inherited from its parent by defining a method with the same signature and same return type.
    - ♦ Objects of the child class will include this method.
- A child may define **additional methods** with signatures different from the parent's method.
    - ♦ Child objects will include these methods in addition to the first one.
- It is an **error** if a child defines a method with the same signature as a parent method, but with a different return type.

These rules are not really as terrible as they seem. After working with inheritance for a while the rules will seem clear. Here is a child of `Parent`:

```
class Child extends Parent
{
    public int compute( int x, String j )
    { . . . }
}
```

The child's `compute()` method correctly overrides the parent's abstract method.

QUESTION 3:

Does the following correctly override the parent's abstract method?

```
class Child extends Parent
{
    public double compute( int x, String j )
    { . . . }
}
```

A good answer might be:

No. It is an **error** if a child defines a method with the same signature as a parent method, but with a different return type.

More Practice

Non−abstract children must override the abstract methods of their parents. Here is our situation:

```
class Parent
{
  public abstract int compute( int x, String j);
}

class Child extends Parent
{
   ____________________________________________  // child's compute method
}
```

Examine each of the following choices for filling the blank. What will the new definition do? **Override** the parent's method, define an **additional method**, or be an **error**? (If the choice defines an additional method, assume that the required method is also defined.)

method defined in Child	Override, Additional, or Error?
`public int compute( int x, String j ){ . . . }`	
`public int compute( int stuff, String str ){ . . . }`	
`public int Compute( int x, String j ){ . . . }`	
`public int compute( String j, int x){ . . . }`	
`public double compute( int x, String j){ . . . }`	

QUESTION 4:

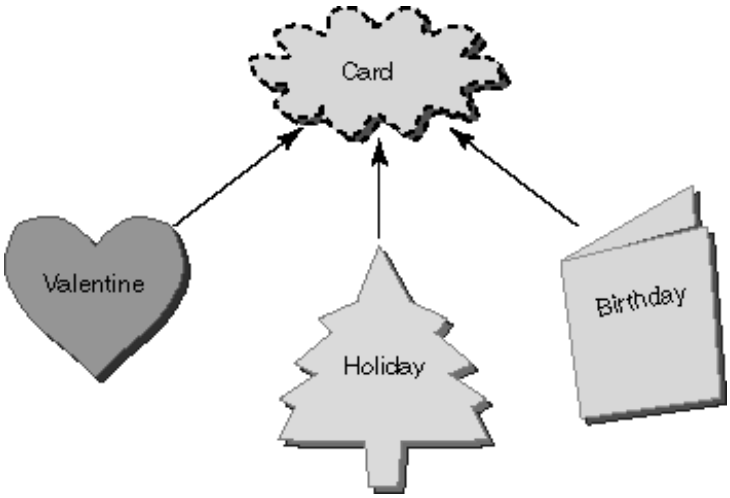
Next

Say that a child class defines just one method, and that method has the same name as the parent's abstract method, but with a different signature. Must the child class be *abstract*?

A good answer might be:

Yes. Since the child did not define a method with the same signature and same return type as the parent, it must be declared to be abstract. It is OK to have an abstract child of an abstract parent.

Card Hierarchy



Let us use the hierarchy of classes used in the previous chapter to illustrate these ideas. The diagram shows that the class `Card` is an abstract class (which, therefore, cannot be instantiated.)

`Card` has a single abstract method, `greeting()`. The other three classes are ordinary classes which can be instantiated. They inherit from `Card`, so each must define a `greeting()` method which is not abstract.

QUESTION 5:

Next

(Review:) Say that a method is invoked as here:

```
referenceVariable.greeting()
```

What determines which method is run? The class of the reference variable or the class of the object?

A good answer might be:

The class of the object.

Card Polymorphism

It makes sense that the object's method is invoked, since, after all, the method is part of the object and the method uses the object's data. Here is an example from the previous chapter:

```
 . . . .                          // class definitions as before

public class CardTester
{
  public static void main ( String[] args ) throws IOException
  {

    Card card = new Holiday( "Amy" );
    card.greeting();                     //Invoke a Holiday greeting()

    card = new Valentine( "Bob", 3 );
    card.greeting();                     //Invoke a Valentine greeting()

    card = new Birthday( "Cindy", 17 );
    card.greeting();                     //Invoke a Birthday greeting()

  }
}
```

The reference variable `card` is used three times, each time with a different class of object. Since `Card` is a parent to the three different classes, the variable `card` can be used for each.

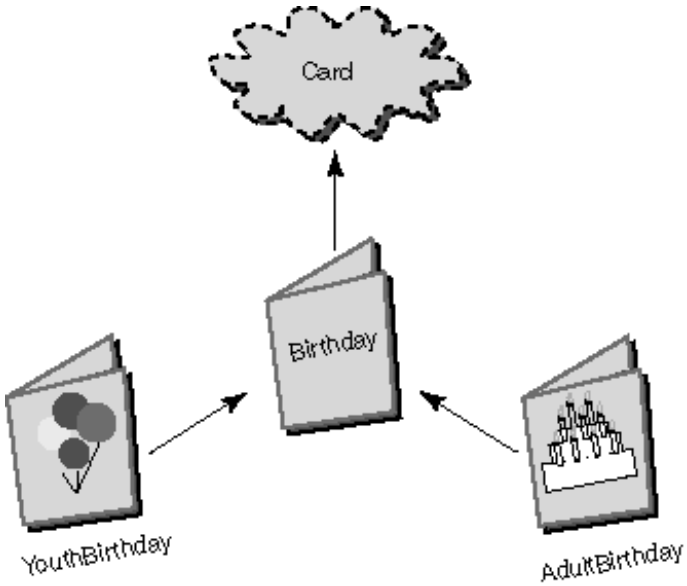
QUESTION 6:

Could a variable `Valentine val` be used with a `Holiday` object?

A good answer might be:

No. `Valentine val` and `Holiday` are siblings. Since they are not related by inheritance, a reference variable of one type cannot be used with an object of the other type.

Extended Hierarchy



The Card hierarchy is extended by adding two new classes:

- A YouthBirthday birthday card for young people.
    - This card will add the line "How you have grown!" to the usual birthday greeting.
- An AdultBirthday birthday card for old people.
    - This card will add the line "You haven't changed at all!" to the usual birthday greeting.

The class `Birthday` is the parent for these two new classes. The diagram omits the other two classes, but assume that they are still defined. Each of the new classes would ordinarily inherit the `greeting()` method from `Birthday`. But we want to override that method with a specialized method in each of the new classes.

Here is the class definition of `Birthday` from the previous chapter:

```
class Birthday extends Card
{
  int age;

  public Birthday ( String r, int years )
  {
    recipient = r;
    age = years;
  }

  public void greeting()
  {
    System.out.println("Dear " + recipient + ",\n");
    System.out.println("Happy " + age + "th Birthday\n\n");
  }
}
```

Here is a class definition of `YouthBirthday`:

```
class ____________  extends  _____________

{
  public  ____________ ( String r, int years )
  {
    ____________ ( r, years )
  }

  public void greeting()
  {
    ____________();
    System.out.println("How you have grown!!\n");
  }
}
```

QUESTION 7:

Fill in the blanks. Don't do this too quickly. Check the requirements and the hierarchy diagram.

A good answer might be:

The blanks are filled in, below:

super()

Invoke the superclass constructor by using `super.` In a constructor, `super` must appear before anything else (although in this example there isn't anything else.) Recall that even if you don't put it in explicitly the compiler will automatically put a call to `super` as the first thing a constructor does.

```
class YouthBirthday  extends  Birthday

{
  public  YouthBirthday ( String r, int years )
  {
    super ( r, years )
  }

  public void greeting()
  {
    super.greeting();
    System.out.println("How you have grown!!\n");
  }
}
```

When the `greeting()` method of a `YouthBirthday` object is invoked, first its superclass's method will run, then the rest of the new method will run.

