

Java Class 22



Airport Baggage Handling System

26 miles of track were constructed to move luggage between gates and terminals in the new Denver airport due to open in March 1994. However the planned opening was delayed until Feb. 1995 due to failures in the baggage handling system. Software errors caused bags to be loaded onto already full carts and bags to be unloaded onto already jammed belts. Costs of these errors were \$286M plus another \$80M for United, which tried to use the system for a while. All baggage movement is now totally controlled and transported by humans.

Interfaces

A Java *interface* is a collection of abstract methods and constants

An *abstract method* is a method header without a method body


An abstract method can be declared using the modifier `abstract`, but because all methods in an interface are `abstract`, usually it is left off

An interface is used to establish a set of methods that a class will implement


Interfaces

interface is a reserved word

None of the methods in an interface are given a definition (body)




```
public interface Doable
{
    public void doThis();
    public int doThat();
    public void doThis2 (double value, char ch);
    public boolean doTheOther (int num);
}
```



A semicolon immediately follows each method header

Interfaces


**implements is a
reserved word**



```
public class CanDo implements Doable
{
    public void doThis ()
    {
        // whatever
    }

    public void doThat ()
    {
        // whatever
    }

    // etc.
}
```



**Each method listed
in Doable is
given a definition**

Interfaces

An interface cannot be instantiated

Methods in an interface have public visibility by default

A class formally implements an interface by:

- stating so in the class header
- providing implementations for every abstract method in the interface

If a class declares that it implements an interface, it must define all methods in the interface

Interfaces

In addition to (or instead of) abstract methods, an interface can contain constants

When a class implements an interface, it gains access to all its constants

A class that implements an interface can implement other methods as well

```
//*****  
// Complexity.java          Author: Lewis/Loftus  
//  
// Represents the interface for an object that can be assigned an  
// explicit complexity.  
//*****  
  
public interface Complexity  
{  
    public void setComplexity (int complexity);  
    public int getComplexity();  
}
```

```

//*****
//  Question.java          Author: Lewis/Loftus
//
//  Represents a question (and its answer).
//*****

public class Question implements Complexity
{
    private String question, answer;
    private int complexityLevel;

    //-----
    //  Constructor: Sets up the question with a default complexity.
    //-----
    public Question (String query, String result)
    {
        question = query;
        answer = result;
        complexityLevel = 1;
    }
}

```

continue

continue

```
//-----  
//  Sets the complexity level for this question.  
//-----  
public void setComplexity (int level)  
{  
    complexityLevel = level;  
}  
  
//-----  
//  Returns the complexity level for this question.  
//-----  
public int getComplexity()  
{  
    return complexityLevel;  
}  
  
//-----  
//  Returns the question.  
//-----  
public String getQuestion()  
{  
    return question;  
}
```

continue

continue

```
//-----  
// Returns the answer to this question.  
//-----  
public String getAnswer()  
{  
    return answer;  
}  
  
//-----  
// Returns true if the candidate answer matches the answer.  
//-----  
public boolean answerCorrect (String candidateAnswer)  
{  
    return answer.equals(candidateAnswer);  
}  
  
//-----  
// Returns this question (and its answer) as a string.  
//-----  
public String toString()  
{  
    return question + "\n" + answer;  
}  
}
```

```

//*****
//  MiniQuiz.java          Author: Lewis/Loftus
//
//  Demonstrates the use of a class that implements an interface.
//*****

import java.util.Scanner;

public class MiniQuiz
{
    //-----
    //  Presents a short quiz.
    //-----
    public static void main (String[] args)
    {
        Question q1, q2;
        String possible;

        Scanner scan = new Scanner (System.in);

        q1 = new Question ("What is the capital of Jamaica?",
                           "Kingston");
        q1.setComplexity (4);

        q2 = new Question ("Which is worse, ignorance or apathy?",
                           "I don't know and I don't care");
        q2.setComplexity (10);
    }
}

```

continue

continue

```
System.out.print (q1.getQuestion());
System.out.println (" (Level: " + q1.getComplexity() + ")");
possible = scan.nextLine();
if (q1.answerCorrect(possible))
    System.out.println ("Correct");
else
    System.out.println ("No, the answer is " + q1.getAnswer());

System.out.println();
System.out.print (q2.getQuestion());
System.out.println (" (Level: " + q2.getComplexity() + ")");
possible = scan.nextLine();
if (q2.answerCorrect(possible))
    System.out.println ("Correct");
else
    System.out.println ("No, the answer is " + q2.getAnswer());
}
}
```

contin

Sample Run

What is the capital of Jamaica? (Level: 4)

