From the book "Integration-Ready Architecture and Design" by Cambridge University Press

Yefim (Jeff) Zhuk

Java and C#: A Saga of Siblings

We will also discuss support of integration-ready and knowledge-connected environments that allow for writing application scenarios. In spite of the fact that most examples that support this method are made in Java, similar environments can be created with other languages. Life outside of Java is not as much fun, but it is still possible.

This appendix provides examples in which the same function is implemented not in Java but in its easiest replacement – the C–Sharp (C#) language. This lucky child inherited good manners, elegance, style, and even some clothing from its stepfather while enjoying its mother's care and her rich, vast NETwork. Java and C# are similar in their language structure, functionality, and ideology. Learning from each other and growing stronger in the healthy competition the siblings perfectly serve the software world.

Java Virtual Machine and Common Language Runtime.

Java compilation produces a binary code according to the Java language specification. This binary code is performed on any platform in which a Java Virtual Machine (JVM) is implemented. The JVM interprets this binary code at run-time, translating the code into specific platform instructions.

C# compilation (through Visual Studio. NET) can produce the binary code in Common Intermediate Language (CIL) and save it in a portable execution (PE) file that can then be managed and executed by the Common Language Runtime (CLR). The CLR, which Microsoft refers to as a “managed execution environment,” in a way is similar to JVM. A program compiled for the CLR does not need a language-specific execution environment and can run on almost any MS Windows system in which CLR is present.

There is one big difference between JVM and CLR. JVM makes Java a multiplatform language. CLR is a multilingual environment implemented on MS Windows platforms. Visual Studio .NET supports several programming languages, such as C, C#, C++, Java, Visual Basic , Practical Extraction and Report Language (Perl), and Cobol, providing compilation from these languages to CIL, followed by execution by the CLR.

Garbage Collection and Performance

Java, as well as .NET environments, does not provide too much control over memory management. Both technologies offer a garbage collector mechanism instead. The garbage collector periodically looks for objects that have no references in the current code and frees (deallocates) memory from these objects.

Keep in mind that memory management is a pretty expensive system operation. If the garbage collection thread starts when your user is waiting for a program’s response, the program's response will be visibly delayed and the user might become frustrated with your program's performance.

For both (Java and .NET) environments, it is possible to escape this situation, or at least make it less likely. The solution is simple and can be addressed by two lines that look almost identical in your Java or C# code.

Assign heavy objects to null as soon as you do not need them:

```
myHeavyObject = null;
```

The garbage collector will almost immediately free this object

Force the garbage collector to work at a time that is not critical for the application. Insert the line below, for example, after your code requests an input/output operation:

```
System.GC.Collect(); // syntax for C#
```

Or

```
System.gc(); // syntax for Java
```

Java and C# Basics: Keywords from "abstract" till "while"

As you can see from the beginning, the two languages are very close in their syntax and mentality. About 90% of their keywords are the same in spelling and meaning, and both Java and C# keywords start with lower-case letters:

abstract – A Java/C# keyword used in a class declaration to specify a class that is not complete, and cannot have object-instances. An abstract class regularly has some abstract methods, and serves as a base class for the subclasses, which implement those abstract methods and can have object-instances.

```
// Java example
```

```
public abstract class Shape {
    // data
    public void draw(); // abstract method with no implementation
    // more code that can include implemented and abstract methods
}
public class Rectangle extends Shape {
    public void draw() {
         // specific implementation of the method
    }
}
// C# example
using System;
public abstract class Shape {
    // data
    public void Draw(); // abstract method with no implementation
    // more code that can include implemented and abstract methods
}
public class Rectangle : Shape {
    public void Draw() {
         // specific implementation of the method
    }
}
// The line below will produce the error:
// Cannot instantiate an object of the abstract class Shape
Shape s = new Shape(); // ERROR line: we try to instantiate the
abstract class
// The line below is valid.
Shape s = new Rectangle(); // OK line, the Rectangle is not an abstract
class
```

assert – A Java keyword that tests a Boolean expression, for example, *assert (a==b)* and throws an *AssertionError* exception if the specified Boolean expression is false. C# does not have this keyword.

boolean – A Java keyword. The C# version of this keyword is **bool**. Java and C# define this keyword as a type that can hold only one of the literal values true and false.

```
// Java example
// Some code that fills strings "a" and "b" with some values
// check if strings are equal
// store the result of comparison in the boolean variable
boolean resultOfComparison = a.equals(b);
// C# example
// Some code that fills strings "a" and "b" with some values
// check if strings are equal
// store the result of comparison in the boolean variable
bool resultOfComparison = a.Equals(b); // C# method names start with
upper case
```

break – A Java/C# keyword that stops the current program block (loop) execution and passes control to the next block of the program.

```
Example (valid for Java and C#)
for(;;) { // indefinite loop
    // some code
    if(a < b) { // check if a is less than b
       // at this point the for loop will be interrupted
       break; // pass control to the next block of the program after
the loop
    }
    // more code
```

```
}
// the next block of the program
// some code
```

byte – A Java keyword that represents a sequence of eight bits as a signed integer number. The C# language defines the corresponding data type as the **sbyte**.

```
// Java example
byte a = 13;
// C# example
sbyte a = 13;
```

case – A Java/C# keyword that follows a conditional switch declaration to define a block of a program to pass the control to, if the expression specified in the switch matches the case value.

```
// Java/C# example:
switch(number) {
    case 1:
       response = "hello";
       break;
    case 2:
       response = "good bye";
       break;
}
```

catch – A Java/C# keyword used in the *try/catch* block of statements. The *catch* block of statements is executed if an exception or run-time error occurs in a preceding *try* block.

```
// Java example:
import java.io.*;
// class definition
```

```
// method definition
try { // IO operation can potentially trigger exception
   File inputFile = new File("myFile.txt");
   FileReader reader = new FileReader(inputFile);
} catch(Exception e) {
    System.out.println("ERROR: " + e);
}
// C# example:
using System;
using System.IO;
// class definition
// method definition
try { // IO operation can potentially trigger exception
   FileStream inputFile = new FileStream("myFile.txt", FileMode.Open);
   StreamReader reader = new StreamReader(inputFile);
} catch(Exception e) {
    Console.Write("ERROR: " + e);
}
```

char – A Java/C# keyword that declares a primitive textual data type, a 16-bit, unsigned, Unicode character.

```
// Java/C# example:
char c = 'c';
```

continue – A Java/C# keyword used to resume program execution at the end of the current loop.

```
// Java/C# example:
int[] numbers = new int[10];
for(int i=0;i < 10; i++) {
    // the first part of the loop
```

```
    numbers[i] = i;
    if(a < b) { // check if the value “a” is less than the value “b”
       // at this point the for loop will be interrupted
       continue; // skip the second part of the loop, increase the
value of i
    }
    // the second part of the loop
    numbers[i] = number[i] * 2; // will not be executed if condition
above is met
}
```

default – A Java/C# keyword used optionally in a *switch* statement after all of the *case* conditions. The default statement will be executed if a case condition does not match the value of the switch variable.

```
// Java/C# example:
switch(number) {
    case 1:
       response = “hello”;
       break;
    case 2:
       response = “good bye”;
       break;
    default:
       response = “Please re-enter your data”;
}
```

do – A Java/C# keyword that declares a loop that will iterate a statement block. The *while* keyword at the end of the block can specify the loop exit condition.

```
// Java/C# example:
int[] numbers = new int[10];
```

```
int i = 0;
do {
    // the first part of the loop
    numbers[i] = i++;
} while(i < 10);
```

double – A Java/C# keyword that defines a floating point number with *double precision*.

```
// Java/C# example:
double preciseNumber = 16.5;
```

else – A Java/#C keyword used in *if-else* block statements. When the test expression specified in the *if* statement is false, the program will execute the *else* block statement.

```
// Java/C# example:
    String response = "";
    if(a < b) { // check if the value "a" is less than the value "b"
       response = "Add value please.";
    } else {
       response = "Enough, thank you.";
    }
```

extends – A Java keyword used to define a subclass that is derived and inherited from a base class. One interface can *extend* another interface by adding more methods. C# uses the ":" character to define inheritance.

```
// Java example
public abstract class Shape {
    // data
    public void draw(); // abstract method with no implementation
    // more code that can include implemented and abstract methods
}
```

```
public class Rectangle extends Shape {
    public void draw() {
        // specific implementation of the method
    }
}
// C# example
using System;
public abstract class Shape {
    // data
    public void Draw(); // abstract method with no implementation
    // more code that can include implemented and abstract methods
}
public class Rectangle : Shape {
    public void Draw() {
        // specific implementation of the method
    }
}
```

final – A Java keyword that defines an unchangeable entity. You cannot change a *final* variable from its initialized value, cannot extend a *final* class, or override a *final* method. C# uses the **sealed** keyword to express the same concept.

```
// Java example:
final private int READ_ONLY_MODE = 9;
// C# example:
sealed private int READ_ONLY_MODE = 9;
```

finally – A Java/C# keyword that is used in *try/catch* block statements to ensure execution of the following block of statements, regardless of whether an *Exception*, or run-time error, occurred in the *try* statement block.

```
// Java example:
import java.io.*;
// class definition
// method definition
try { // IO operation can potentially trigger exception
   File inputFile = new File("myFile.txt");
   FileReader reader = new FileReader(inputFile);
} catch(Exception e) {
    System.out.println("ERROR: " + e);
} finally {
      reader.close();
}
// C# example:
using System;
using System.IO;
// class definition
// method definition
try { // IO operation can potentially trigger exception
   FileStream inputFile = new FileStream("myFile.txt", FileMode.Open);
   StreamReader reader = new StreamReader(inputFile);
} catch(Exception e) {
    Console.Write("ERROR: " + e);
} finally {
      reader.close();
}
```