QUESTION 8:

What will be written by the following:

```
YouthBirthday yb = new YouthBirthday( "Valerie", 7 );
yb.greeting();
```

A good answer might be:

```
Dear Valerie,

Happy 7th Birthday


How you have grown!!
```

Using Polymorphism

Assume that `AdultBirthday` has been defined. It will look much like `YouthBirthday` but with a slightly different birthday message. Here is a program fragment (part of `main()` in another class):

```
AdultBirthday ab = new AdultBirthday( "Walter", 47 );
ab.greeting();
```

It will write out:

```
Dear Walter,

Happy 47th Birthday


You haven't changed at all!
```

Now inspect the following code:

```
Card crd = new YouthBirthday( "Valerie", 7 );
crd.greeting();

crd      = new AdultBirthday( "Walter", 47 );
crd.greeting();

crd      = new Birthday( "Zoe", 30 );
crd.greeting();
```

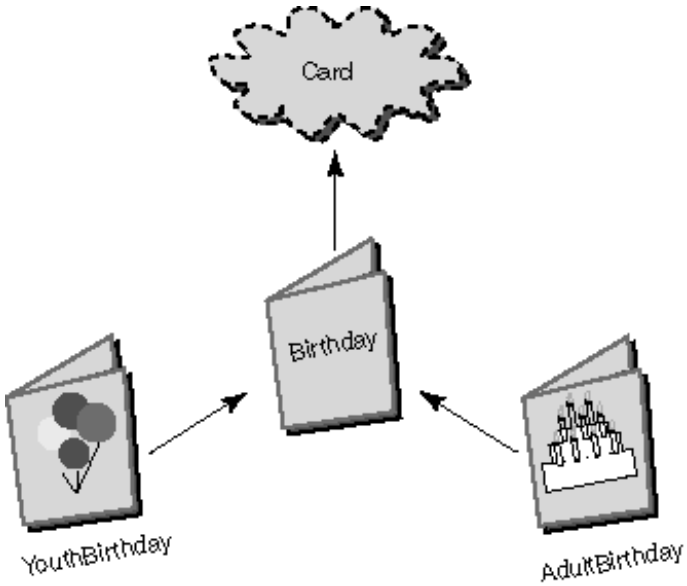
QUESTION 9:

Is this code correct? Can `crd` be used for each of the three objects?

A good answer might be:

Yes.

Example Output



The program will output:

```
Dear Valerie,

Happy 7th Birthday


How you have grown!!

Dear Walter,

Happy 47th Birthday


You haven't changed at all!

Dear Zoe,

Happy 30th Birthday
```

The abstract class `Card` is related to each of the three classes by inheritance, and so the variable `crd` can be used with each of them. Now inspect the following:

```
YouthBirthday birth;
birth = new Birthday( "Terry", 23 );
```

QUESTION 10:

Is this code correct?

Is this code correct?

```
YouthBirthday birth;
birth               = new Birthday( "Terry", 23 );
```

A good answer might be:

No. A reference variable for a child class (YouthBirthday) cannot be used for an object of a parent class (Birthday).

New Definition of `YouthBirthday`

Parents can accommodate children, but children can't accommodate parents. If you find this rule hard to remember, just think of yourself and your parents:

> It is OK for you to go home and stay in your parent's house for a few days, but it's not OK for them to stay in your dorm room for a few days.

Now let us re−write `YouthBirthday` in order to show more about polymorphism. Say that you want two `greeting()` methods for `YouthBirthday`:

1. The parent's `greeting()` method.
   - ♦ This will be included by inheritance.
2. A method with the name of the sender as a parameter.
   - ♦ This will be an additional method it will not override the parent's method.
   - ♦ In addition to what the parent's method does, this method will write out "How you have grown!! Love," followed by the name of the sender.

Here is a skeleton:

```
// Revised version
//
class YouthBirthday  extends Birthday

{
  public  YouthBirthday ( String r, int years )
  {
    super ( r, years )
  }

  // additional method−−−does not override parent's method
  public void greeting( ____________  _____________ )
  {
    super.greeting();
    System.out.println("How you have grown!!\n");
    System.out.println("Love, " + ____________ + "\n" );
  }
}
```

QUESTION 11:

Fill in the missing parts.

A good answer might be:

The revised class is seen below.

Choice of Methods

Now objects of the `YouthBirthday` class have two methods: `greeting()`, and `greeting( String )`.

```
// Revised version
//
class YouthBirthday  extends Birthday

{
  public  YouthBirthday ( String r, int years )
  {
    super ( r, years )
  }

  // additional method---does not override parent's method
  public void greeting( String  sender )
  {
    super.greeting();
    System.out.println("How you have grown!!\n");
    System.out.println("Love, " + sender +"\n" );
  }
}
```

Here is a `YouthBirthday` object using each of its two methods:

```
YouthBirthday yBday = new YouthBirthday( "Henry", 12 );
yBday.greeting();
yBday.greeting( "Alice" );
```

QUESTION 12:

What will this program fragment write out?

A good answer might be:

```
Dear Henry,
Happy 12th Birthday

Dear Henry,
Happy 12th Birthday
How you have grown!!
Love, Alice
```

Top of the Hierarchy

The "top" of the hierarchy of cards is the class `Card`. Sometimes this is called the `root` of the hierarchy. A variable of the root type, `Card`, can be used with any of its descendants:

```
Card crd      = new YouthBirthday( "Valerie", 7 );
crd.greeting();             // print a youth birthday greeting

crd           = new AdultBirthday( "Walter", 47 );
crd.greeting();             // print an adult birthday greeting

crd           = new Birthday( "Zoe", 30 );
crd.greeting();             // print a generic birthday greeting

Holiday   hol = new Holiday( "Kelly" );
Valentine val = new Valentine( "Jill", 42 );

crd           = hol;
crd.greeting();             // print a holiday greeting

crd           = val;
crd.greeting();             // print a valentine greeting
```

When a program uses several related objects, it is often useful to have a reference variable than can be used with any of them.

QUESTION 13:

(Thought Question:) Say that you had a collection of 12 greeting card objects and wanted to keep them in an array. What type of array should you use?

A good answer might be:

An array of `Card`:

```
Card[] cards = new Card[12];
```

Array of Related Objects

Now any object that belongs in the hierarchy can fit into any slot of the array. This is like the greeting card display at the drug store that holds a different selection of cards in different seasons of the year. Here is a program snippet:

```
Card[] cards = new Card[12];

cards[0]     = new YouthBirthday( "Valerie", 7 );
cards[1]     = new AdultBirthday( "Walter", 47 );
cards[2]     = new Birthday( "Zoe", 30 );
cards[3]     = new Holiday( "Kelly" );
cards[4]     = new Valentine( "Jill", 42 );

for ( int j=0; j <= 4; j++ )
  cards[j].greeting();
```

This code will create 5 cards objects of various varieties and put them into the array. Then it will ask each object in the array to write out its own version of the greeting. This will work fine:

```
Dear Valerie,
Happy 7th Birthday
How you have grown!!

Dear Walter,
Happy 47th Birthday
You haven't changed at all!

Dear Zoe,
Happy 30th Birthday

Dear Kelly,
Season's Greetings!

Dear Jill,
Love and Kisses,
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

(I've eliminated some blank lines from the actual output.)

QUESTION 14:

Now say that this code follows the above:

```
Card temp;

temp     = cards[0];
cards[0] = cards[1];
cards[1] = temp;
```

- Is this code correct?
- What did it do?

A good answer might be:

- Is this code correct?
    - ♦ Yes.
- What did it do?
    - ♦ It exchanged the objects in the first two slots of the array.

Review of the `Object` class

This sort of activity is very common in programming. You will see it a lot in the future. Notice that no new objects were created; but different reference variables were used for already−existing objects.

In Java, there is a pre−defined class that is the ultimate ancestor of all other classes. This class is called `Object`. When you define a class of your own like this:

```
class MyNewClass
{

}
```

You are actually defining a child of the `Object` class. The following is the exact equivalent of the above:

```
class MyNewClass extends Object
{

}
```

Every class in Java your own and every class in every library descends from `Object`. This will be important to remember when you get to graphical user interfaces.

QUESTION 15:

- Is a `String` an `Object`?
- Is a `Card` an `Object`?
- Is a `Applet` an `Object`?

A good answer might be:

- Is a `String` an `Object`?
  - ♦ Yes
- Is a `Card` an `Object`?
  - ♦ Yes
- Is a `Applet` an `Object`?
  - ♦ Yes

Arrays of Object

Since every class is a descendant of `Object`, an `Object` reference variable can be used with an object of any class. For example:

```
Object obj;

String        str = "Yertle" ;
Double        dbl = new Double( 32.0 );
YouthBirthday ybd = new YouthBirthday( "Ian", 4 );

obj = str;
obj = dbl;
obj = ybd;
```

Each of the last three statements is correct (although in this program, useless.) It would not be correct if this followed the above statements:

```
obj.greeting();
```

It is true that `obj` refers to a `YouthBirthday` object, which has a `greeting()` method. But the compiler needs to be told this with a **type cast**. The following is OK:

```
((YouthBirthday)obj).greeting();
```

A typecast is used to tell the compiler what is "really" in a variable that itself is not specific enough.

QUESTION 16:

Is the following OK?

```
Object obj;
String str = "Yertle" ;

obj = str;
((YouthBirthday)obj).greeting();
```

Is the following OK?

```
Object obj;
String        str = "Yertle" ;

obj = str;
((YouthBirthday)obj).greeting();
```

A good answer might be:

No.

The `instanceof` Operator

A typecast is used to tell the compiler what is "really" in a variable that itself is not specific enough. You have to tell the truth. In a complicated program, a reference variable might end up with any of several different objects, depending on the input data or other unpredictable conditions. To deal with this, the `instanceof` operator is used.