Kingston

Correct

Which is worse, ignorance or apathy? (Level: 10)

apathy

No, the answer is I don't know and I don't care

```
System.out.println();
System.out.print (q2.getQuestion());
System.out.println (" (Level: " + q2.getComplexity() + ")");
possible = scan.nextLine();
if (q2.answerCorrect(possible))
    System.out.println ("Correct");
else
    System.out.println ("No, the answer is " + q2.getAnswer());
}
```

Interfaces

A class can implement multiple interfaces

The interfaces are listed in the `implements` clause

The class must implement all methods in all interfaces listed in the header

```
class ManyThings implements interface1, interface2
{
    // all methods of both interfaces
}
```

Interfaces

The Java API contains many helpful interfaces

The `Comparable` interface contains one abstract method called `compareTo`, which is used to compare two objects

The `String` class implements `Comparable`, giving us the ability to put strings in lexicographic order

The Comparable Interface

Any class can implement `Comparable` to provide a mechanism for comparing objects of that type

```
if (obj1.compareTo(obj2) < 0)
    System.out.println ("obj1 is less than obj2");
```

The value returned from `compareTo` should be negative if `obj1` is less than `obj2`, 0 if they are equal, and positive if `obj1` is greater than `obj2`

It's up to the programmer to determine what makes one object less than another

The Iterator Interface

An iterator is an object that provides a means of processing a collection of objects one at a time

An iterator is created formally by implementing the `Iterator` interface, which contains three methods

- The `hasNext` method returns a boolean result – true if there are items left to process
- The `next` method returns the next object in the iteration
- The `remove` method removes the object most recently returned by the `next` method

Interfaces

The use of interfaces is a quick and simple scheme to speed the development of large-scale Java applications

Java interfaces are a blueprint for the functionality contained in an associated object

Interfaces

Another reason to code using interfaces is that it provides higher efficiency in the various phases of a system's lifecycle:

- **Design:** the methods of an object can be quickly specified and published to all affected developers
- **Development:** the Java compiler guarantees that all methods of the interface are implemented with the correct signature and that all changes to the interface are immediately visible to other developers
- **Integration:** there is the ability to quickly connect classes or subsystems together, due to their well-established interfaces
- **Testing:** interfaces help isolate bugs because they limit the scope of a possible logic error to a given subset of methods

Algorithms

An *algorithm* is a step-by-step process for solving a problem. A recipe is an example of an algorithm. Travel directions are another example of an algorithm. Every method implements an algorithm that determines how that method accomplishes its goals.

An algorithm is often described using *pseudocode*, which is a mixture of code statements and English phrases. Pseudocode provides enough structure to show how the code will operate, without getting bogged down in the syntactic details of a particular programming language or becoming prematurely constrained by the characteristics of particular programming constructs.

Method Design

As we've discussed, high-level design issues include:

- identifying primary classes and objects
- assigning primary responsibilities

After establishing high-level design issues, it's important to address low-level issues such as the design of key methods

For some methods, careful planning is needed to make sure they contribute to an efficient and elegant system design

A method should be relatively small, so that it can be understood as a single entity

A potentially large method should be decomposed into several smaller methods as needed for clarity

Method Decomposition

Let's look at an example that requires method decomposition – translating English into Pig Latin

Pig Latin is a language in which each word is modified by moving the initial sound of the word to the end and adding "ay"

Words that begin with vowels have the "yay" sound added on the end

book → ookbay

table → abletay

item → itemyay

chair → airchay

Method Decomposition

The primary objective (translating a sentence) is too complicated for one method to accomplish

Therefore we look for natural ways to decompose the solution into pieces

Translating a sentence can be decomposed into the process of translating each word

The process of translating a word can be separated into translating words that:

- begin with vowels
- begin with consonant blends (sh, cr, th, etc.)
- begin with single consonants

```
//*****
//  PigLatin.java      Author: Lewis/Loftus
//
//  Demonstrates the concept of method decomposition.
//*****

import java.util.Scanner;

public class PigLatin
{
    //-----
    //  Reads sentences and translates them into Pig Latin.
    //-----
    public static void main (String[] args)
    {
        String sentence, result, another;

        Scanner scan = new Scanner (System.in);

continue
```


continue

```
do
{
    System.out.println ();
    System.out.println ("Enter a sentence (no punctuation):");
    sentence = scan.nextLine();

    System.out.println ();
    result = PigLatinTranslator.translate (sentence);
    System.out.println ("That sentence in Pig Latin is:");
    System.out.println (result);

    System.out.println ();
    System.out.print ("Translate another sentence (y/n)? ");
    another = scan.nextLine();
}
while (another.equalsIgnoreCase("y"));
}
```

continue

do

{

Syst

Syst

sent

Syst

resu

Syst

Syst

Syst

Syst

anot

}

while

}

}

Sample Run

Enter a sentence (no punctuation):

Do you speak Pig Latin

That sentence in Pig Latin is:

oday ouyay eakspay igpay atinlay

Translate another sentence (y/n)? y

Enter a sentence (no punctuation):

Play it again Sam

That sentence in Pig Latin is:

ayplay ityay againyay amsay

Translate another sentence (y/n)? n

```
//*****
//  PigLatinTranslator.java          Author: Lewis/Loftus
//
//  Represents a translator from English to Pig Latin. Demonstrates
//  method decomposition.
//*****
```

```
import java.util.Scanner;
```

```
public class PigLatinTranslator
{
```

```
    //-----
    //  Translates a sentence of words into Pig Latin.
    //-----
```

```
    public static String translate (String sentence)
    {
```

```
        String result = "";
```

```
        sentence = sentence.toLowerCase();
```

```
        Scanner scan = new Scanner (sentence);
```

```
        while (scan.hasNext())
```

```
        {
            result += translateWord (scan.next());
            result += " ";
        }
```

continue

continue

```
        return result;
    }

    //-----
    //  Translates one word into Pig Latin. If the word begins with a
    //  vowel, the suffix "yay" is appended to the word. Otherwise,
    //  the first letter or two are moved to the end of the word,
    //  and "ay" is appended.
    //-----
    private static String translateWord (String word)
    {
        String result = "";

        if (beginsWithVowel(word))
            result = word + "yay";
        else
            if (beginsWithBlend(word))
                result = word.substring(2) + word.substring(0,2) + "ay";
            else
                result = word.substring(1) + word.charAt(0) + "ay";

        return result;
    }
}
```

continue

continue

```
//-----  
//  Determines if the specified word begins with a vowel.  
//-----  
private static boolean beginsWithVowel (String word)  
{  
    String vowels = "aeiou";  
  
    char letter = word.charAt(0);  
  
    return (vowels.indexOf(letter) != -1);  
}
```

continue

continue

```
//-----  
//  Determines if the specified word begins with a particular  
//  two-character consonant blend.  
//-----  
private static boolean beginsWithBlend (String word)  
{  
    return ( word.startsWith ("bl") || word.startsWith ("sc") ||  
             word.startsWith ("br") || word.startsWith ("sh") ||  
             word.startsWith ("ch") || word.startsWith ("sk") ||  
             word.startsWith ("cl") || word.startsWith ("sl") ||  
             word.startsWith ("cr") || word.startsWith ("sn") ||  
             word.startsWith ("dr") || word.startsWith ("sm") ||  
             word.startsWith ("dw") || word.startsWith ("sp") ||  
             word.startsWith ("fl") || word.startsWith ("sq") ||  
             word.startsWith ("fr") || word.startsWith ("st") ||  
             word.startsWith ("gl") || word.startsWith ("sw") ||  
             word.startsWith ("gr") || word.startsWith ("th") ||  
             word.startsWith ("kl") || word.startsWith ("tr") ||  
             word.startsWith ("ph") || word.startsWith ("tw") ||  
             word.startsWith ("pl") || word.startsWith ("wh") ||  
             word.startsWith ("pr") || word.startsWith ("wr") );  
}
```

Objects as Parameters

Another important issue related to method design involves parameter passing

Parameters in a Java method are *passed by value*

A copy of the *actual parameter* (the value passed in) is stored into the *formal parameter* (in the method header)

When an object is passed to a method, the actual parameter and the formal parameter become aliases of each other