float – A Java/C# keyword that defines a floating point number with *single precision*.

```
// Java/C# example:
float singlePrecisionNumber = 1.459F;
```

for – A Java/C# keyword that declares a loop with an optional initial statement. This statement includes a condition to exit and additional executable statements.

```
// Java/C# example
int[] numbers = new int[10];
for(int i=0;i < 10; i++) {
    // block of statements
    numbers[i] = i;
}
```

if – A Java/C# keyword that evaluates a conditional statement, and then executes a statement block if the result of the evaluation is true.

```
// Java/C# example:
    String response = “”;
    if(a < b) { // check if the value “a” is less than the value “b”
       response = “Add value please.”;
    }
```

implements – A Java keyword, an optional part of a class declaration, that specifies interfaces that are implemented by the class. C# supports the same concept with the “:” character, as it does for base class–subclass relationships.

```
// Java example:
public class KnowledgeService implements ServiceScenario {
    // class definition
}
// C# example:
public class KnowledgeService : ServiceScenario {
    // class definition
}
```

import – A Java keyword that, at the beginning of a source, points to a class from another package or a whole package of classes that are needed by this class. C# provides the **using** keyword for the same purpose.

```
// Java example:
import java.awt.Toolkit; // a single class
import java.io.*;             // a package
// C# example:
using System;
using System.Net;
```

instanceof – A Java keyword that tests whether the specified run-time object type is an instance of the specified class in the same evaluation expression. C# uses the **is** keyword instead.

```
// Java example:
if(aShape instanceof Rectangle) {
    // do something
}
// C# example:
if(aShape is Rectangle) {
    // do something
}
```

interface – This Java/C# keyword defines a collection of method definitions and constants. A class usually implements an interface.

```
// Java/C# example:
public interface ServiceScenario {
     // method definitions and abstract methods
     // constants
}
```

long – This Java/C# keyword defines a 64-bit numeric integer variable.

```
// Java/C# example:
long preciseNumber = 64000L;
```

native – This Java keyword may be used in method declarations to specify the method implemented in a non-Java programming language and located in some library file. The *System.loadLibrary()* method loads this library file and makes the native method available for the Java run-time environment. C# uses the **extern** keyword to indicate a non-C# method. An external or native method declaration has no actual implementation in the current source because it was implemented in a different language. In this regard, its syntax is similar to abstract methods. The *DLLImport* attributes point to a library (Dynamic Link Library) and the parameters needed to invoke the method.

```
// Java example:
public class ITSNativeExample {
    private native void copyFile (String inputFilename, String
copyFilename);
   static
   {
      System.loadLibrary("myNativeLibrary");
   }
   public static void main(String[] args) {
        ITSNativeExample example = new ITSNativeExample();
        example.copyFile("source", "sourceCopy");
   }
}
// C# example:
[DllImport("KERNEL32.DLL", EntryPoint="CopyFileW",  SetLastError=true,
CharSet=CharSet.Unicode, ExactSpelling=true,
```

```
CallingConvention=CallingConvention.StdCall)]
public static extern bool CopyFile(String inputFilename, String
copyFilename);
```

new – A Java/C# keyword that creates a new object-instance of a class.

```
// Java/C# example:
File f = new File("notes.txt");
```

package – A Java keyword that declares that the current class is a member of a package, in other words, a library of classes. C# uses the **namespace** keyword for the same purpose.

```
// Java example
package com.its.connector;
import java.io.*;
public class IOMaster {
     // class definition
}
// C# example
namespace ITS.Connector;
using System.IO;
public class IOMaster {
     // class definition
}
```

private – A Java/C# keyword used in a method or variable declaration to restrict access to the method or variable to only elements of its own class.

```
// Java/C# example
private int number;
```

protected – A Java/C# keyword that provides more than private, but less than public, access to a method or variable. Protected class members are visible in Java not only to classes derived from any package but also to any class from the same package. Unlike Java, C# opens access to protected data and methods only to derived classes but does not allow access for any other classes, even from the same *namespace*.

```
// Java/C# example:
protected Hashtable services;
```

public – A Java/C# keyword used in a method or variable declaration to open access to the method or variable to all classes.

```
// Java/C# example:
public final int READ_ONLY_MODE = 9;
```

return – A Java/C# keyword that ends the execution of a method. The keyword may be followed by a value required by the method declaration.

```
// Java/C# example:
public int calculateNumbers(int a, int b) {
    int c = a+b;
    return c;
}
```

short – A Java/C# keyword used to define a 16-bit integer.

```
// Java/C# example:
short scaryNumber = 13;
```

static – A Java/C# keyword used to define a single copy of a class variable or method shared by all object-instances of the class. One can access a static variable or static method even without creating an object of the class.

```
// Java/C# example:
// In Java case two classes below must be stored in a different files
// with the filenames "Math.java" and "MathActions.java" accordingly
// C# has no restrictions on source filenames
```

```
// Unlike Java, C# allows us to keep more than one public class in a
source file
public class Math {
    public static double PI = 1.4591;
    // more code
}
public class MathActions {
    public double getCircleLength(double radius) {
        return 2 * Math.PI * radius;
    }
    // more code
}
```

strictfp – A Java language keyword-modifier that means *strict floating point arithmetic*. This type of modifier may apply to a class, interface, or method to declare an expression *FP-strict*. You cannot use the *strictfp* keyword on constructors or methods within interfaces, although you can declare a *strictfp* class to make all constructors and methods of the class *FP-strict.*

When is *strictfp* important? It is needed to guarantee common floating-point arithmetic across different Java implementations or hardware platforms. The *strictfp* keyword in the following example keeps the expression from overflowing and produces a final result that is within range, even if the *price* argument is close to the maximum value of a double (*Double.MAX_VALUE*).

C# has no adequate keyword and has no need for this concept because the language runs on "wintel" (Windows–Intel) platforms.

```
// Java example
    public strictfp class ITSStrictFPExample {
```

```
        public double getDoublePrice(double price) {
            double doublePrice = 2.0 * d;
            return doublePrice;
        }
    }
```

super – A Java keyword used in a subclass to access members of a base class inherited by the subclass. C# uses the keyword **base**. You can do the same things in C# with the *base* keyword that you can in Java with *super*, except one: you cannot invoke a base class constructor this way. Don't worry; there is a way to invoke a base class constructor in C# from a subclass. For example:

```
 // Java example:
public class Child extends Parent {
    public Child () {
     `    // invoke a Parent's constructor
         super();
         // invoke a method from the Parent class
         super.takeCareOfChildren();
         // do something childish
    }
}
// C# example:
public class Child : Parent {
    // the Child constructor starts with a Parent's constructor
invocation
    public Child ()  : base Parent() { // invoke a Parent's constructor
first
         // invoke a method from the Parent class
         base.takeCareOfChildren();
```

```
        // do something childish
    }
}
```

switch – A Java/C# keyword used to compare an expression or a variable with a value specified by the *case* keyword in order to execute a group of statements following the case if the case value matches the *switch* expression. The *switch* expression must be of type *char*, *byte*, *short*, or *int*.

```
// Java/C# example
switch(a+b) {
    case 4:
    response = "you are almost there";
    break;
   case 5:
    response = "Winner!";
    break;
   default:
    response = "try again";
}
```

synchronized – A Java modifier-keyword that can be applied to a method or a block of code to ensure mutually exclusive access to specified objects and guarantee that only one thread at a time executes that code. C# introduces the **lock** modifier-keyword to express the same idea.

C#'s *System.Threading.Monitor* class contains the *Enter* method, which assures that it will be the only thread in the block. C# adds more flexibility to the game of threads and atomic operations. The *Monitor* class also contains the *TryEnter* method, which will try to obtain a lock, perhaps by blocking the piece of code. If this attempt appears to be a failure, the method indicates the failure to lock the object by returning a false value.

I cannot fail to mention C#'s *System.Threading.Interlocked* class with its *Increment*, *Decrement*, and *Exchange* methods, which enable a program to synchronize access to variables that are shared among several threads.

```
// Java example:
   public class ITSSyncExample  {
      private int treasure;
      public synchronized int getTreasure ()  {
          return treasure;
      }
      public synchronized void setTreasure (int value) {
           treasure = value;
      }
   }
// C# example:
   public class ITSSyncExample  {
      private int treasure;
      public lock int GetTreasure ()  {
          return treasure;
      }
       public lock void SetTreasure (int value)  {
           treasure = value;
       }
   }
// another C# example:
using System;
using System.Threading;
```

```
public class ITSAtomicOperationExample  {
    public static int treasure = 1;
    public static void AtomicDecrement() {
        Interlocked.Decrement( ref treasure ); // atomic operation
    }
}
```

throw – A Java/C# keyword that allows the programmer to throw (pass) an object of a class that is inherited from the *Throwable* class to a calling method. In most cases, programmers throw an *Exception* object. This action allows a programmer to escape the hard work of writing *try/catch* statements at the moment of truth (when exception actually happens) and to delegate the responsibility of hunting for the *Exception* to the upper-level method. Note that the *ServiceNotFoundException* extends the *Extension* class.

throws – A Java keyword used in method declarations to specify which exceptions are not handled within the method but are instead passed to the next higher level of the program. In the example below, the *requestService()* method throws the *ClassNotFoundException* if the class is not found in the *classpath*, or throws the *ServiceNotFoundException* if the class is not the *Service* type. The *requestService()* method declaration includes the *throws Exception* statement, which covers all possible exceptions (including the *ServiceNotFoundException*) that extend the *java.lang.Exception* class.

```
// Java example:
public Object requestService ( String serviceName) throws Exception {
    Object service = Class.forName(serviceName).newInstance();
    If(service instanceof Service) {
```

```
        throw new ServiceNotFoundException();
    }
    return service;
}
```