```
variable instanceof Class
```

This operator evaluates to `true` or `false` depending on whether the variable refers to an object of type `Class.` For example in the following fragment, `instanceof` is used to ensure that the object referenced by `obj` is used correctly:

```
Object obj;
YouthBirthday ybd = new YouthBirthday( "Ian", 4 );
String        str = "Yertle" ;

obj = ybd;

if ( obj instanceof YouthBirthday )
  ((YouthBirthday)obj).greeting();

if ( obj instanceof String )
  System.out.print( (String)obj );
```

`instanceof` will also return `true` if the class of the object on the left is a child (or grandchild or greatgrandchild or ...) of the class on the right.

QUESTION 17:

If you don't know in advance what classes a variable might hold, will `instanceof` help?

A good answer might be:

No. You can use it to test for several possible classes, but only if you know what the possibilities are.

End of the Chapter

You have reached the end of the chapter. You may wish to review the following. Click on a blue subject that interests you to go to where it was discussed. To get back here, click on the "back arrow" button of your browser.

- The signature of a method.
- Overriding parent methods.
- Using `super` in a child method.
- The root of a hierarchy.
- Arrays of related objects.
- The Object class.
- Using a type cast with a variable.
- The `instanceof` operator.

The next chapter will discuss `interfaces` yet another aspect of Java you will need to know to do graphical user interface programming.

Back to the main menu. You have reached the end of the chapter. creation: 3/6/99, revised: 5/7/99, 01/25/00, 01/26/02, 07/22/02

CHAPTER 53 Interfaces

Java has single inheritance, only. This means that a child class inherits from only one parent class. Mostly this is all you need. But sometimes multiple inheritance would be convenient. *Interfaces* give Java some of the advantages of multiple inheritance without the disadvantages.

Chapter Topics:

- *Interfaces.*
- *Defining an Interface.*
- *How a class implements an interface.*
- *Using an interface as a data type.*
- *Public interfaces.*
- *Implementing multiple interfaces.*
- *Extending an interface.*

The chapter includes some small examples, but you will have to wait until you start programming graphical user interfaces to see interfaces used realistically. Interfaces are crucial to GUI programming.

QUESTION 1:

Is your car a vehicle or a taxable property?

A good answer might be:

Both. Usually objects in the world can be regarded in several ways.

Interface

Object oriented programming makes software behave like "real world" objects. This makes programs easier to think about and more reliable. In the real world, you often think about an object in several different ways. You can think of your car as a vehicle or as taxable property. It would be convenient if software objects, also, could be thought of in several ways. But a Java object belongs to just one class.

In Java, an **interface** is used to express an aspect of a class other than what it inherits from its parent.

> An **interface** is a collection of constants and method declarations. The method declarations do not include an implementation (there is no method body.)

A child class that *extends* a parent class can also *implement* an interface to gain some additional behavior. Here is what an interface definition looks like:

```
interface InterfaceName
{
  constant definitions
  method declarations (without implementations.)
}
```

A *method declaration* is simply an access modifier, return type, and method signature followed by a semicolon.

This looks somewhat like a class definition. But no objects can be constructed from it. Objects can be constructed from a class that *implements* an interface. A class definition implements an interface like this:

```
class SomeClass extends SomeParent implements interfaceName
{

}
```

A class always extends just one parent but may implement several interfaces.

QUESTION 2:

(Review:) If a class definition does not say *extends SomeClass*, what class does it extend?

A good answer might be:

Object

Example interface

Here is an example interface definition:

```
interface MyInterface
{
  public final int aConstant = 32;        // a constant
  public final double pi = 3.14159;       // a constant

  public void methodA( int x );           // a method declaration
  double methodB();                       // a method declaration
}
```

The constants don't have to be separated from the methods, but doing so makes the interface easier to read. A method in an interface cannot be made `private`. A method in an intervace is `public` by default. In the example `methodB` is `public` even though it does not say so.

- A class that *implements* an interface must implement each method in the interface.
- Each method must be *public* (this happens by default).
- Constants from the interface can be used as if they had been defined in the class.
- Constants should not be redefined in the class.

QUESTION 3:


```
interface SomeInterface
{
  public final int x = 32;
  public double y;

  public double addup( );
}
```

Inspect the interface. Is it correct? Click here for a

A good answer might be:

No. Variables (such as `y`) cannot be put in an interface. Only constants and methods.

```
interface SomeInterface
{
  public final int x = 32;
  public double y;

  public double addup( );
}
```

Implementing an Interface

A class definition must always extend one parent, but it can *implement* zero or more interfaces:

```
class SomeClass extends Parent implements SomeInterface
{

   ordinary class definition body

}
```

The body of the class definition is the same as always. However, since it implements an interface the body must have a definition of each of the methods from the interface. The class definition can use access modifiers as usual. Here is a class definition that implements three interfaces:

```
public class BigClass extends Parent
    implements  InterfaceA, InterfaceB, InterfaceC
{

   ordinary class definition body

}
```

Now `BigClass` must provide a method definition for every method in each of the interfaces. Any number of classes can implement the same interfaces. Here is another class definition:

```
public class SmallClass implements InterfaceA
{

   ordinary class definition body

}
```

QUESTION 4:

Is the above class definition correct?

A good answer might be:

Yes. You might wonder that it does not extend a base class, but it does. If no other class is extended, `Object` is the base class. `SmallClass` extends `Object` and implements InterfaceA.

Example Problem

Let us create a database program for a store. The store sells:

- `Goods`, each of which has the attributes:
  - ♦ description
  - ♦ price
- The goods are either:
  - ♦ Food with an attribute "calories"
  - ♦ Toy with an attribute "minimumAge", or
  - ♦ Book with an attribute "author."

Of these goods, toys and books are taxable, but food is not. There are many other things that are taxable, such as services or entertainment so we want to have the concept "taxable" as a separate concept, not part of the concept of `Goods`.

Here is what the concept `Taxable` looks like:

- A `Taxable` item,
  - ♦ has a `taxRate` of 6 percent,
  - ♦ has a `calculateTax()` method.

When implemented in Java, these concepts will appear as classes and an interface.

QUESTION 5:

(Design question:) Decide on an implementation for each concept:

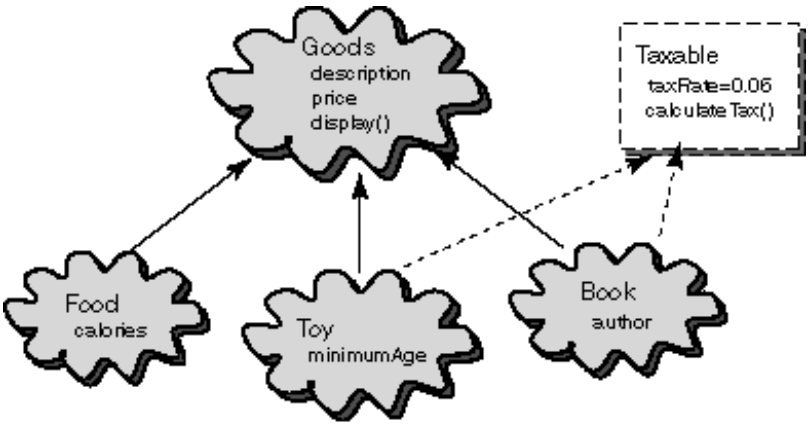
Goods	
Food	
Toy	
Book	
Taxable	

A good answer might be:

Goods	Parent Class
Food	Child Class

Toy	Child Class
Book	Child Class
Taxable	Interface

Picture of the Classes



Food, Toy, and Book extend their parent, Goods

Toy and Book implement the interface Taxable

Here is a picture that shows the classes and the interface. The children classes **extend** their parent. This is shown with a solid arrow pointing at the parent.

The solid arrows show *inheritance*. The three children inherit the `display()` method.

Two of the classes **implement** the interface. This is shown with a dotted arrow pointing at the interface.

The dotted arrow show what a class *must implement*. The `Toy` and `Book` classes must implement the `calculateTax()` method.

QUESTION 6:

Next

According to the picture,

- What method(s) will `Food` have?
- What method(s) will `Toy` have?

A good answer might be:

- What method(s) will `Food` have?
  - ♦ `display()`
- What method(s) will `Toy` have?
  - ♦ `display()` and `calculateTax()`

Starting the Program

Here is a class definition for `Goods`:

```
class Goods
{
  String description;
  double price;

  Goods( String des, double pr )
  {
    description = des;
    price       = pr;
  }

  void display()
  {
    System.out.println( "item: " + description +
        " price: " + price );
  }
}
```

Here is a skeleton of the class `Food`. Recall that it adds the variable `calories`.