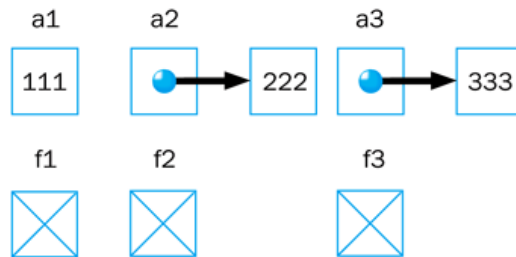
Passing Objects to Methods

What a method does with a parameter may or may not have a permanent effect (outside the method)

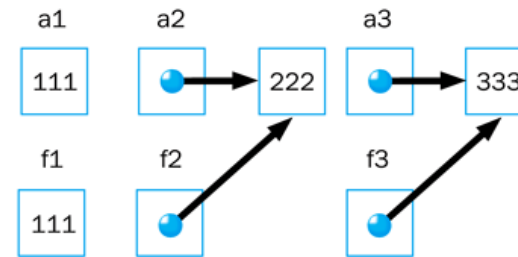
Note the difference between changing the internal state of an object versus changing which object a reference points to

STEP 1

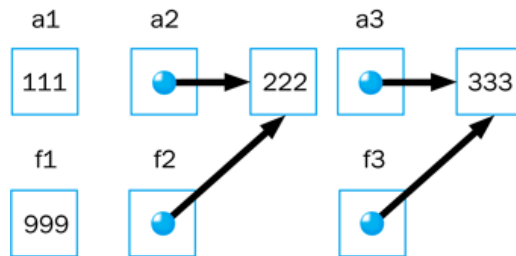
Before invoking changeValues

**STEP 2**

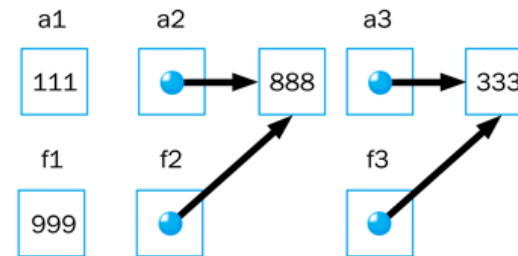
tester.changeValues (a1, a2, a3);

**STEP 3**

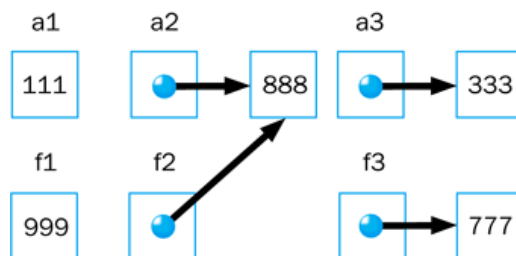
f1 = 999;

**STEP 4**

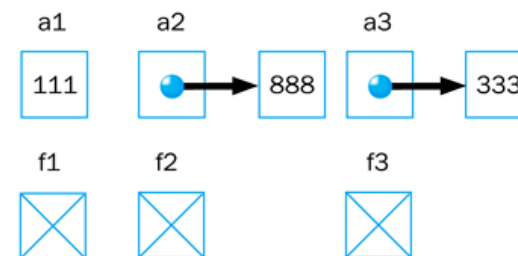
f2.setValue (888);

**STEP 5**

f3 = new Num (777);

**STEP 6**

After returning from changeValues



Method Overloading

Let's look at one more important method design issue: method overloading

Method overloading is the process of giving a single method name multiple definitions in a class

If a method is overloaded, the method name is not sufficient to determine which method is being called

The *signature* of each overloaded method must be unique

The signature includes the number, type, and order of the parameters

Method Overloading

The compiler determines which method is being invoked by analyzing the parameters

```
float tryMe(int x)
{
    return x + .375;
}
```

Invocation

```
result = tryMe(25, 4.32)
```

```
float tryMe(int x, float y)
{
    return x*y;
}
```



Method Overloading

The `println` method is overloaded:

```
println (String s)
```

```
println (int i)
```

```
println (double d)
```

and so on...

The following lines invoke different versions of the `println` method:

```
System.out.println ("The total is:");
```

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

Overloading Methods

The return type of the method is not part of the signature

That is, overloaded methods cannot differ only by their return type

Constructors can be overloaded

Overloaded constructors provide multiple ways to initialize a new object

Group Exercises

Ex: 7.1

Ex: 7.2

Ex: 7.10

Ex: 7.11

Ex: 7.12

Assignment for Class 23

Review Question, MiniQuiz, PigLatin, PigLatinTranslator

Read Chapter 9.1, 9.2, 9.3, 9.4, 9.5