C# does not have a keyword that can be used in a method declaration to announce that the method can throw an exception. As you can see in the following C# example, the program still can throw exceptions without any announcements in the method declaration.

```
// C# example:
using System;
using System.Reflection;
using System.Collections;
public Object requestService ( String serviceName) {
     // load a class from an assembly at runtime
    Type actingClass = Type.GetType(className);
     // activate (instantiate) an object of the type
    Object service = Activator.CreateInstance( actingClass );
    If(!(service is Service)) {
        throw new ServiceNotFoundException();
    }
    return service;
}
```

transient – A Java keyword indicating that a field is not a part of the serialized form of an object. This keyword helps us exclude some fields from the serialized version of the object. Keep in mind that not all Java objects can be serialized. For example, the *Thread* object is not serializable. When Java tries to serialize a bigger object that includes nonserializable objects, the program fails and produces an exception if all nonserializable objects are not marked *transient*. There is no such keyword in C#.

```
// Java example:
public class ServiceProvider {
    private String serviceName; // will be serialized
    private transient ServiceThread; // extends Thread, not
serializable
    // more data
    // more code
}
```

try – A Java/C# keyword that defines a block of statements inside a method that might throw a Java language exception. An optional *catch* block can handle specific exceptions thrown within the *try* block. An optional *finally* block is executed whether or not an exception is thrown. Java enforces the handling of *Exceptions*, whereas C# is more liberal and leaves it up to the programmer whether or not to use *try/catch* statements in the code.

```
// Java example:
import java.io.*;
// class definition
// method definition
try { // IO operation can potentially trigger exception
   File inputFile = new File("myFile.txt");
   FileReader reader = new FileReader(inputFile);
} catch(Exception e) {
    System.out.println("ERROR: " + e);
}
// C# example:
using System;
```

```
using System.IO;
// class definition
// method definition
try { // IO operation can potentially trigger exception
   FileStream inputFile = new FileStream("myFile.txt", FileMode.Open);
   StreamReader reader = new StreamReader(inputFile);
} catch(Exception e) {
    Console.Write("ERROR: " + e);
}
// another C# example:
using System;
using System.IO;
// class definition
// method definition
// no try/catch statements… and C# compiler will "OK" this !
FileStream inputFile = new FileStream("myFile.txt", FileMode.Open);
StreamReader reader = new StreamReader(inputFile);
```

void – A Java/C# keyword used in method declarations to specify that the method does not return a value.

```
// Java/C# example:
public void setName(String name) {
    this.name = name;
    // no return value!
}
```

volatile – A Java/C# keyword used in variable declarations to prohibit reordering instructions related to accessing such variables. Reordering may appear, for example, because of compiler optimizations. Declaring a volatile variable forces the compiler to "take special precautions" against collisions. Concurrent threads will modify the volatile variable (as in the following example) asynchronously, according to the order specified by the source. The volatile modifier cannot be used in interface constants or *final* (*sealed* in C#) variables.

```
// Java/C# example:
public class ITSVolatileDataExample {
    private volatile int counter1, counter2;
    public void setCounters(int counter) {
        counter1 = counter;
        counter2 = counter;
    }
    public void increaseCounters() {
        counter1++;
        counter2++;
     }
     public boolean compareCounters() {
         // should always be true even with multiple concurrent
operations
         return (counter1 == counter2);
     }
}
```

while – A Java/C# keyword that declares a loop that iterates a programming block. The *while* statement specifies a loop exit condition. C# also offers the **foreach** keyword, a convenient way to iterate over the elements of an array.

```
// Java example:
import java.util.*;
public class ITSWhileExample {
```

```
    public ITSWhileExample() {
        Hashtable table = Stringer.parse(xml);
        Enumeration keys = table.keys();
        while(keys.hasMoreElements()) {
            String key = (String)keys.nextElement();
            System.out.println("key=" + key + " value=" +
table.get(key));
         }
    }
}
// C# example:
using System;
using System.Collections;
namespace ITSCsExamples {
    public class ITSWhileExample {
        public ITSWhileExample( String xml ) {
            Hashtable table = Stringer.parse(xml);
            ICollection keys = table.Keys;
            IEnumerator enumerator = keys.GetEnumerator();
            while( enumerator.MoveNext() ) {
                String key = (String)enumerator.Current;
                Console.WriteLine("key=" + key + " value=" +
table[key]);
            }
        }
    }
}
// another C# example with the foreach keyword instead of the while
keyword
```

```
using System;
using System.Collections;
namespace ITSCsExamples {
    public class ITSForeachExample {
        public ITSForeachExample( String xml ) {
            Hashtable table = Stringer.parse(xml);
            ICollection keys = table.Keys;
            foreach( object o in keys ) {
                String key = (String)o;
                Console.WriteLine("key=" + key + " value=" +
table[key]);
            }
        }
    }
}
```

We are done with basic keywords!

With this ammunition, we can climb higher and more difficult peaks on the programming trail from Java to C#.

From Basics to the Next Level on the Java/C# Programming Trail

You've already noticed that every primitive data type in Java has the same name in C#. We found out that C# accepts almost all Java keywords and adds some of its own. For example, C# includes unsigned primitive data types such as *ushort*, *uint*, and *ulong*. The byte primitive in C# is also unsigned, unlike the Java byte. When Java says "byte," C# says "sbyte" – signed byte. The following paragraphs discuss some of the differences between Java and C#; some are only cosmetic.

Exceptions: Java Is Strict, C# Is More Liberal

Java never crashes, it just gives exceptions. Java compilers enforce *try/catch* statements in all input/output and network operations. If exceptions are omitted, the compiler produces error messages. C# has almost exactly same the exception mechanism but leaves the decision of when to use it up to the programmer.

Class Inheritance: Java Says "Extends" and C# Says ":"

C# inherits its terms of inheritance from C++. When Java says that *public class B extends A* – C++, as well as C#, prefers to say that *public class B: A.*

Interfaces: Java Says "Implements" and C# Still Says ":"

Both Java and C# have interfaces. Java makes a very distinctive interface inheritance from class inheritance. Java says, "public class D implements C," and we immediately understand that C is an interface not a class. C# does not really care. C# can say "public class D: C" as well as "public class D: B," where C is an interface and B is a class.

Nevertheless, the meaning of inheritance is the same for both Java and C#. For a derived class (or a subclass that extends a base class), class inheritance means the benefit of ownership of all the nonprivate class members of the base class, including data and methods.

When a class inherits (or as Java rightly says, "implements") an interface, this class has an obligation to provide implementations for all interface methods. Java and C# agree on the function. Both languages allow for a single class inheritance and multiple interfaces. These inheritance rules are very different from those of C++, which allow for multiple class inheritance and have no interfaces, just abstract classes.

Java and C# Languages Allow Us to Use True Polymorphism, But…

C#, as well as C++, requires marking methods, which we plan to override in subclasses as "virtual." Java assumes that all methods are virtual methods and frees programmers from placing such markers.

Polymorphism is the ability of objects to appear in multiple forms. For example, an object of the *Shape* class can appear as a rectangle or a triangle if the programmer had provided such inheritance in the design and source code. The benefit of polymorphism is that one can replace hundreds of lines of procedural code with two lines of object-oriented polymorphic code. Here is the procedural code below (written in C++ /C#):

```
Object aShape =
pictureWithShapes.getNextShapeFromThePictureWithManyShapes();
// check a type of an object and invoke a proper method
if(aShape is Rectangle) {
    drawRectangle();
} else if(aShape is Triangle) {
    drawTriangle();
} else if(aShape is Circle) {
    drawCircle();
}  // etc., etc, etc.
```

The number of shapes may vary, and the source code grows with every new shape added to the picture. Each time a user requires a new shape, the developer has to adjust the code accordingly. Looks like job security, doesn't it?

Here is the source code written using polymorphism:

```
Object aShape =
pictureWithShapes.getNextShapeFromThePictureWithManyShapes();
// Invoke a proper "draw" method defined for a specific subclass
```

```
aShape.draw(); // if aShape is a Rectangle it invokes the "draw" of the
Rectangle class
```

The beauty of polymorphism is that this source code does not grow or change when we add ten, or even a thousand, more shapes to the picture. The line *aShape.draw()* invokes a proper *draw()* method of a proper shape. A proper method is chosen at run-time instead of being bound during compilation. C++, as well as C#, calls such methods *virtual* and uses the *virtual* keyword for these cases. Java considers all methods potentially virtual and omits this keyword.

Polymorphism comes at price: developers must invest the extra time to design it right, defining a base class with methods (e.g., *draw*) that may be overridden by derived classes.

Java Example: A Base Class and Subclasses

```
/**
* The Shape (base) class definition must be stored in the Shape.java
file
`*/
public class Shape {
    // data description
    protected int size;
    // more data description
    // metods
    public void draw() {
         // some implementation
    }
    // more methods
}
/**
```

```
* The Rectangle (child) class definition must be stored in the
Rectangle.java file
*/
public class Rectangle extends Shape {
      // data specific to the Rectangle
      // override methods for the subclass
      public void draw() {
          // method re-definition
      }
 }
/**
* The Triangle (child) class definition must be stored in the
Triangle.java file
*/
public class Triangle extends Shape {
      // data specific to the Triangle
      // override methods for the subclass
      public void draw() {
          // method re-definition
      }
 }
```

C# Example: A Base Class and Subclasses

Note that C# precisely names *virtual* and *overridden* methods.

```
using System;
public class Shape {
    // data description
    protected int size; // read below about a slight difference on
protected keyword
```

```
    // more data
    // metods
    // Note the virtual keyword in the base class method
    public virtual void draw() {
         // some implementation
    }
    // more methods
}
public class Rectangle : Shape {
      // specific to the Rectangle data
      // override methods for the subclass
      public override void draw() {
          // method re-definition
      }
 }
public class Triangle : Shape {
      // specific to the Triangle data
      // override methods for the subclass
      public override void draw() {
          // method re-definition
      }
 }
```

Packages: Java Says "Package" and C# Says "Namespace"

In both cases, it is an additional dimension of encapsulation. In the same way a class defines and encapsulates an object type with its data and behavior, a package defines and encapsulates a set of classes, likely to be reused as a library, that provides a function in a specific area. For example, base language types and functionality are represented in the *java.lang package* in Java and the *System* package in C#.