```
class Food _____________ _____________
{
  double calories;

  Food( String des, double pr, double cal)
  {
    super( ______, ______ );
    calories = _____________ ;
  }

  void display()
  {
    super._____________ ;
    System.out.println( "calories: " + calories );
  }
}
```

QUESTION 7:

Fill in the blanks. Click here for a

A good answer might be:

The blanks are filled in, below.

Adding the Interface

The child class `Food` extends the parent class. It uses `super` to use the parent's constructor and the parent's display method.

```
class Food extends Goods
{
  double calories;

  Food( String des, double pr, double cal)
  {
    super( des, pr );
    calories = cal ;
  }

  void display()
  {
    super.display( );
    System.out.println( "calories: " + calories );
  }
}
```

Here is `Taxable`:

- A `Taxable` item,
  - ♦ has a `taxRate` of 6 percent, which should be a `double` constant.
  - ♦ and a `calculateTax()` method. which should return a `double` value.

The `Taxable` interface looks like this:

```
interface Taxable
{
  final double _____ = ______ ;
  double ___________() ;
}
```

The `final` says that what follows is a constant, not a variable (variables are not allowed in interfaces.) In fact, the `final` can be omitted since the identifier that follows will automatically be a constant. The " = value " cannot be omitted.

The method declaration (in the second line) is `public` by default.

QUESTION 8:

Fill in the blanks.

A good answer might be:

```
interface Taxable
{
  final double taxRate = 0.06 ;
  double calculateTax() ;
}
```

Adding another Class

Since `taxRate` is a constant, it must be set to a value, here, 6 percent. Here is a partial definition of `Toy`. Recall that it:

- Has a parent `Goods`.
- Adds a variable `minimumAge`.
- Is taxable.

```
class Toy extends Goods _____________ _____________
{
  int minimumAge;

  Toy( String des, double pr, int min)
  {
    super( des, pr );
    minimumAge  = min ;
  }

  void display()
  {
    super.display() ;
    System.out.println( "minimum age: " + minimumAge );
  }

  public double _____________()  // implementing the interface
  {
    return price * _____________ ;
  }
}
```

QUESTION 9:

Fill in the blanks. Click here for a

A good answer might be:

The class is completed below.

Adding the Remaining Class

The constant `taxRate` is used in the `calculateTax()` method just as if it had been defined in the `Toy` class. Also, it can be used in methods of the class other than those listed in the interface.

```
class Toy extends Goods implements Taxable
{
  int minimumAge;

  Toy( String des, double pr, int min)
  {
    super( des, pr );
    minimumAge  = min ;
  }

  void display()
  {
    super.display() ;
    System.out.println( "minimum age: " + minimumAge );
  }

  public double calculateTax()
  {
    return price * taxRate ;
  }
}
```

The `calculateTax()` method must be made public.

QUESTION 10:

Can another class implement the `Taxable` interface?

A good answer might be:

Yes. Any number of classes can implement the same interface.

The Completed Book Class

There is one remaining class in our example, `Book`, which looks like this:

- Has a parent `Goods`.
- Adds a variable `author`.
- Is taxable.

You might wish to review the diagram that shows the relationships between the classes. Here is the usual blank−ridden class definition:

```
class Book  _____________ Goods _____________ Taxable
{
  String _____________;

  Book( String des, double pr, String auth)
  {
    super( des, pr );
    ____________  = auth ;
  }

  void display()
  {
    super.display() ;
    System.out.println( "author: " + _______________ );
  }

  public double _____________()
  {
    return price * _____________ ;
  }
}
```

You might wish to consult the Taxable interface.

QUESTION 11:

Next

Rid the definitions of those nasty blanks.

A good answer might be:

See below.

The Completed Book Class

```
class Book  extends Goods implements Taxable
{
  String author;

  Book( String des, double pr, String auth)
  {
    super( des, pr );
    author  = auth ;
  }

  void display()
  {
    super.display() ;
    System.out.println( "author: " + author );
  }

  public double calculateTax()
  {
    return price * taxRate ;
  }
}
```

QUESTION 12:

Is the `taxRate` for `Book` the same as for `Toy`?

A good answer might be:

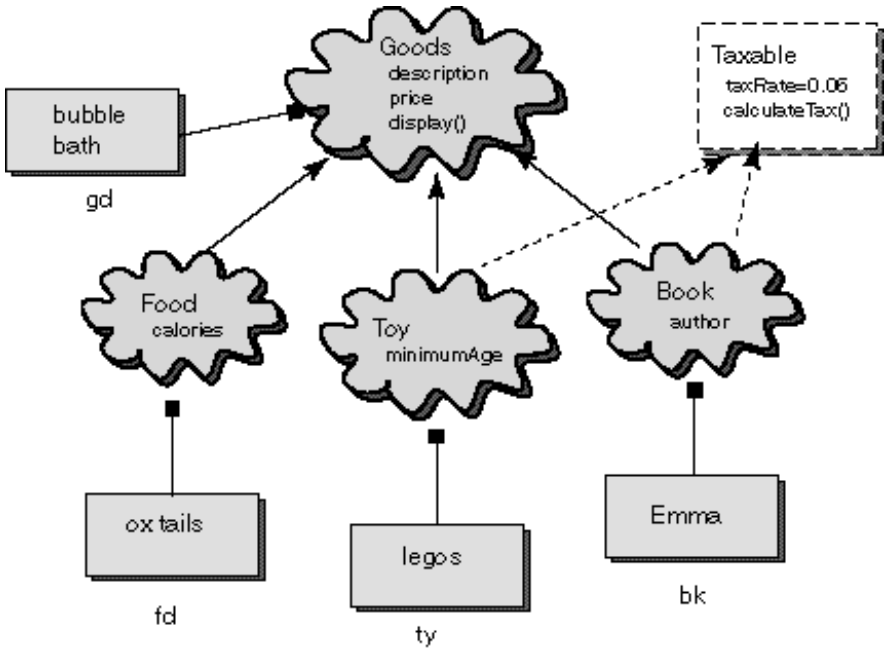
Yes. By using an interface, a constant can be used by several classes. This helps keep the classes consistent.

Testing Program

Here is a tiny program that tests the classes. If you want to run the program, copy and paste all the classes and the interface to a file called `Store.java`.

```
public class Store
{

  public static void main ( String[] args )
  {
    Goods gd = new Goods( "bubble bath", 1.40 );
    Food  fd = new Food ( "ox tails", 4.45, 1500 );
    Book  bk = new Book ( "Emma", 24.95, "Austin" );
    Toy   ty = new Toy  ( "Legos", 54.45, 8 );

    gd.display();

    fd.display();

    ty.display();
    System.out.println("Tax is: " + ty.calculateTax() + "\n" );

    bk.display();
    System.out.println("Tax is: " + bk.calculateTax() + "\n" );

  }
}
```

The `calculateTax()` method is only used with objects whose class implements the interface Here is a picture that shows the classes and their objects:



In the picture, clouds represent classes. Arrows with pointed head connect child classes to parent classes. The dotted rectangle represents the interface; a dotted arrow shows which classes implement it. Rectangles represent objects. Arrows with square head connect an object to its class.

QUESTION 13:

- Could these 4 objects be kept in an array?
- What would the array type be?

A good answer might be:

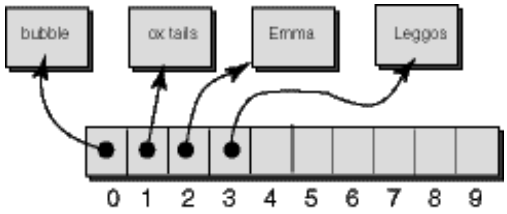
- Could these 4 objects be kept in an array?
    - ♦ Yes.
- What would the array type be?
    - ♦ Goods[]

Array of Goods Objects

Here is a modified testing program that uses an array:

```
public class Store
{

  public static void main ( String[] args )
  {
    Goods[] inventory =  new Goods[10];
    inventory[0] = new Goods( "bubble bath", 1.40 );
    inventory[1] = new Food ( "ox tails", 4.45, 1500 );
    inventory[2] = new Book ( "Emma", 24.95, "Austin" );
    inventory[3] = new Toy  ( "Leggos", 54.45, 8 );

    inventory[0].display();
    inventory[1].display();
    inventory[2].display();
    inventory[3].display();
  }
}
```

Since each child class is−a `Goods`, an array of type `Goods[]` can be used with any of them. The array `inventory` has 10 slots, but the program uses only 4 of them, like this:



As always, each slot of the array is a reference variable which can refer to an object that has been created with a `new`.

QUESTION 14:

Next

Is each of the classes `Toy` and `Book` taxable?

A good answer might be:

Yes.

Interface as a Type

An interface can be used as a data type for a reference variable. Since `Toy` and `Book` implement `Taxable`, they can both be used with a reference variable of type `Taxable`:

```
  public static void main ( String[] args )
  {
    Taxable item1 = new Book ( "Emma", 24.95, "Austin" );
    Taxable item2 = new Toy  ( "Leggos", 54.45, 8 );

    System.out.println( "Tax on item 1 "+ item1.calculateTax() );
    System.out.println( "Tax on item 2 "+ item2.calculateTax() );

  }
```

The compiler has been told in the interface that all `Taxable` objects will have a `calculateTax()` method, so that method can be used with the variables.

QUESTION 15:

Would the following work?

