Java 5 catches up with C#

Barry Cornelius
Computing Services, University of Oxford
Date 1st December 2005; first created: 7th August 2004
http://www.barrycornelius.com/papers/
mailto:barry.cornelius@oucs.ox.ac.uk

1 Introduction

Java 5 (also known as *Tiger*, JDK 1.5 and J2SE 5.0) was released during September 2004.

Unusually, with this release changes were made to the language as well as to the APIs. So Java 5 has a number of new features that will affect the way in which we write Java programs.

The new feature that has received most publicity is the use of generic Collection classes, but there is much more that is new in Java 5.

Although Java 5 introduces many new APIs, this document concentrates on the changes that have been made to the language. As well as generics, it looks at keyboard input, the enhanced for statement, covariant return types, autoboxing, enumeration types and annotations.

See [21], [25], [26], [4], [14], [15], [3] and [13] for other presentations about Java 5.

2 The evolution of Java

Java came into being in 1995. During the last ten years, although there has been a substantial development of new APIs, there have been few changes to the language itself.

Back in April 1999, Sun Microsystems made a suggestion that Java should be altered to include *generic types*. This proposal was then considered using the *Java Community Process*. Although that proposal went to public review in May 2001 it is only now with the release of Java 5 that it is officially part of the language.

The first release of Microsoft's C# did not include generics. However, these are included in *C# 2.0*.

In the early days of Java, Microsoft provided support for Java on the Windows platform. But when Microsoft started to add new constructs to its Java and to leave some APIs out, a legal battle started between Sun and Microsoft. Since then, Microsoft have moved away from Java, and in June 2000 Microsoft produced their own rival language, the language C#.

You can view C# as Microsoft's answer to Java. Their language is significantly similar to Java: however, it does have many features that distinguish it from Java. Examples are: properties, operator overloading and delegates.

But Java 5 is not just about generics: with the release of Java 5, a lot of other significant changes have been made to the language. In particular, Java 5 includes:

- an enhanced for statement;
- methods with a variable number of parameters;
- autoboxing and auto-unboxing;
- enumeration types;
- annotating code with metadata.

An interesting point is: all of these features are already in C#.

Making significant changes to a programming language should be done infrequently. Major changes to the FORTRAN language took place in 1966, 1977, 1990, 1995 and 2004+. It has been ten years since Java was first released: so maybe we were due for a big change. However, in this case, in my opinion, it is C# that has caused these radical changes to be made to the Java language.

3 Using the `Scanner` class for keyboard input

3.1 The problem

Although the use of a GUI is the more interesting way to get input from a user, books that teach programming usually avoid this to begin with. Instead of creating a window displaying a textbox and processing the characters that are typed into the textbox, to begin with a book will usually use some statement that simply reads a value from the keyboard attached to a command shell window.

This approach is adopted for two reasons:

- the number of new ideas involved is smaller;
- the amount of code needed is smaller.

However, although this is the case with many programming languages, until now it has not been easy to read a value from the keyboard in Java.

My book on Java ('Understanding Java' [6]) tries to introduce keyboard input as early as possible. The technical preface to the book says 'the horrors of Java's keyboard input are introduced early (Chapter 4):

```
0008:       BufferedReader tKeyboard =
0009:             new BufferedReader(new InputStreamReader(System.in));
0010:       System.out.print("Type in your first number: ");
0011:       String tFirstString = tKeyboard.readLine();
0012:       double tFirst = Double.parseDouble(tFirstString);
```

Note the use of both instance methods and class methods and the need to use the wrapper classes of the primitive types.' Whoops, there is another bit of hassle: I failed to mention the need for a throws clause for `java.io.IOException`:

```
0006:    public static void main(String[] pArgs) throws IOException
```

So the code is horrible and there are a lot of new ideas to explain.

Often a book will introduce its own class for keyboard input in order to avoid these complications. There are reasons why I think this is not a good idea. However, we need not go into them here, because Java 5 makes it easy to read from the keyboard.

3.2 Simple uses of the `Scanner` class

Java 5 provides a class called `Scanner` (in the `java.util` package) that can be used for keyboard input. It allows you to write code like:

```
0026:       Scanner tScanner = new Scanner(System.in);
0027:       System.out.print("Type in your first number: ");
0028:       double tFirst = tScanner.nextDouble();
```

Again, you do not have to handle `java.io.IOException`.

Here is a complete program using this class:

```
0021: import java.util.Scanner;                                    // SumProgNew.java
0022: public class SumProgNew
0023: {
0024:    public static void main(String[] pArgs)
0025:    {
0026:       Scanner tScanner = new Scanner(System.in);
0027:       System.out.print("Type in your first number: ");
0028:       double tFirst = tScanner.nextDouble();
0029:       System.out.print("Type in your second number: ");
0030:       double tSecond = tScanner.nextDouble();
0031:       double tSum = tFirst + tSecond;
0032:       System.out.println("The sum of " + tFirst +
0033:                          " and " + tSecond + " is " + tSum);
0034:    }
0035: }
```

This program produces a dialogue like the following:

```
0036: Type in your first number: 27
0037: Type in your second number: 42
0038: The sum of 27.0 and 42.0 is 69.0
```

3.3 Other details about the `Scanner` class

The class `Scanner` allows you easily to associate a variable with one of a number of different input streams such as the keyboard, a file or a string:

```
0047:       Scanner tScannerKeyboard   = new Scanner(System.in);
0048:       Scanner tScannerInputFile  = new Scanner(new File("input.txt"));
0049:       Scanner tScannerSomeString = new Scanner(tSomeString);
```

You can then apply methods such as `nextInt` to read a value from the input stream. Here, an unchecked exception `java.util.InputMisMatchException` will be thrown if the next token is not an int.