Unlike C#, Java Relates Package Names with File System Directory Names

A package name in Java is the same as the name of the directory in which the classes of the package are located. The package structure in Java dictates the class file structure. In C#, *namespaces* may be located in any directory (folder), regardless of the name of the particular *namespace*.

Final (Not Modifiable): Java Says "Final" and C# Says "Sealed"

For example, Java may say "public final int b," whereas C# would say "public sealed int b." In both cases, the variable *b* that marked such a modifier will not be modified. It is "final" (I mean "sealed").

Java Says "Instanceof" and C# Says "Is"

In both cases, the meaning is precisely the same. For example, "if (b instanceof A)" is the Java way to determine whether the object named *b* is an instance of class *A*. The same question sounds a bit clearer in C#: "if (b is A)."

Java Says "Synchronized" and C# Says "Lock"

Actually, C# may also say "Synchronized," and in all cases, this is about making data "thread safe." This means locking related data so other threads cannot damage them until a particular task is over. Note that I actually use the term *lock* to describe this function. C# (as well as Microsoft in general) consistently looks for simplicity and an intuitive approach recognized by the user.

Java says "import" when the program uses additional libraries. C# prefers to grab the keyword *using,* which Larry Wall introduced in Perl to indicate that a source requires (uses) additional libraries to run. When a Java program says “import javax.xml.parsers.*,” C# would say something like “using System.Xml.”

C# has also kept keywords that are familiar to C++ programmers, such as *struct*, *stackalloc*, and *sizeof*. These keywords provide a bridge not only to C++ but also to C programmers, allowing them, for example, to create data structures inside and outside classes, and giving them more freedom in writing non–object-oriented code.

Java and C#, Unlike C++, Disallow Global Methods

All methods must belong to some class. It is harder but still achievable to write non–object-oriented spaghetti code in Java or C#, even though whatever one writes in Java or C#, it must be a set of classes.

There are textbooks that provide examples of Java or C# code encapsulated in a single ***main*** method that can sometimes be very long. Such examples still have a lot of value for an instructor in the second part of his or her presentation on “good and bad programming practices.”

A good object-oriented program describes object data at the beginning of the class definition; provides main behavior patterns in class methods that “get,” “set,” and change the data; and initiates data in their class constructors. The main method usually includes a couple of lines that create the main object and invoke one of its methods. For example, in Java we would write:

```
public static void main(String[] args) {
     A a = new A();
     a.go();
}
```

The C# source code looks very similar:

```
public static void Main(String[] args) {
     A a = new A();
     a.go();
}
```

The only difference is the main method name.

Java Says "main," C# Says "Main."

Method Naming Conventions Are Different for Java and C#.

Java style recommends that method names begin with lower-case letters. This is a part of the Java naming convention. The naming convention for C# method names is different from Java's, but (not surprisingly) the same as C++'s. Method names in C#/C++ start with a capital letter.

Java and C# Both Have the *String* Class with the Same Spelling and Behavior, But …

Java says "String." C# says "String," too. However, C# also allows us to use "string," beginning with lower-case *s*. In Java, we deal with the *java.lang.String* class, and in C#, the *System.String* class. In both cases, it is an immutable (nonchangeable) object. Each operation on a string creates a new string copy that is returned as a result of the operation.

Java and C# Both Say "Object," But C# Also Allows Us to Use "object"

We will discuss two major functions of the programming environment that supports the writing of application scenarios: handling XML and providing direct call connections to a knowledge engine and regular services with the reflection mechanism.

Different defaults for data and method access

Java and C# may be silent about data and method access, but their silent defaults are different. Here is a Java example:

```
package com.its.examples;
public class JavaExample {
    String text = "I am visible to everyone from my folder.";
    // more code
}
```

A class member with the default access (e.g., string text) is visible to all the classes located in the same package. Here is an example of access to the string text from another class in the same package:

```
package com.its.examples;
public class AnotherJavaExample {
    // method that access the text variable from the JavaExample class
    public void printThisText() {
        JavaExample example = new JavaExample();
        System.out.println(example.text); // will print the text
    }
}
```

Default access in C#, as well as C++, is *private*. If one omits an access modifier on a variable or a method in C#/C++, such a class member is considered private, visible only inside the same class. Here is a C# example:

```
namespase ITS.Examples;
using System;
public class CsExample {
    String text = "I am private!. Only members of this class can access
me.";
    // more code
```

```
    // the getString() publicmethod provides access to the private
(default) text
    public String getText() {
        return text;
    }
}
```

A class member with default access (e.g., string text) is visible to all classes located in the same package. Here is an example of access to the string text from another class.

```
namespaspase ITS.Examples;
using System;
public class AnotherCsExample {
    // method that access the text variable from the JavaExample class
    public void printThisText() {
        CsExample example = new CsExample();
        Console.WriteLine(example.getText()); // will print the text
    }
}
```

Java and C# both say “protected” to allow derived classes to access parent class members, but Java’s *protected* is more generic. Protected class members are visible in Java, not only to derived classes from any package but also to any class from the same package. C# is more specific and consistent than C++ in its definition of protected access. A protected member in C# can only be accessed by member methods in that class or member methods in derived classes, but is not accessible by any other classes.

C# has two more specific access modifiers that can open wider access for class members. The *internal* modifier opens access to a class member from other classes in the same assembly. What is assembly? If you are a Java programmer, think of assembly as a

Java Archive (JAR) file. The assembly is a set of classes usually stored as *.EXE or *.DLL files, unlike Java JARs that are stored in ZIP format.

C# also has the combined *internal protected* access modifier that makes a member visible to derived classes or classes that are in the same assembly.

Networking and File Access in Java and C#

Java, as well as C#, uses streams for data communications. Working with files or networks, we establish input and/or output streams between communication points. The endpoint of communications can be, for example, a file or a socket.

Read and Write Files in Java

Fig. A1.1 presents the *readBinFile()* method of the *JavaIOExample* class. The *readBinFile()* method reads a file from a file system or from the Internet. The method starts with a simple check to see if a file name is actually a URL. In this case, the *fetchURL()* method will be called to retrieve data from the Internet. Otherwise (the file name is not a URL), the method creates a file object based on the file name and uses the *length()* method of the *File* class to get the size of the file in bytes. Then, the *readBinFile()* method creates an input stream to the input file and the *while* loop reads the data into the *data* byte array.

[Fig.A1-1]

Fig. A1.2 shows the *writeBinFile()* method of the *JavaIOExample* class. The first method presented in the figure is just a convenience wrapper that can accept only two arguments: a file name and bytes to write. The real work is done in the following method that, besides the file name and the array of bytes, expects the offset and actual number of bytes of the array that will be written into an output file.

[Fig.A1-2]

Fig. A1.3 displays the *copyTextFile()* method, which reads an input file and immediately writes data into an output file.

[Fig.A1-3]

In all cases, we create a file object and establish a stream to the object. For example:

```
File file=new File(filename);

FileOutputStream out=new FileOutputStream(file);
```

We can streamline these two operations into one line:

```
FileOutputStream out=new FileOutputStream(new
File(filename));
```

Then, we use the stream object to read or write data. For example:

```
out.write(iData, iOffset, iSize);
```

Then, we eventually close the stream:

```
out.close();
```

Read and Write Files in C#

Fig. A1.4 demonstrates a C# example of copying an input file into an output file. The method *CopyTextFile()* of the *CsIOExample* class looks like a sibling of the method *copyTextFile()* of the *JavaIOExample.*

[Fig.A1-4]

We create an input file and output file objects. Then, we attach streams to the files, and we use a loop to read-write (copy) data.

You can see that the Java and C# source codes are very similar. In the C# code, we go through the same steps, and even the method names are almost the same.

Retrieve Data from the Web in Java

Fig. A1.5 presents the *fetchURL()* method of the *JavaIOExample* class. The *fetchURL()* method creates a *URLConnection* object based on the URL provided as an argument to the method. The endpoint is not a file but a *URLConnection*.

[Fig.A1-5]

The following steps are the same as those for reading files. Based on the endpoint (in this case, the *URLConnection* object), we create an input stream and read data from this stream using the *readFromStream()* helper method.

When we deal with a network object, it is harder to define the object size upfront as we did while reading file objects. Reading files, we were able to allocate a fixed-size byte array.

In the *readFromStream()* method we create a *ByteArrayOutputStream* instead and use this stream to write data directly into memory. This is a very convenient way to use memory to accumulate data.

The *while* loop that reads data from the Internet is finished when there is nothing more to read. At this point, we take the *tempBuffer* stream object in which we accumulated data and convert it into a regular byte array.

Done!

Retrieve Data from the Web in C#

Fig. A1.6 presents the *FetchURL()* method of the *CsIOExample* class. The *FetchURL()* method creates a *WebClient* object and uses the *DownloadData()* method of the *WebClient* class to download the data from the net based on the URL provided as an argument to the method.

[Fig.A1-6]

Java and C# Sockets

Server Socket Listener in Java

Fig. A1.7 presents an example of a server socket listener on the local network. This is a simple example of a server socket in which the server daemon is waiting for client requests and starts a service thread for each client. The constructor takes a port number argument and creates a server socket.

[Fig.A1-7]

The *run()* method starts the *while* loop listening for client service requests and starts a service thread for each client. The *setListening()* method is a convenient helper that can set the *listening* flag that serves as a condition for the *while* loop. The *main()* method gives the class the test; it creates a server socket daemon object on port number 11000 and starts the listening thread. Remember that according to Java specifications, *thread.start()* invokes the *run()* method of the thread.