```
item1.display() ;
```

Would the following work?

```
item1.display() ;
```

A good answer might be:

No. All the compiler has been told is that `item1` implements the methods in the interface `Taxable`. The `display()` method is not in the interface.

Type Casts

When you use a variable of type `Taxable` you are asking to use the "taxable" aspect of the object. Many different kinds of objects might be referred to by the variable. (In a larger program there may be Taxable classes that are not `Goods`.) The compiler can only use the methods it knows that the object must have those in the interface

However, you can use a `type cast` to tell the compiler that in a particular statement in the program the variable will refer to an object of a specific class:

```
public static void main ( String[] args )
{
  Taxable item1 = new Book ( "Emma", 24.95, "Austin" );
  System.out.println( "Tax on item 1 "+ item1.calculateTax() );

  ((Book)item1).display();
}
```

This program is not very sensibly written, since if the variable `item1` were of type `Book` everything would work without the need for a type cast. But in programs with more complicated logic such casts are sometimes needed.

QUESTION 16:

Why are the red parentheses needed in:

```
((Book)item1).display();
```

```
((Book)item1).display();
```

A good answer might be:

Because you want to:

- First, cast `item1` to type `Book`.
- Then, invoke the `display()` method that objects of type `Book` have.

When is a Type Cast Needed?

Now examine the following:

```
public static void main ( String[] args )
{
  Book    book ;
  Taxable tax = new Book ( "Emma", 24.95, "Austin" );

  book = tax;
  book.display();
  System.out.println( "Tax on item 1 "+ book.calculateTax() );
}
```

QUESTION 17:

Is a type cast necessary? Where should it go?

A good answer might be:

A type cast is necessary to tell the compiler that the variable `tax` in this case contains a `Book` object:

```
public static void main ( String[] args )
{
  Book    book ;
  Taxable tax = new Book ( "Emma", 24.95, "Austin" );

  book = (Book)tax;
  book.display();
  System.out.println( "Tax on item 1 "+ book.calculateTax() );
}
```

More Practice

Now Consider the following code:

```
public static void main ( String[] args )
{
  Goods   toy ;
  Taxable tax = new Toy ( "Grobot", 1.49, 6 );

  toy = tax;
  toy.display();
  System.out.println( "Tax: "+ toy.calculateTax() );
}
```

QUESTION 18:

Are type casts necessary? Where should they go? Hint: there are several correct answers.

A good answer might be:

Here is one correct answer:

```
public static void main ( String[] args )
{
  Goods   toy ;
  Taxable tax = new Toy ( "Grobot", 1.49, 6 );

  toy = (Toy)tax;
  toy.display();
  System.out.println( "Tax: "+ ((Taxable)toy).calculateTax() );
}
```

The first type cast must cast `tax` to a type `Goods` (or an ancestor). The second must cast `toy` to `Taxable` or to a class that implements `Taxable`.

Yet More Practice

Here is another possible answer:

```
public static void main ( String[] args )
{
  Goods   toy ;
  Taxable tax = new Toy ( "Grobot", 1.49, 6 );

  toy = (Goods)tax;
  toy.display();
  System.out.println( "Tax: "+ ((Toy)toy).calculateTax() );
}
```

QUESTION 19:

Is this answer correct?

A good answer might be:

Yes.

Additional Facts about Interfaces

There are features of interfaces that the example did not show: A class can implement several interfaces:

```
class SomeClass extends SomeParent
    implements InterfaceA, InterfaceB, InterfaceC
{

}
```

Now `SomeClass` must implement all the methods listed in all the interfaces.

QUESTION 20:

Could the interfaces contain several definitions of the same constant?

Could the interfaces contain several definitions of the same constant?

A good answer might be:

No. To ensure consistancy, a constant should be defined only once, in one interface.

Public Interfaces

However, it is OK if two interfaces ask for the same method. A class that implements both interfaces only needs to provide one complete method definition to satisfy both interfaces.

An interface can be made `public`. In fact, this is usually what is done. When a class or interface is `public` it must be the only public class or interface in the file that contains it. (These notes have been avoiding public classes so that the "copy paste save and run" method can be used with just one file.)

A public interface can be implemented by any class in any file. Many graphical user interface components implement public interfaces. You must use them to work with the GUI features of Java.

QUESTION 21:

Can an interface be made `private`?

A good answer might be:

No. `private` would mean that nobody could use it, which is not sensible.

Hierarchy of Interfaces

An interface can be an extension of another interface (but **not** an extension of a class):

```
public interface ExciseTaxable extends Taxable
{
  double extraTax = 0.02;
  double calculateExtra();
}
```

A complex hierarchy of interfaces can be constructed using this feature. This is an advanced feature which you will probably not need to use.

QUESTION 22:

Can a class extend an interface?

Can a class extend an interface?

A good answer might be:

No. A class always extends just one class.

End of the Chapter

You have reached the end of the chapter. You may wish to extend your knowledge by examining the following. Click on a blue subject that interests you to go to where it was discussed. To get back here, click on the "back arrow" button of your browser.

- What an interface is.
- How a class implements an interface.
- Implementing several interfaces.
- An example of a constant in an interface.
- Using an interface as a type of a reference variable.
- Use of a type cast with a reference variable.
- Public interfaces.
- Extending an interface.

The next chapter will discuss the `Vector` class and the `Enumeration` interface.

Back to the main menu.

You have reached the end of the chapter. created 01/01/00; revised: 01/25/00, 07/22/02

CHAPTER 54 Vectors and Enumerations

It is very common for a program to manipulate data that is kept in a list. You have already seen how this is done using arrays. Arrays are a fundamental feature of Java and most programming languages. But because lists are so useful, the Java Development Kit includes the `Vector` class, which works much like an array but has additional methods and features.

Like an array, a `Vector` contains elements that are accessed using an integer index. However, unlike an array, the size of a Vector will expand if needed as items are added to it.

Chapter Topics:

- *Review of fixed−length arrays.*
- *The `Vector` class.*
- *`Vector` constructors.*
- *capacity and size.*
- `addElement(), setElementAt(), insertElementAt()`
- `removeElementAt(), firstElement(), lastElement()`

QUESTION 1:

(Review:) What is the *length* of the array referenced by `list`?

```
String[] list = new String[5];
```

```
String[] list = new String[5];
```

A good answer might be:

The array object has five slots.

Fixed−length Array

Once an array object has been constructed, the number of slots will not change. If an array object is to be used for the entire run of a program, its length must be large enough to accommodate all expected data. It is often hard to guess the correct length in advance. It would be nice if there were a kind of array where you could keep adding data no matter what length it was originally.

QUESTION 2:

Say that an array has been declared and completely filled with information:

```
String[] list = new String[3];
list[0] = "ant" ;
list[1] = "bat" ;
list[2] = "cat" ;
```

After running for a while, the program needs to add information to the array. Will the following statement make the array larger?

```
list = new String[5];
```

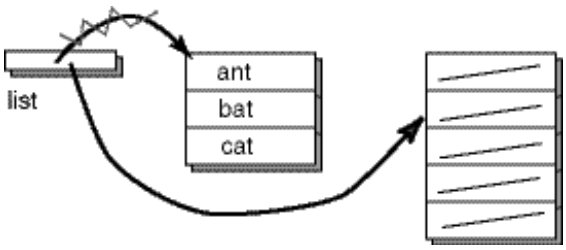
Will the following statement make the array larger?

```
list = new String[5];
```

A good answer might be:

No. The statement creates a *completely new* array object (with room for 5 slots). All the information in the old array object is now garbage.

Old Information is Lost



In the example, the reference variable *list* is set to the newly constructed object. The information in the old object is not automatically transferred. To keep the old information, the program would have to copy it to a new array after it was constructed. This is not too hard, but it is a nuisance.

In the diagram, the diagonal slashes represent **null**, the value a reference has when it refers to no object.

QUESTION 3:

Next

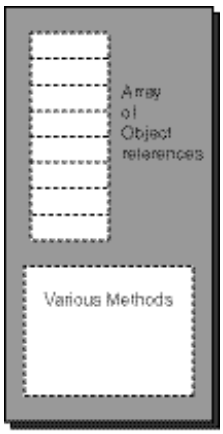
What happens if information is added to an array slot that does not exist?