The ServiceThreadExample serves an XML-based service request

Fig. A1.8 shows the *ServiceThreadExample*. This is a service thread example that serves network clients connected over TCP/IP sockets. The service thread takes a client socket as an argument in its constructor.

[Fig.A1-8]

The *run()* method of the *ServiceThreadExample* retrieves a service request and parameters from the input stream attached to the socket. Then, the *run()* method passes the client request to the *performService()* method of the *ServiceConnector* class. The clientRequest line may be present in XML format or as a method signature. Here is an example of the XML format:

```
 <act service="ITSeMailClient" action="sendMail"
  to="jeff.zhuk@javaschool.com" subject="hi!" body="How are you?" />
```

And an example of the method signature:

```
ITSeMailClient.sendMail("jeff.zhuk@javaschool.com","hi!.", "How are
you?");
```

The *performService()* method of the *ServiceConnector* class parses the client request string and performs exactly the same operation in both cases. Also in both cases, the *ITSeMailClient* class is loaded and its object instance is created. Then, the method *sendMail()* of this class is called with three parameters.

The *ServiceConnector* class has a registry of objects, which helps load new classes only for the first service call. Then, the same object may be reused for subsequent service requests. The *setPerform()*method is a convenient helper that can set the *perform* flag that serves as a condition for the *while* loop.

A Server Socket Listener in C#

Fig. A1.9 displays an example of a server socket listener in C#. The

CsServerSocketExample class is a simple server socket example. The constructor of the

CsServerSocketExample takes a port number argument and creates a server socket.

[Fig.A1-9]

The *Start()* method starts the *while* loop listening for client service requests and uses the *PerformService()* method of the *ServiceConnector* class to serve clients. (Wait for the Reflection topic to consider the *ServiceConnector* class in C#). A slight difference in handling data with C# sockets is that the *Receive()* method of the *Socket* class in C# creates streams internally, on the fly.

We use the *Encoding.ASCII.GetString()* method to convert the byte array into a string.

This example closes the socket after a single service request, which is completely optional. In some cases, it is preferable to continue client-server communications beyond a single service request.

The *Main()* method gives the class the test. The method creates a server socket daemon object on port number 11000 and invokes the *Start()* method with its *while* loop that listens to client requests.

As you can see, C# and Java socket handling is very similar.

A Client Socket Example in Java

Fig. A1.10 presents the *JavaClientSocket* class that can work with both server daemons: the *JavaServerSocketExample* implemented in Java and the *CsServerSocketExample* implemented in C#.

[Fig.A1-10]

The constructor establishes a socket connection to a server using two arguments: the host name and the port number.

The *getService()* method requests a service and returns a service response. The method takes a service request as a string argument and converts the string into a byte array. Then, the method sends the *serviceRequest* as the byte array to the server.

The *getService()* method reuses the *readFromStream()* method provided in the *JavaIOExample* class to receive the response from the server. The method closes the client socket streams; however, this action is optional and depends on the client-server protocol. In some cases, it is preferable to continue client-server communications beyond a single service request.

The *getService()* method receives the server response as an array of bytes and immediately converts the array into a string just to return the string to a calling procedure.

The *main()* method serves as the testing mechanism. We start the *main()* method by setting basic parameters, such as a host name, port number, and service request. The *main()* method creates the *JavaClientSocket* object and invokes the *getService()* method on this object. We end up with the *main()* method displaying the server response on the screen.

More Ways to Create Sockets in C#

C# offers several socket classes, so there is more than one way to create sockets in C#. For example, we can use the *TcpListener* class (part of the *System.Net.Sockets* namespace) to create a server listener:

```
                TcpListener tcpListener = new TcpListener(11000); //
port number
                tcpListener.Start();     // start listening!
                // Accept requested connections
                Socket clientSocketCounterpart =
tcpListener.AcceptSocket();
```

C# offers input and output stream classes, such as *NetworkStream*, *StreamWriter*, and *StreamReader*, for reading and writing data to and from sockets.

Here is an example of C# code establishing streams to read and write text data.

```
if (socketForClient.Connected) {
```

```
    // creat generic network stream a base for input/output
stream
    NetworkStream networkStream =
      new NetworkStream(clientSocketCounterpart);
    // create output stream
    StreamWriter streamWriter = new StreamWriter(networkStream);
    // create input stream
    StreamReader streamReader = new StreamReader(networkStream);
    // read text line from the network
    string line = streamReader.ReadLine();
     // write text line to the network
     streamWriter.WriteLine(line);
}
```

Throughout this book, we consistently used XML-based scenarios to invoke needed services. We also used them in the *JavaServerSocketExample*, as well as in the *CsServerSocketExample*. In both cases, we referred to the *ServerConnector* class that performed a needed service for us.

We need to find a proper class, discover the methods in the class, and according to an XML scenario, invoke the proper methods at run-time. This ability is called *reflection*. There are two packages, *java.lang.ref* and *java.lang.reflect*, that help us perform this magic in Java.

Perform Services Using Java Reflection

Fig. A1.11 shows the *performService()* and *getInstance()* methods of the *ServiceConnector* class which uses the power of Java Reflection.

[Fig.A1-11]

The *performService()* method determines whether the service request is an XML string. For example:

```
<act service="ITSeMailClient" action="sendMail"
   to="jeff.zhuk@javaschool.com" subject="hi!" body="How are
you?" />
```

In this case, the *performService()* method uses the XML parser to create a table of service parameters that includes a class name, a method name, and other relevant parameters. After retrieving service parameters from the XML string, the *performService()* method calls the *act()* method of the *ServiceConnector* class to actually perform the service. The *Stringer.parse()* method can be found in Appendix 3.

The service request can also have a form of method signature. For example:

```
ITSeMailClient.sendMail("jeff.zhuk@javaschool.com", "hi!.", "How
are you?");
```

Such a request is parsed by the *performAction()* method. In both cases, an instance of the *ServiceConnector* class is used to process the service request.

The *getInstance()* method of the *ServiceConnector* class is a static method. This method helps to create (if necessary) and support a single instance of the *ServiceConnector* class per application. The method implements the Singleton design pattern. The implementation is based on the *instance* static variable. The method creates this object if the object has not been created yet (object value equals null).

Why is it important to follow the Singleton design pattern and limit the service connector by a single instance? The service connector keeps service objects in the table (*Hashtable*) of acting objects. It is important to invoke methods on the same objects in their current state instead of creating new objects upon new service requests. Later in this appendix, we will see examples of service object reuse and service invocations.

Fig. A1.12 displays the *act()* method of the *ServiceConnector* class. The *act()* method uses a default acting class if a class name is not provided in the service parameters. The method checks to see if it is necessary to change the current acting object and changes it (if necessary) using the *changeActingClass()* method. Then, the *act()* method examines the parameter objects and tries to identify parameter types or classes using the *object.getClass()* method.

[Fig.A1-12]

Now we are armed with information about the acting object, acting method name, and method parameter types. This should be enough to find a proper method of the class using the powerful reflection method *getMethod()* of the *java.lang.Class* class.

Unfortunately, the *Class.getMethod()* has a problem: parameter types are often subclasses of required parameter classes. For example, the *readFromStream()* method of the *JavaIOExample* class provided in Fig. A1.6 expects the *InputStream* parameter. This method can still accept any of the subclasses of the *InputStream*, such as the *FileInputStream*, *ByteArrayInputStream*, and *FilterInputStream*.

Fig. A1.6 also shows the *fetchURL()* method that invokes the *readFromStream* method with the *DataInputStream* argument. This is perfectly alright for the run-time method invocation. Unfortunately, the *Class.getMethod()* can find the method only if we pass the exact parameter types. The *Class.getMethod()* throws an exception if we try to find the method that has the name *readFromStream* and may take the *DataInputStream* argument.

To work around this problem, I wrote another version of the *getMethod()* presented in Fig. A1.13. If the first attempt to find a proper method with the *Class.getMethod()* fails the second time, this version will be used instead.

[Fig.A1-13]

The *getMethod()* helps us work around a common problem in the *Class.getMethod()* that expects exact parameter type matches. Exact matches rarely happen in real programs. Parameter types are often subclasses of required argument classes.

The version of the *getMethod()* presented in Fig. A1. uses the more sensitive Java Reflection mechanism offered by the *Class.isAssignableFrom()* method while checking for method compatibility. The *Class.isAssignableFrom()* method returns true not only on the exact match of a class but also on its subclasses.

Perform Services Using Reflection in C#

C# is also very familiar with reflection features. The *System.Reflection* namespace provides classes such as *MethodInfo*, and *ParameterInfo*, similar to Java classes.

There is a slight difference between reflection in Java and in C#. Java can load the class file from a targeted classpath or resource file. The classpath may include JAR files that Java can uncompress on the fly.

Reflection in C# is provided at the assembly level, whereas reflection in Java is done at the class level. Assemblies are typically stored in DLLs or EXE files, and the programmer must know the proper file name, with the DLL or executable assembly, in which the class file is located.

Examples of the *ServiceConnector* class implementation in C# are presented in Figs. A1.14 through A1.16. The *ServiceConnector* class invokes a selected method on a selected class instance. The *ServiceConnector* can actually play not only its own (object) role but as a good actor, can also play objects of any (existing) type. If necessary, the *ServiceConnector* loads a new class at run-time. The *ServiceConnector* also has a registry in which it keeps (and reuses) all object-actors.

The beginning of the *ServiceConnector* class is shown in Fig. A1.14. The *ServiceConnector* class members include the *actingClass*, *actingObject*, and currently performed *method* objects. The table of *actingObjects (Hashtable)* serves as a registry to keep and reuse actor-objects that once were called onstage.

[Fig.A1-14]

The default constructor sets a current class and current object as current actors. The other constructor takes a class name as an argument and uses the *changeActingClass()* method to set this particular class as the current object-actor.

Fig. A1.15 shows the *act()* method of the *ServiceConnector* class. The *act()* method uses its arguments (class name, method name, and method parameters) to find a proper object and a method with the reflection mechanism and to invoke this method on the selected object.

[Fig.A1-15]

We find that the code in this section is very similar to that in Java implementation. Two key lines look exactly like Java lines, except for the method name capitalization.

```
// find the method
method = actingClass.GetMethod(methodName, classes);
// invoke the method
result = method.Invoke(actingObject, parameters);
```

Fig. A1.16 displays the rest of the *ServiceConnector* example in C#.

[Fig.A1-16]

The *changeActingClass()* method takes a class name as the argument and tries to bring an object of this type onto the stage. The method uses its arguments (class name, method name, and method parameters) to find the proper object and method with the reflection mechanism and to invoke this method on the selected object. If this does not return true, the *changeActingClass()* method looks into the table of actors, and if an object of the proper type is there, it just reuses this object.

Real work needs to be done if a fresh, new class type is needed to perform a service. This time, the magic of reflection does its best in the following two lines:

```
// load a class from an assembly at runtime
actingClass = Type.GetType(className);
// activate (instantiate) an object of the type
actingObject = Activator.CreateInstance( actingClass
);
```

The next thing the program does is register the object in the table of acting objects:

```
// register the object
actingObjects.Add(className, actingObject);
```

Handling XML Scenarios with C#

Microsoft .NET (besides being a marketing term) supports XML with a set of classes collected in several namespaces.

The *System.Xml* namespace contains major XML classes to read and write XML documents. There are four reader classes: *XmlReader, XmlTextReader*, *XmlValidatingReader*, and *XmlNodeReader*. There are also two writer classes – *XmlWriter* and *XmlTextWriter* – plus several more classes helping to navigate through nodes and perform other tasks.

The *System.Xml.Schema* namespace includes classes such as *XmlSchema*, *XmlSchemaAll*, *XmlSchemaXPath*, and *XmlSchemaType*, that work with XML schemas.

The *System.Xml.Serialization* namespace contains classes responsible for serialization of C# objects into XML documents.

The *System.Xml.XPath* namespace contains *XPathDocument*, *XPathExression*, *XPathNavigator*, and *XpathNodeIterator* classes that use *XPath* specification to navigate though XML documents.

The *System.Xml.Xsl* namespace is responsible for XSL/T transformations.

As you can see, the .NET side of the development community is well armed for XML processing. There is a great support in .NET (as well as in the Java world) for Simple API (application program interface) for XML (SAX) and Document Object Model (DOM) XML processing.

If we want to proceed with a thin solution (minimum package-namespaces), we can use a source similar to *Stringer.parse()* (see Appendix 3). We can even directly translate this Java code into C# using the Microsoft Java Language Conversion Assistant (JLCA) [1], which helps convert existing Java language source code into C#.

The *ParseXML.cpp* source written in C++ (see Appendix 3) may be considered for the limited resources of wireless devices.

Graphical User Interfaces in Java and C#

C# programs use MS Windows calls to create screen widgets. Windows Forms are a new style of MS Windows application built around the System.WinForms namespace. Windows programmers can use a unified Windows Forms API from any language (including C#) supported by MS Visual Studio .NET. While the Windows Forms solution supports a unified graphical user interface (GUI) API through multiple languages targeting a single (Windows) platform, Java focuses on multiple platforms that support the Java language.

The Java GUI is based on the Abstract Window Toolkit (java.awt) package that provides basic graphic abstractions and primitives. The java.awt package makes a unified GUI for all platforms and serves as a base for rich graphic components built with the set of Swing (javax.swing) packages.

An alternative GUI is provided in Java with the Standard Widget Toolkit (SWT). SWT uses a Java native interface (JNI) to C to invoke the native operating system graphics widgets from Java code. SWT enforces *one-to-one mapping* between Java native methods and operating system calls, which makes a very significant difference in the appearance and performance of graphics.

SWT is a pure Java code (although not from Sun Microsystems) widely accepted by the Java development community and supported by IBM and other organizations under the eclipse.org [2] umbrella.

We will briefly look into the most commonly used traditional Java graphics with java.awt (heavyweight, with native interface code) and javax.swing (lightweight, pure Java code that sits on the top of AWT) packages.

There are three basic elements in the java.awt package:

1. Graphic attributes, such as color and font, are defined.
2. All graphic components, such as buttons, lists, checkboxes, and other widgets, have graphic attributes. The crucial point is some components are actually containers that may contain other components.

3. Layout managers can easily control the complexity of different container-component layouts.

Fig. A1.17 provides a simple example of a Java applet. The Java applet uses its network capabilities to load an image from the network and draws this image along with several lines of Java poetry.

[Fig.A1-17]

A Web browser invokes the *init()* method as soon as the applet is loaded. Then, the browser refreshes the screen by calling the *paint()* method of the applet. More precisely, the browser calls the invisible *update()* method, which in turn, schedules the execution of the *paint()* method. These are the mechanics of painting Java components.

An applet or an application can draw different items on the java.awt.Component using the *paint(Graphics)* method. Images, lines, circles, polygons, or any other graphic items are drawn on a Graphics instance. The system (in the applet's case, it is the browser) passes the graphics instance to the *paint(Graphics)* method.

The *paint(Graphics)* method might be called several times throughout the life of an applet or application. The system allocates a thread and gives this thread a specific time in which to accomplish its painting job. If the drawing is complicated, this time frame may not be long enough and the *paint()* method may be called several times for a single screen refresh, which often causes a flickering effect.

When does the system call the *paint()* method? If an applet changes location on the screen or any time it needs to be refreshed (redrawn), *paint(Graphics)* is called. Programmers do not directly call *paint(Graphics)*. To have java.awt.Component paint itself, programmers call *repaint()*, which calls *update(Graphics)*, which in turn calls *paint(Graphics)*.

Be aware that *update(Graphics)* assumes that the component background is not clear, so it clears the background first. If you don't want to waste time clearing the background each time you want to paint, it's a good idea to override *update(Graphics)* with your own code that only calls the *paint*(*Graphics*) method. Fig. A1.18 projects the image created by the Web page that contains this applet.

[Fig.A1-18]

Commonly Used Layout Managers

The three most commonly used layout managers are the *FlowLayout*, *BorderLayout*, and *GridLayout*.

The *FlowLayout* is the default layout manager for the simplest Java container, the *Panel* class. The *FlowLayout* has three justifications: *FlowLayout.CENTER*, *FlowLayout.LEFT*, and *FlowLayout.RIGHT*. *FlowLayout.CENTER* is the default setting. Fig. A1.19 illustrates an example of the *FlowLayout* manager.

[Fig.A1-19]

The Java applet serves two roles: networking service and graphical component. The *Java Applet* class graphically represents java.awt.Panel. The default layout for the *Panel* is the *FlowLayout*.

The *FlowLayout* places components next to each other into one line. When the components fill the width of a container, the *FlowLayout* starts a new row. Fields in the *FlowLayout* keep their sizes (i.e., do not grow) when the user stretches the window.

Fig. A1.20 illustrates the use of the *BorderLayout*. The *BorderLayout* (default for the *Frame* and *Dialog* classes) has five regions: *North, South, East, West*, and *Center*. When a user resizes the window with the *BorderLayout*, the *North* and *South* regions grow horizontally while the *East* and *West* regions grow vertically. The *Center* region grows in both directions. It is **not** a requirement to fill all five fields of the *BorderLayout* (see Fig. A1.20).

[Fig.A1-20]

The sequences of the strings in Figs. A1.19 and A1.20 can be combined into four lines written by a Java student:

The training programs are just great!

I have elected one.

My valued time I dedicate

To Java from the Sun.

Fig. A1.21 displays an example of code that helps find the cheapest airfare from Denver to several destinations. This example greatly simplifies the backend operations required for such a search; it focuses on the graphics and event handling instead.

[Fig.A1-21]

The buttons are placed into the frame with the *GridLayout*. The number of regions for a *GridLayout* is specified in the constructor by rows and columns. In this case, we specified three rows with two columns for the layout. The *GridLayout* makes all the regions the same size, equal to the largest cell.

Creating a frame makes this applet less dependable on the Web browser. The frame stays on the screen while the user continues to browse other pages.

The *Tickets* class definition includes the promise to implement the *ActionListener* interface. This simply means that *ActionEvent* handling happens in *this* class and this class must provide the *actionPerformed()* method implementation.

When any of the destination buttons is pressed, a button label is changed to show the corresponding fixed price. Figs. A1.22 and A1.23 show this applet at work.

[Fig.A1-22]

[Fig.A1-23]

From a Hard-Coded to a Flexible Set of Reusable Services

Several simple changes can turn this hard-coded GUI example into a set of reusable services.

- First, we separate the applet code from the basic GUI that appears in the frame.
- Then, we pass service names and related actions in an XML-based scenario with a set of parameters.
- Finally, we use the *ServiceConnector.performService()* method to perform the necessary services.

Fig. A1.24 displays the *WebContainer* applet class. The *WebContainer* source code is very short. The single *init()* method retrieves the URL parameter from the applet tag on the Web page that points to the *WebContainer* applet.

[Fig.A1-24]

Here is an example of the applet tag:

```
<applet code="WebContainer" width="1" height="1">
  <parameter name="scenario" value="ServiceSetExample.xml" />
</applet>
```

The *init()* method uses the *JavaIOExample.fetchURL()* method to fetch the scenario file that must be located on the same server as the applet and the original Web page. This is one of the applet's restrictions: it can network only to its own server-host.