```
String[] list = new String[3];
list[0] = "ant" ;
list[1] = "bat" ;
list[2] = "cat" ;

list[3] = "dog" ;

String[] list = new String[3];
list[3] = "dog" ;
```

A good answer might be:

An `ArrayIndexOutOfBounds` exception is thrown and (ordinarily) the program halts.

`Vector` class



A Vector Object

The `Vector` class builds upon the capabilities of arrays. A `Vector` object contains an array of object references plus many methods for managing that array. The biggest convenience of a `Vector` is that you can keep adding elements to it no matter what size it was originally. The size of the `Vector` will automatically increase and no information will be lost.

However, this convenience comes at a price:

1. The elements of a `Vector` are *object references*, not primitive data like `int` or `double`.
2. Using a `Vector` is slighly slower than using an array directly.
3. The elements of a `Vector` are references to `Object`. This means that often you will need to use type casting with the data from a `Vector`.

QUESTION 4:

Next

(Review:) What is the `Object` class?

A good answer might be:

`Object` is the class that all other classes have as an ancestor.

Constructors for `Vector` Objects

To declare a reference variable for a `Vector`, do this:

```
Vector myVector;        // myVector is a reference to a future Vector object
```

You do not say what type of object you are intending to store. A `Vector` is like an array of references to `Object`. This means that any object reference can be stored in a `Vector`. To declare a variable and to construct a `Vector` with an unspecified initial capacity do this:

```
Vector myVector = new Vector();   // myVector is a reference to a Vector object.
                                  // The Java system picks the initial capacity.
```

This may not be very efficient. If you have an idea of what the capacity you need, start your `Vector` with that capacity. To declare a variable and to construct a `Vector` with an initial capacity of 15 do this:

```
Vector myVector = new Vector(15);   // myVector is a reference to a Vector object
                                    // with an initial capacity of 15 elements.
```

The initial capacity is the size that the `Vector` starts with. It can expand beyond this capacity if you add more elements. Expanding the capacity of a `Vector` is slow; and will slow down your program if it happens too often. For more control, specify how many new slots will be added each time the `Vector` expands:

```
Vector myVector = new Vector(15, 5);   // myVector is a reference to a Vector object
                                       // with an initial capacity of 15 elements,
                                       // and an increment size of 5.
```

Now when additional capacity is needed, 5 slots at a time will be added. You don't have to fill the new slots, but now they are there if you need them.

QUESTION 5:

You are writing a program to keep track of the students in a class. There are usually about 12 students in a class, but often a few students add in during the first few weeks. Declare and construct a `Vector` suitable for this situation.

```
Vector csClass = new Vector( __________, __________ ) ;
```

A good answer might be:

```
Vector csClass = new Vector( 15, 5 ) ;
```

Rough estimates are good enough. It is better to overestimate the capacity slightly than to underestimate it. A few unused slots do not hurt.

Capacity and Size

A `Vector` object has a `capacity` and a `size`. To find out the current capacity of a `Vector` use its `capacity()` method. To find out the current size of a `Vector` use its `size()` method.

- The `capacity` is the number of slots available.
- The `size` is the number of slots that have data in them.
- Slots 0 up through size−1 have data in them.
- You cannot skip slots; before slot N gets data, slots, 0, 1, 2, ... N−1 must be filled with data.

The elements of a `Vector` are accessed using an integer index. As with arrays, the index is an integer value that starts at 0.

- For retrieving data from a `Vector` the index is 0 to size−1.
- For setting data in a `Vector` the index is 0 to size−1.
- For inserting data into a `Vector` the index is 0 to size.
    - ♦ When you insert data at index `size`, you are adding data to the end of the `Vector`.

(These rules will be explained in detail in a few pages.)

QUESTION 6:

What is the value of the following boolean expression?

```
myVector.size() <= myVector.capacity()
```

A good answer might be:

```
myVector.size() <= myVector.capacity()
```

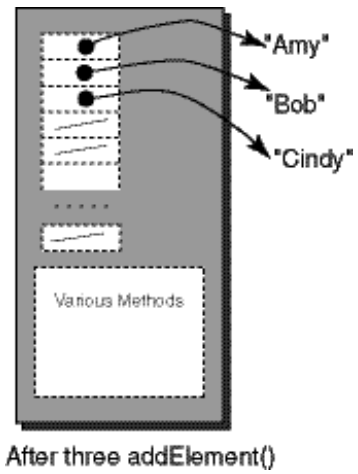
will always be true.

Adding Elements to a `Vector`

To add an element to the end of a `Vector` use:

```
addElement( Object ) ;  // add a reference to an Object to the end of the Vector,
                        // increasing its size by one.  The capacity of the Vector
                        // will increase by increment if needed.
```

Here is an example program. To use the `Vector` you must import the `java.util` package:



After three addElement()

```
import java.util.* ;

class VectorEg
{

  public static void main ( String[] args)
  {
    Vector names = new Vector( 20, 5 );

    System.out.println("capacity: "
        + names.capacity() );
    System.out.println("size: "
        + names.size() );

    names.addElement("Amy");
    names.addElement("Bob");
    names.addElement("Cindy");

    System.out.println("capacity: "
        + names.capacity() );
    System.out.println("size: "
        + names.size() );

  }
}
```

The initial capacity of the `Vector` is set to 20 with an increment of 5. References to three `String` objects are added. (The `String` objects are constructed by enclosing characters in quote marks.)

As the diagram shows, the added objects are not part of the `Vector`. The `Vector` contains an array of references to the objects added to it.

QUESTION 7:

What does the program print out?


```
capacity:
size:

capacity:
size:
```

A good answer might be:

```
capacity: 20
size: 0
```

```
capacity: 20
size: 3
```

Setting and Accessing `Vector` Elements

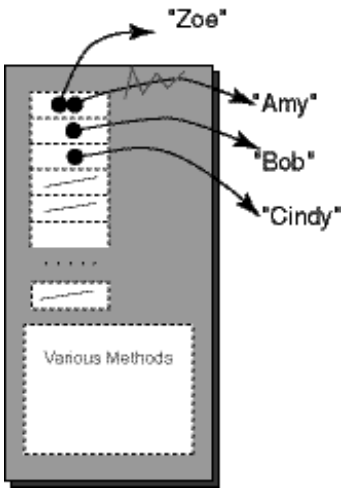
You just saw that the `addElement()` method adds to the end of a `Vector`. Sometimes you want to set the data at a particular index.

```
setElementAt( objectReference, index )
```

The `index` should be within 0 to size−1. The data previously at `index` is replaced with *objectReference*. To access the object at a particular index use:

```
elementAt( index )
```

The `index` should be 0 to size−1. Here is the example program:




```
import java.util.* ;

class VectorEg
{

  public static void main ( String[] args)
  {
    Vector names = new Vector( 20, 5 );

    names.addElement("Amy");
    names.addElement("Bob");
    names.addElement("Cindy");

    System.out.println("slot 0: "
        + names.elementAt(0) );
    System.out.println("slot 1: "
        + names.elementAt(1) + "\n");

    names.setElementAt( "Zoe", 0 );
    System.out.println("slot 0: "
        + names.elementAt(0) );
    System.out.println("slot 1: "
        + names.elementAt(1) );

  }
}
```

QUESTION 8:

What does the program print out?

Next

```
slot 0:
slot 1:

slot 0:
slot 1:
```

A good answer might be:

```
slot 0: Amy
slot 1: Bob

slot 0: Zoe
slot 1: Bob
```

Printing all the Elements in a `Vector`

A `Vector` is a linear list of elements, just like an array. As with an array a counting loop can be used to visit each element in order. (**Note:** an alternate method, using an `Enumeration` will be discussed later.) The following is an example:

```
import java.util.* ;

class VectorEg
{

  public static void main ( String[] args)
  {
    Vector names = new Vector( 10 );

    names.addElement( "Amy" );
    names.addElement( "Bob" );
    names.addElement( "Chris" );
    names.addElement( "Dan" );
    names.addElement( "Elaine" );
    names.addElement( "Fred" );

    for ( int j=0; j < names._________(); j++ )
      System.out.println( j + ": " + names._________(j) );

  }
}
```

QUESTION 9:

Fill in the blanks to complete the program.

A good answer might be:

The compete program is below.

Completed Program

The counting loop uses the `size()` method to ensure that every element is accessed. A `Vector` has data in slots 0 to `size()−1`.

```
import java.util.* ;

class VectorEg
{

  public static void main ( String[] args)
  {
    Vector names = new Vector( 10 );

    names.addElement( "Amy" );
    names.addElement( "Bob" );
    names.addElement( "Chris" );
    names.addElement( "Dan" );
    names.addElement( "Elaine" );
    names.addElement( "Fred" );

    for ( int j=0; j < names.size(); j++ )
      System.out.println( j + ": " + names.elementAt(j) );

  }
}
```

In most programs, the size of the `Vector` changes as the program executes. For bug−free code, use `size()` so that the loop always works.

QUESTION 10:

Examine the following program. What will it print?