The last line of the *init()* method initializes the *ServiceFrame* object and passes to the object the scenario that was just retrieved two lines above. Fig. A1.25 displays an example of the scenario.

[Fig.A1-25]

The scenario includes two lines that define the dimensions of the grid layout for the *ServiceFrame* object with the number of rows and columns, and a set of services to perform. Each service has a service name and a service instruction that includes class name, method name, and parameters when needed.

The *ServiceFrame* class begins in Fig. A1.26. This class extends the *java.awt.Frame* and promises to implement the *ActionListener* interface.

[Fig.A1-26]

The constructor of the ServiceFrame uses the Stringer.parse() method to process the scenario string and transform the string into a table (Hashtable) of service parameters. The constructor creates a grid of service buttons using the number of rows and columns retrieved from the service parameters. A panel with the grid of service buttons is placed in the "Center" of the frame. We set the bottom of the frame (the "South") with the text area in which we expect to display service responses if any.

The set of services fills the *services* table that was prepared by the *Stringer.parse()* method. We create a set of buttons and label them according to the names provided in the *services* table.

The *ServiceFrame* resizes itself with the *pack()* method, which makes it necessarily small, makes its components visible, and makes it ready to rock and roll.

Fig. A1.27 shows the *actionPerformed()* method, which handles button clicks, and the *main()* method. The *actionPerformed()* method is an event handler that is called by the system whenever a button is pressed. Note that when we created the buttons, we added an action listener to every button, pointing to THIS object as event handler.

[Fig.A1-27]

When any button is pressed, the action event is generated by the system and passed to the *actionPerformed()* method. The very first thing we need to do is recognize which button was pressed. This is an easy task. The *getActionCommand()* method of the action event object returns us a button label. We use the label to retrieve the service instruction from the table of services using the label as the key.

The next line performs the service using the *ServiceConnector.performService()* method. Then, the program changes the background color of the pressed button and sends a service response to the text area.

The *main()* method can test the class performance. We provide a URL to an XML scenario as the argument in the *main()* method. The method reads the scenario and passes the XML string to the *ServiceFrame* object. The right scenario is all this class needs to perform.

Java GUI with Swing Classes: The ServiceBrowser Example

Java Swing classes offer a big set of rich graphical widgets with very powerful functionality. I provide only one example to demonstrate some of these features.

The *JEditorPane* class is the focus of the following example. The *JEditorPane* is a Web browser emulator. Yes! This single class is as capable of loading and showing Web pages as almost any real browser. The *JEditorPane* does not perform JavaScript functions, and it is very picky about HTML code correctness, unlike most browsers, which forgive Web designers for small errors.

The *JEditorPane* class can also display documents with rich text format (RTF) prepared by Microsoft Word, Open Office, or Adobe tools.

The code example displayed below in Fig.A1.28 Author: In Fig. 1.28?-thanks loads the Web page with images, text, and hyperlinks that look like regular HTML links. Some of these links are not usual. A regular Web browser cannot interpret them, but our code will exceed the browser's capacities. We write the link handler method, and we can greatly extend the browser's functionality.

[Fig.A1-28]

The *ServiceBrowser* class (Fig. 1.28) is an example of the *javax.swing.JEditorPane* in action. Like the *java.awt.Frame*, the *javax.swing.JFrame* is the main window in Swing GUI. The *ServiceBrowser* class extends the *JFrame* and implements the *javax.swing.event.HyperlinkListener*. This means that the class must include the *hyperlinkUpdate()* method to handle link events.

Two GUI components cover all our needs: the *JEditorPane*, in which we plan to display the documents, and the *JScrollPane*, which provides scrolling skills for our browser. We place the *JEditorPane* into the *JScrollPane*. Now they can work together as a team.

We add another *JEditorPane* component to the top of the window (North). This component contains a Web page with control-links.

The constructor takes the initial URLs as parameters and creates a GUI in several lines. We set the *JEditorPane* objects *display* and *controls*. Then we create a *ContentPane* object to hold all the *JFrame* components. We place the objects in the center and at the top of the content pane, respectively.

There are several lines of calculations in which we set the component sizes based on the client screen resolution surveyed by the *Toolkit.getDefaultToolkit().getScreenSize()* method.

The constructor uses the image URL to load the logo icon and set the icon on the frame, to the left of the frame title. The program sets the title of the frame to the initial URL of the page that will be displayed first. The link event handler will continue this tradition, providing the current URL as the frame title each time a new link is activated.

The constructor finishes its GUI work by enabling the link handler, setting the display and controls pages, and finally, making the window visible. Note that the single *setPage()* method of the *JEditorPane* loads the page from the Internet or a local computer and displays the page.

The last lines of the constructor register the main objects at the *ServiceConnector* registry. Later on, we will be able to order these objects to change their behavior by providing the proper link instructions. For example, with the instructions below, we can change the mode of the display window:

```
javax.swing.JEditorPane.setContentMode("text/plain")
```

The link with this instruction will look like this:

```
<a href=service://
javax.swing.JEditorPane.setContentMode("text/plain")>Plain Text</a>
```

Fig. A1.29 shows the link handler. The *hyperlinkUpdate()* method is the required implementation of the *HyperlinkListener* interface. The method considers the type of event first. The *HTMLFrameHyperlinkEvent* is a notification about the link action. We can use this notification to retrieve the document for some analysis, but this is not the primary function of this method.

[Fig.A1-29]

The primary function of the *handleEvent()* method is as a regular *HyperlinkEvent* that happens when a user clicks on a link. The program tries to retrieve a link URL and act upon the URL instructions. If this is a regular link that points to some web of local files, the program loads such a file and shows its content in the display window.

The most important case is when the instruction starts with the *service://* string. This means that this is not a regular URL schema like *http://*, *file://*, or *ftp://*. The *service://* is the beginning of our service instruction that can be interpreted and performed by the *ServiceConnector.performService()* method. The result of the operation may be a URL to a document, a formatted Web page, an RTF, or a plain text document. The program looks into all these cases and acts accordingly.

Whenever a user changes the page, the method stores the current URL and the previous URL, which might be used to support the *Back* function.

This is the core of the service browser. We can add more methods to the service browser to support the controls at the top of the frame. For example, we can have controls to set the display to editable mode. We can also add the *Back* control to the controls.html page and support this link with the *back()* method provided in Fig.30 below.

Fig. A1.30 displays these two helper methods and the *main()* method to test the browser. The *back()* method sets the page to a previous URL stored one click before. The *setEditable()* method changes the mode of the display from read-only to read-and-write, and back.

[Fig.A1-30]

Fig. A1.31 shows an example of the controls.html file. There are only three control-links provided in the example, but this page can be easily extended with links to provide specific services based, for example, on the Open Office package.

[Fig.A1-31]

The *ServiceBrowser*'s functionality is not limited by the functions embedded in the source code of this class. Unlike a lot of applications and browsers, the *ServiceBrowser* is just an engine able to perform any integration-ready service. The presentation layer of the *ServiceBrowser* is also not defined by the source code. The screens and controls are driven by HTML descriptions.

Imaging the XML browser driven by XML scenarios. The scenarios would describe GUI as well as program functionality. Describing GUI in XML is not a terribly new idea. Developers from Netscape (currently AOL) submitted their XUL (XML User interface Language) specification [3] via the Mozilla Organization.

XUL describes the typical dialog controls, as well as widgets such as toolbars, trees, progress bars, and menus. Unlike HTML, which defines a single-page document, XUL describes the contents of an entire window, which could contain several Web pages.

There are existing implementations, such as Thinlet [4], that use XUL to define GUI widgets. There is even the XUL visual editor program “Theodore” [5] by Wolf Paulus, which ensures the generation of valid XUL descriptors.

The language of XML application scenarios discussed in this book can complement and integrate GUI descriptions (done with XUL) with extended service definitions.

Building GUI in C# with Windows Forms

Windows Forms [6] are a new style of application built around classes in the .NET Framework class library's System.Windows.Forms namespace (much richer than the Windows API).

Windows Forms streamline the programming model, providing a consistent API. For example, Microsoft Foundation Classes or the Windows API does not allow you to apply a style that is meaningful only at creation time to an existing window. Windows Forms take care of this problem by quietly destroying the window and recreating it with the specified style.

Windows Forms leverage several technologies, including a common application framework and managed execution environment. Programmers can use Windows Forms while connecting to XML Web services and building rich, data-aware applications with any of the languages supporting .NET development tools.

So, what would be an appropriate definition for Windows Forms?

1. Windows GUI APIs?
2. A library of classes?
3. A set of tools?
4. All of the above?

If you picked the last item, you win!

It is not necessary to install the full .NET environment to develop the Windows Forms GUI, but you need at least the .NET Framework distribution.

What is the programming model of Windows Forms? Traditionally, starting from Visual Basic, the term *forms* means top-level rectangular windows that can serve as dialog boxes, standard windows, or multiple document interface (MDI) windows.

Forms can present information for a user and accept the user's data. Forms are objects, instances of classes that define their properties and methods. The properties are translated at run-time into visible attributes, and the methods define user interface behavior.

Programmers prefer to use Windows Forms Designer to create and modify forms, although Code Editor lets you enjoy manual code writing.

The main starting points for writing GUI applications with Windows Forms are two classes of the System.Windows.Forms namespace: the *System.Windows.Forms.Application* and the *System.Windows.Forms.Form.* The static method *Run* of the *System.Windows.Forms.Application* displays a window and runs message events loop.

The *System.Windows.Forms.Form* class defines the window properties and behavior. This component (similar to the *java.awt.Component* class) handles user interface with virtual methods such as *OnPaint*, *OnMouseDown*, *OnKeyDown*, and *OnClosing*. Programmers write their code to override these default methods (that in most cases are empty placeholders) and provide actual handlers.