```
import java.util.* ;
class VectorEg
{
  public static void main ( String[] args)
  {
    Vector names = new Vector( 10 );

    names.addElement( "Amy" );
    names.addElement( "Bob" );
    names.addElement( "Chris" );

    names.setElementAt( "Zoe", 0 );
    names.setElementAt( "Bart", 1 );

    for ( int j=0; j < names.size(); j++ )
      System.out.println( j + ": " + names.elementAt(j) );

  }
}
```

A good answer might be:

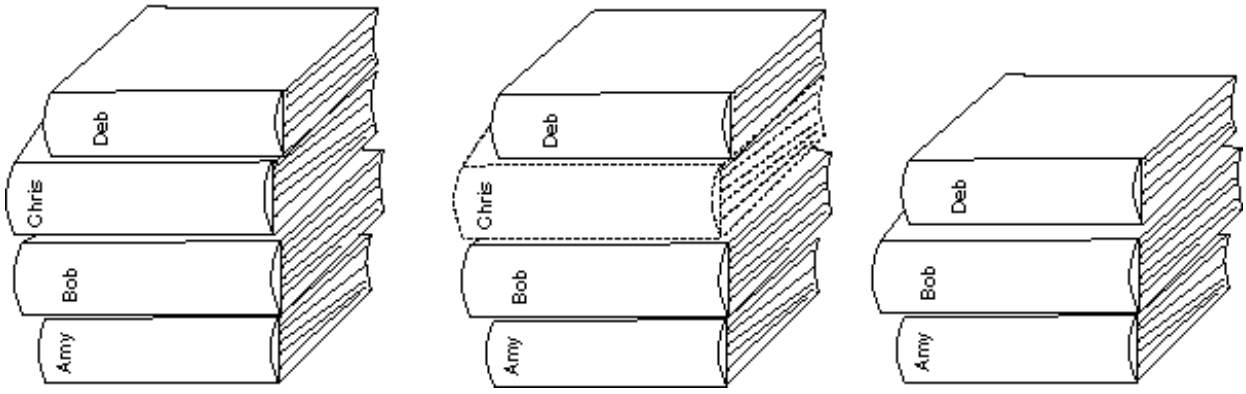
```
0: Zoe
1: Bart
2: Chris
```

Removing an Element

Frequently you wish to *remove* an element from a list. The `Vector` class has a method that will do this without leaving a hole in place of the deleted element:

```
removeElementAt(int index)  //  Deletes the element at index. Each element with an index
                            //  greater than index is shifted downward to have an index
                            //  one smaller than its previous value.
```

The element at location index will be eliminated. Elements at locations index+1, index+2, ... , size()−1 will each be moved down one to fill in the gap. This is like pulling out a book from the middle of a stack of books.



QUESTION 11:

Next

Examine the following program. What will it print?

```
import java.util.* ;
class VectorEg
{
  public static void main ( String[] args)
  {
    Vector names = new Vector( 10 );

    names.addElement( "Amy" );
    names.addElement( "Bob" );
    names.addElement( "Chris" );
    names.addElement( "Deb" );

    names.removeElementAt(2);

    for ( int j=0; j < names.size(); j++ )
      System.out.println( j + ": " +
          names.elementAt(j) );

  }
}
```

A good answer might be:

```
0: Amy
1: Bob
2: Deb
```

Inserting Elements

Sometimes you wish to *insert* an element into a `Vector` at a particular index. When an element is inserted at `index` the element previously at `index` is moved up to `index+1`, and so on until the element previously at `size()−1` is moved up to `size()`. The size of the `Vector` has now increased by one, and the capacity will increase if more room is needed.

```
insertElementAt(Object element, int index)  // Inserts the element at index.
                                            // Each element with an index
                                            // equal or greater than index is
                                            // shifted upward by one more than
                                            // its previous value.
```

Inserting is different from *setting* an element. When `setElementAt(objectReference, index )` is used, the object reference previously at `index` is replaced by the new `objectReference`. No other elements are effected, and the size does not change.

QUESTION 12:

Examine the following program. What will it print?

```
import java.util.* ;
class VectorEg
{
  public static void main ( String[] args)
  {
    Vector names = new Vector( 10 );

    names.addElement( "Amy" );
    names.addElement( "Bob" );
    names.addElement( "Chris" );
    names.addElement( "Deb" );

    names.insertElementAt( "Elaine", 2);

    for ( int j=0; j < names.size(); j++ )
      System.out.println( j + ": " + names.elementAt(j) );

  }
}

names.insertElementAt( "Elaine", 2);
```

A good answer might be:

```
0: Amy
1: Bob
2: Elaine
3: Chris
4: Deb
```

As requested, "Elaine" is inserted at position 2 and all elements previously at position 2 and higher are moved up one to make room.

`firstElement()` and `lastElement()`

The `firstElement()` and `lastElement()` methods are sometimes useful. They do what you imagine:

```
import java.util.* ;
class VectorEg
{
  public static void main ( String[] args)
  {
    Vector names = new Vector( 10 );

    names.addElement( "Amy" );    names.addElement( "Bob" );
    names.addElement( "Chris" );  names.addElement( "Deb" );

    System.out.println( names.firstElement() );
    System.out.println( names.lastElement() );

  }
}
```

The program prints out:

```
Amy
Deb
```

QUESTION 13:

If there are no elements in the `Vector` what do you think happens?

A good answer might be:

An exception is thrown, and (ordinarily) the program halts.

isEmpty()

The `elementAt()`, `removeElementAt()`, and other methods will also throw an exception when the designated element does not exist. To check if a `Vector` has elements use

```
boolean isEmpty()
```

which returns `true` if the `Vector` has no elements.

QUESTION 14:

(Review:) What algorithm is used to find the first occurrence of a particular element in an array?

A good answer might be:

Linear Search

Searching for an Element

Linear search is starting at the first element and examining them one by one until the target element is found. You could write linear search for a `Vector` (and it would be a good exercise to do so); but there is a method that does this for you:

```
int indexOf(Object elem)    //  Search for the first occurrence of
                            //  elem, testing for equality
                            //  using the equals method of elem.
```

The method returns the index of the first occurrence of `elem` or −1 if `elem` is not found.

QUESTION 15:

Examine the following program. What will it print?

```
import java.util.* ;
class VectorEg
{
  public static void main ( String[] args)
  {
    Vector names = new Vector( 10 );

    names.addElement( "Amy" );    names.addElement( "Bob" );
    names.addElement( "Chris" );  names.addElement( "Deb" );
    names.addElement( "Chris" );  names.addElement( "Joe" );

     System.out.println( names.indexOf( "Bob" ) );
     System.out.println( names.indexOf( "Elaine" ) );
  }
}
```

A good answer might be:

```
1
−1
```

Searching for the next Element

Often a `Vector` will have several of the same item. The `indexOf()` method finds only the first. Use the following method:

```
int indexOf(Object elem, int index)   //  Search for the first occurrence of
                                      //  elem starting at index
```

The method returns the index of the first occurrence of `elem` at or past the beginning index, or −1 if `elem` is not found.

The method is overloaded it has the same name as the previous method but one additional parameter.

QUESTION 16:

Examine the following program. What will it print?

```
import java.util.* ;
class VectorEg
{
  public static void main ( String[] args)
  {
    Vector names = new Vector( 10 );

    names.addElement( "Amy" );    names.addElement( "Bob" );
    names.addElement( "Chris" );  names.addElement( "Deb" );
    names.addElement( "Chris" );  names.addElement( "Joe" );

     System.out.println( names.indexOf( "Bob", 0 ) );
     System.out.println( names.indexOf( "Chris", 0 ) );
     System.out.println( names.indexOf( "Chris", names.indexOf("Chris")+1 ) );
     System.out.println( names.indexOf( "Elaine", 2 ) );
  }
}
```

A good answer might be:

```
1
2
4
−1
```

Enumeration

It is very common for a program to access the elements of a `Vector` one by one, in order. This can be done using a counting loop (as has been done so far this chapter); however, an object that implements the `Enumeration` interface may also be used. To get an `Enumeration` object for a `Vector` use this method:

```
elements()  // Returns an enumeration of the components of the vector.
```

Once you have an `enumeration` object, the `hasMoreElements()` and the `nextElement()` methods are used to move through the elements:

```
boolean hasMoreElements()   // return true if not all elements have been visited

Object nextElement()        // Returns the next element of the enumeration.
```

Here is a program that prints out every element in the `Vector`:

```
import java.util.* ;
class VectorEg
{
  public static void main ( String[] args)
  {
    Vector names = new Vector( 10 );

    names.addElement( "Amy" );    names.addElement( "Bob" );
    names.addElement( "Chris" );  names.addElement( "Deb" );
    names.addElement( "Elaine" ); names.addElement( "Frank" );
    names.addElement( "Gail" );   names.addElement( "Hal" );

    Enumeration enum = names._____________();

    while ( enum._____________() )
      System.out.println(enum._____________()  );

  }
}
```

Unfortunately, the program is full of blanks.

QUESTION 17:

Fill the blanks by choosing from the above methods.

A good answer might be:

The complted program is below.

Completed Program

If the `Vector` has no elements, then the enumeration will be empty, and `hasMoreElements()` will evaluate to *false*.

```
import java.util.* ;
class VectorEg
{
  public static void main ( String[] args)
  {
    Vector names = new Vector( 10 );

    names.addElement( "Amy" );    names.addElement( "Bob" );
    names.addElement( "Chris" );  names.addElement( "Deb" );
    names.addElement( "Elaine" ); names.addElement( "Frank" );
    names.addElement( "Gail" );   names.addElement( "Hal" );

    Enumeration enum = names.elements();

    while ( enum.hasMoreElements() )
      System.out.println(enum.nextElement());

  }
}
```

QUESTION 18:

(Review:) What is the type of each slot of a `Vector` object? (In other words, what is the contents of each slot?)

A good answer might be:

Each slot holds a reference to an `Object`.

Phone Book Application

```
class Entry
{
  String name;
  String number;

  // constructor
  Entry( String n, String num )
  {
    name = n; number = num;
  }

  // various methods
  . . .
}
```

`Vector`s are especially convenient when you want to maintain a list of your own object types.

Since all classes, even those of your own invention, descend from the `Object` class, a `Vector` can be used with objects of any type.

For example say that you wanted an application that maintained a phone book. Each entry in the phone book will contain a `name` and a `number`, along with various methods.

QUESTION 19:

Suggest two methods that will be useful for `Entry`.

A good answer might be:

It will be useful to have an `equals()` method which `indexOf()` will use, and it will be useful to override the `toString()` method.

Complete `Entry`

You may have thought of some methods like `setName()` and `setNumber()` that would be useful in a more realistic example. But let us stop with just two methods.

```
class Entry
{
  String name;
  String number;

  // constructor
  Entry( String n, String num )
  {
    name = n; number = num;
  }

  // methods
  public boolean equals( Object other )
  {
    return name.equals( ((Entry)other).name );
  }

  public String toString()
  {
    return "Name: " + name + " Number: " + number;
  }

}
```

The `toString()` method overrides the `toString()` method that all objects have. Our method returns a more useful string than the inherited method.

QUESTION 20:

What does the following code write?

```
Entry ent = new Entry( "Amy", "123-4567");

System.out.println( ent );
```

A good answer might be:

```
Name: Amy Number: 123-4567
```

The `equals( Object )` Method

```
class Entry
{
  String name;
  String number;

  . . . . .

  // methods
  public boolean equals( Object other )
  {
    return name.equals( ((Entry)other).name );
  }

  . . . . .
}
```

Carefully study the `equals( Object )` method. The reason for this phone book example is to discuss this method. This method presents a common problem (using an Object reference) and uses the typical solution (a type cast to a specific type.)

All classes have an `equals( Object )` method because the `Object` class defines it. (You may wish to look at the Java documentation for `Object` at this point.) Most classes override the method with a more appropriate method.

The parameter list is `( Object other )`. This says that `equals()` will be used with a reference to any other object. However, in our particular application we will be comparing two `Entry` objects with each other. (One entry will contain the name we are searching for; the other will be an entry in a `Vector` of data.)

We know what the parameter will be (since we are writing the code), but the compiler does not. Without more specific information, the compiler will assume only the methods and members of `Object`. To tell the compiler what to expect, a type cast must be used:

```
    return name.equals( ((Entry)other).name );
```

This tells the compiler that `other` is a reference to an `Entry`. This must be done to access the `name` field of the `Entry` object.

You might wonder: why not write the `equals()` method this way:

```
public boolean equals( Entry other )
{
  return name.equals( other.name );
}
```

This is a correct method, but it *does not override* the `equals( Object )` method that all objects have. It is this last method that is used with the `indexOf()` method. We must override it to use that built−in searching method of `Vector`.

QUESTION 21:

What will the compiler do with the following method:

```
class Entry
{
  . . . . .

  // methods
  public boolean equals( Object other )
  {
    return name.equals( other.name );
  }

  . . . . .
}
```

A good answer might be:

It will find a syntax error because `name` is not an instance variable of `Object`.

Test Program

These ideas are a little bit confusing. It is a good idea to write a test program to see if things work as expected:

```
import java.util.* ;
import java.io.*;

class Entry
{
  String name;
  String number;

  // constructor
  Entry( String n, String num )
  {
    name = n; number = num;
  }

  // methods
  public boolean equals( Object other )
  {
    return name.equals( ((Entry)other).name );
  }

  public String toString()
  {
    return "Name: " + name + " Number: " + number;
  }

}

class PhoneBookTest
{
  public static void main ( String[] args) throws IOException
  {
    Vector phone = new Vector( 10 );

    phone.addElement( new Entry( "Amy", "123-4567") );
    phone.addElement( new Entry( "Bob", "123-6780") );
    phone.addElement( new Entry( "Hal", "789-1234") );
    phone.addElement( new Entry( "Deb", "789-4457") );
    phone.addElement( new Entry( "Zoe", "446-0210") );

    // look for Hal in phone using our equals() method
    int spot = phone.indexOf( new Entry( "Hal", "") ) ;

    System.out.println( "indexOf returns: " + spot ) ;
  }
}
```

The entry we build as the target of the search has an empty string for the phone number. The `equals()` method will compare it with entries that have a non-empty phone number. But `equals()` is written to look only at the `name` of the two entries, so this is OK.

QUESTION 22:

What will the program write?

A good answer might be:

indexOf returns 2

Complete Program

After playing with the test program a while to check that things work you can write the complete program:

```
import java.util.* ;
import java.io.*;

class Entry
{
  String name;
  String number;

  // constructor
  Entry( String n, String num )
  {
    name = n; number = num;
  }

  // methods
  public boolean equals( Object other )
  {
    return name.equals( ((Entry)other).name );
  }

  public String toString()
  {
    return "Name: " + name + " Number: " + number;
  }

}

class PhoneBookAp
{
  public static void main ( String[] args) throws IOException
  {
    Vector phone = new Vector( 10 );

    phone.addElement( new Entry( "Amy", "123-4567") );
    phone.addElement( new Entry( "Bob", "123-6780") );
    phone.addElement( new Entry( "Hal", "789-1234") );
    phone.addElement( new Entry( "Deb", "789-4457") );
    phone.addElement( new Entry( "Zoe", "446-0210") );

    String name;
    BufferedReader stdin = new BufferedReader(
      new InputStreamReader( System.in ) );

    System.out.print("Enter name -->");
    name = stdin.readLine().trim();

    while( !name.equals("quit" ) )
    {
      int spot = phone.indexOf( new Entry( name, "") ) ;

      if ( spot >= 0 )
        System.out.println( phone.elementAt( spot ) ) ;
      else
        System.out.println( name + " not found" ) ;

      System.out.print("Enter name -->") ;
      name = stdin.readLine().trim() ;
    }

  }
```

```
}
```

Of course, this is a small example and not a practical program. But the techniques it uses are used in many industrial−strength programs. Carefully examine:

1. How a `Vector` of user−defined objects is used.
2. How the `equals()` method is written.
    - ♦ It needs to override the inherited method for the `indexOf()` method to work.
    - ♦ Type casting must be used inside the method.
3. Use of the `indexOf()` method to search the vector.
    - ♦ A "target" object is constructed for use with the `indexOf()` method.

You may regard all this as more bother than it is worth, since the previous "phone book" example worked fine using a plain array. But in large programs a linear arrangement of data is so common, and the operations on it are so frequent, that the `Vector` class is a worthwhile time saver.

QUESTION 23:

The application would be more practical if it would give the user more freedom in entering the target name. For example, the user should be able to enter all lower case or all caps. Where would you make a change to the program to allow this?

A good answer might be:

The `equals()` method can be made more user friendly.

End of the Chapter

You have reached the end this chapter. You may wish to review the following. Click on a blue subject that interests you to go to where it was discussed. To get back here, click on the "back arrow" button of your browser.

- `Vector` class overview.
- `Vector` constructors.
- Capacity and size of a `Vector`.
- addElement() method.
- setElementAt() and elementAt() method.
- removeElementAt() method.
- insertElementAt() method.
- indexOf() method.
- enumeration objects and the elements() method.
- Need to override the equals() method.

Back to the main menu.

You have reached the end of the chapter.