The next layer of Windows Forms classes represents numerous controls such as: System.Windows.Forms.Menu

- System.Windows.Forms.Button
- System.Windows.Forms.TextBox
- System.Windows.Forms.ListView
- System.Windows.Forms.FileDialog
- System.Windows.Forms.MonthCalendar
- System.Windows.Forms.ColorDialog
- System.Windows.Forms.PrintDialog
- System.Windows.Forms.FontDialog, etc.

Similar controls (although not all) can be found in the java.awt or javax.swing packages. There are several controls in Windows Forms that I miss in Java. For example, the System.Windows.Forms.LinkLabel is a hyperlink control. You can use the rich facilities of javax.swing.JEditorPane (see the *ServiceBrowser*, Figs. A1.27 and Fig. A1.28), or you can write your own class (I did it once then found out that M. Berthou created one [7]) to add this control to the java.awt building blocks.

Fig. A1.32 shows a simple example of a Windows Forms application. You can save this code in any file – for example, *example1.cs* – and compile it with the command line in MS DOS windows.

```
csc example1.cs
```

[Fig.A1-32]

You will find a new file, *example1.exe,* in the same folder. This is your Windows Forms application. Double click on the file and see a window with the proud title "Windows Forms Example."

You can add more components to this application. For example: You wamt to add a button with the label *Submit*. You create the button and define its properties. Then, you can add the button to other controls on the form:

```
System.Windows.Forms.Button submitButton = new
System.Windows.Forms.Button();
submitButton.Name = "submitButton"; // reference name
// place the button at this location on the form
submitButton.Location = new System.Drawing.Point(20, 40);
// define a style of the button
submitButton.FlatStyle = System.Windows.Forms.FlatStyle.System;
// provide button’s size, make it well visible (can resize itself by
default)
submitButton.Size = new System.Drawing.Size(100, 30);
// here is the label on the button
submitButton.Text = "Submit";
// Add to the form
this.Controls.Add(submitButton);
```

This code looks very similar to Java code, but it is not exactly the same.

Unified Terms to Define GUI: XUL and Views XML

The XML-based User Interface is a good candidate for such direction. Windows Forms deliver unified GUI terms for Microsoft languages. Providing XML-based APIs to Windows Forms can produce a great cocktail that even comes in a beautiful glass.

A group of developers from the University of Pretoria and the University of Victoria (sponsored by Microsoft Research) is working on the VIEWS (Vendor Independent Event and Windowing Systems) project [8], which deserves a special attention.

The group created the Views XML specification 1.0 (similar to XUL) to describe Windows Forms in XML terms. The Views XML specification was designed with emphasis on the user's convenience in writing XML descriptions. The specification, for example, does not require double quotes around attribute values and is not case sensitive regarding tags and attribute names.

The group provided an implementation of the concept in both C# and Java with the class name *View*. (You may find the *View* class is similar in some way to the Java *Thinlet* implementation). The *View* class constructor takes a long string of user interface definition in XML format as an argument and creates the GUI based on this definition.

The C# version of the *View* class uses Windows Forms supported by the .NET Framework. The Java version is implemented with Swing in SDK 1.4.

Fig. A1.33 shows an example of creating an object of the *View* class, which accepts an XML-based screen description. The XML string describes the screen in a manner similar to HTML: from the top down and from left to right.

[Fig.A1-33]

The "vertical" tags encapsulate several items or several groups of items from the top. The "horizontal" tags describe several items or several groups of items on the same vertical level. Fig. A1.34 shows the screen displayed by the *View* object regardless of Java or C# implementation.

[Fig.A1-34]

Programmers can access information within the controls and handle events with C# or Java code using *View* object methods, such as *GetText()*.

JDK 1.5 Makes Programming Simpler

Comparing Java to C Sharp we found C Sharp more liberal on several occasions. JDK 1.5 known as “Tiger” makes a step in the right direction to easy some rules. [1]

JDK1.5 makes easy strict rules that distinguish objects from primitives. You remember that all Collections like ArrayList, Hashtable, and etc. can only deal with objects.

We could not store a primitive value in collections without wrapping this value in the object.

For example:

Vector v = new Vector();
v.add(5); // JDK1.4 would give a compiler ERROR because “5” is not an object but an integer

You have to wrap this primitive value into an Integer object.

Vector v = new Vector();
Integer wrapper = new Integer(5);
v.add(wrapper);

Then, you have another problem when you want to retrieve your primitive value back from the collections.

int primitive = v.elementAt(0); // JDK1.4 would give you a compiler ERROR

You would need to add couple more lines to satisfy JDK1.4.

Integer wrapper = (Integer) v.elementAt(0);
int primitive = wrapper.intValue();

JDK1.5 removed the problem by introducing *autoboxing* and *unboxing* concepts when compiler takes care about casting and wrapping primitives to objects.

Vector v = new Vector();
v.addElement(5); // is OK!!!

……

int primitive = v.elementAt(0); // OK!!!

Java 1.5 introduced a new base type "*enum*" similar to the C/C++ *enum* that supports Typesafe *enum* [2] design pattern. This new keyword allows us to define a class that represents a single element of the enumerated type without providing a public constructor for the class.

Although *enum* represents integer values it is more informative as we operate with names. They (*enum*) are still classes and objects; we can add to them fields and methods and put them into collections.

In the example below 12 months are presented with the *enum* type.

```
public enum Months   {
  // The constant declarations below invoke a constructor
  // The constructor like January(1) passes the month number
  January(1), February(2), March(3), April(4), May(5), June(6), July(7), August(8),
September(9), October(10), November(11), December(12);

    private final int month;
    Months (int month) {
       this.month = month;
    }
    public int getMonth() {
        return month;
    }
 }
```

In the other example we define four seasons (plus Rainy Season that acts as default in some places). We do not bother ourselves with numbering the seasons. The compiler will do the job. The next element's value is always bigger than the value of the previous element. You can see a great feature that shows superiority of Java *enum* in comparison to its C/C++ siblings. We can use Java *enum* in switch statements. (Remember that switch statements only take integers. Java compiler is smart enough to represent *enum* as integer when necessary.

```
 // The Seasons are conveniently defined with no constructors
  public enum Seasons {Winter, Spring, Summer, Fall, RainySeason}

 // Note that unlike its C/C++ siblings Java enum can be used in switch statements
  public static Seasons getMonthLength (Months month) {
    switch (month) {
      case Months.December:
      case Months.January:
      case Months.February: return Seasons.Winter;

      case Months.March:
      case Months.April:
      case Months.May: return Seasons.Spring;
```

```
        case Months.June:
        case Months.July:
        case Months.August: return Seasons.Summer;

        case Months.September:
        case Months.October:
        case Months.November: return Seasons.Fall;

        default: return Seasons. RainySeason;
        }
      }
   }
```

One of the greatest features added by the JDK1.5 is called Generics. What it is all about? There are several sides of generics but all sides look attractive to us, developers, as they make our code more robust and even simpler.

Here is the example from the past (JDK1.4).

```
// count characters in the collection of strings
public int countCharacters(Collection words) {
   int counter = 0;
   for (Iterator i = words.iterator(); i.hasNext(); ) {
      String s = (String) i.next();
      counter += s.length();
   }
   return counter;
}
```

The code will compile OK but at run-time can crash if the collection passed to the method above is not the collection of strings.

Here is the code with that can be provided for JDK1.5. The code below includes generics. It takes full control on situation and even looks shorter!

```
// Generics ensure that a passed collection must be a collection of strings.
// If we try to pass to this method a different type of collection –
// We face a compiler ERROR instead – this is much better than run-time exception
public int countCharacters(Collection<String> words) {
   int counter = 0;
   for (Iterator<String> i = words.iterator(); i.hasNext(); ) {
       counter += i.next().length();
   }
   return counter;
}
```

Even this beautiful source can be enhanced with one more JDK1.5 feature that is named as “enhanced for”. This feature makes easier writing *for* loops. It is very similar to Perl’s *foreach* keyword. We basically say “for all elements in the collection” and we can skip the iteration line. Here is the source example.

```
// Note the “:” character in the enhanced for loop
public int countCharacters(Collection<String> words) {
   int counter = 0;
   for (String s : words) { // read it as “for all string s in collection of words”
       counter += s.length();
   }
   return counter;
}
```

Summary

This appendix is a reference for Java and C# programming. Every keyword and every concept is illustrated with one or more examples. The focus of this reference is on object-oriented concepts, reflection, and the approach to GUI development.

The target audience includes (but is not limited to) students and programmers who would like to follow an integration-ready strategy and recreate and extend “integrated with knowledge” systems that can run application scenarios in Java and/or C#. These systems enable invocation of existing services, simplify service descriptions, decrease coding efforts, and integrate parts of application definitions into XML-based application scenarios.

The application scenarios explained in this book extend XML-based GUI descriptions with references to grammar rules and direct access to services (not necessarily Web services, but any available services), including knowledge engine queries and assertions.

Exercises

1. Add several lines of Java code that will allow you to close the frame in Fig. A1.21. (Hint: the class should implement *WindowListener*.)
2. Research and list the similarities and differences between the Thinlet and VIEWS projects.

References

1. The Microsoft Java Language Conversion Assistant: *http://msdn.microsoft.com/vstudio/downloads/tools/jlca/default.asp*.
2. Eclipse.org: *http://www.eclipse.org*.
3. XML User Interface Language 1.0: *http://www.mozilla.org/projects/xul/xul.html.*
4. Thinlet: *http://Thinlet.com.*
5. The Theodore XUL Editor: *http://www.carlsbadcubes.com/theodore/*.
6. Windows Forms: *http://msdn.microsoft.com/vstudio/techinfo/articles/clients/winforms.asp*.
7. Berthou, M. The linkLabel.java: *http://www.javaside.com/zip/srcs/linkLabel_java.html*.
8. VIEWS or XGUI: *http://www.cs.up.ac.za/~jbishop/rotor/*.