To avoid the possibility of this exception occurring, a method like `hasNextInt` can be used to determine if the next value is an int, as shown in this program:

```
0067: import java.util.Scanner;                                    // Analyse.java
0068: public class Analyse
0069: {
0070:    public static void main(String[] pArgs)
0071:    {
0072:       Scanner tScanner = new Scanner(System.in);
0073:       System.out.print("Type in a value: ");
0074:       if (tScanner.hasNextInt()) {
0075:          int tFirst = tScanner.nextInt();
0076:          System.out.println("Got an int: " + tFirst);
0077:       }
0078:       else if (tScanner.hasNextDouble()) {
0079:          double tFirst = tScanner.nextDouble();
0080:          System.out.println("Got a double: " + tFirst);
0081:       }
0082:       else {
0083:          String tFirst = tScanner.nextLine();
0084:          System.out.println("Got a string: " + tFirst);
0085:       }
0086:    }
0087: }
```

3.4 Reading more than one value from a line

The `Scanner` class also helps when you want the user to be able to type more than one value on a line. Suppose we want to read in an age, a height and the number of children. So the data might be something like:

```
42 1.85 2
```

Before Java 5, you would need to use something like `StringTokenizer` to break apart a line into separate items (often called *tokens*):

```
0100:       String tLine = tKeyboard.readLine();
0101:       StringTokenizer tTokens = new StringTokenizer(tLine);
0102:       String tAgeString = tTokens.nextToken();
0103:       int tAge = Integer.parseInt(tAgeString);
0104:       String tHeightString = tTokens.nextToken();
0105:       double tHeight = Double.parseDouble(tHeightString);
0106:       String tNumberOfChildrenString = tTokens.nextToken();
0107:       int tNumberOfChildren = Integer.parseInt(tNumberOfChildrenString);
```

However, the `nextXXX` methods of the `Scanner` class can cope with several items being on the same line. So the data:

```
42 1.85 2
```

can be read by the following code:

```
0120:       int tAge = tScanner.nextInt();
0121:       double tHeight = tScanner.nextDouble();
0122:       int tNumberOfChildren = tScanner.nextInt();
```

By default, tokens are assumed to be separated by whitespace. However, you can easily arrange for the scanner to use delimiters other than whitespace.

3.5 More about the `Scanner` class

The API documentation for the `Scanner` class ([32]) contains a lot of easy-to-understand useful detail.

4 Enhanced for statement

4.1 Details about the change

Suppose we have an array such as:

```
0132:       int[ ] tMonthStarts =
0133:             { 1,32,60,91,121,152,182,213,244,274,305,335 };
0134:       System.out.println(Arrays.toString(tMonthStarts));
```

One pattern of code we will frequently write involves looking at each element of the array:

```
0136:       for (int tMonthNumber = 0;
0137:                tMonthNumber<tMonthStarts.length; tMonthNumber++)
0138:       {
0139:          System.out.println(tMonthStarts[tMonthNumber]);
0140:       }
```

Java 5 has an enhanced for statement that enables us to write this more succinctly:

```
0141:       for (int tMonthStart : tMonthStarts)
0142:       {
0143:          System.out.println(tMonthStart);
0144:       }
```

This variant of the for statement is sometimes called a *foreach loop* or a *for-in loop*. The designers were not very adventurous with the syntax: they did not add any new keywords as they were worried that existing code might already be using `foreach` or `in` as identifiers.

4.2 When can you use it?

So the new variant has the form:

```
for ( decl : expr )
```

where `decl` is a declaration and `expr` is an expression that delivers a value of the type `Iterable`. This type is new. It is an interface type declared as follows:

```
0148: package java.lang;                                        // Iterable.java
0149: import java.util.Iterator;
0150: public interface Iterable
0151: {
0152:    public Iterator iterator();
0153: }
```

Obviously, from the above example, arrays implement this interface. As we shall see later, we can also use foreach loops with the Collection classes as they also implement this interface. And if we provide our own class that implements the `Iterable` interface, we could also use a foreach loop with an object of that class.

There are many situations where the traditional code for looking at each element will still have to be used, e.g., if you wish to know the element number containing the largest value of an array.

5 Autoboxing and auto-unboxing

5.1 Using an `ArrayList`

Java has a number of Collection classes. One of these is `ArrayList` which allows you to represent an *array* whose size grows as you add more objects to it. There are a few classes similar to `ArrayList` all of which satisfy the `List` interface.

So, here is how you might add an element to an `ArrayList` and then retrieve it:

```
0161:       List tList = new ArrayList();
0162:       tList.add("Hello world");
0163:       String tNewString = (String)tList.get(0);
0164:       System.out.println(tNewString.substring(0, 5));
```

Here, a `String` is being stored in the first element of this `ArrayList`.

You can store objects of any class in an `ArrayList`. Usually, the objects you store in an `ArrayList` will all be of the same class: this is called a *homogeneous list*. However, occasionally you will want a mix. So let us add a `Date` object to this `ArrayList`:

```
0165:       Date tDate = new Date();
0166:       tList.add(tDate);
0167:       Date tNewDate = (Date)tList.get(1);
0168:       System.out.println(tNewDate.getMonth());
```

This can be done because `List`'s `add` method has a parameter that is of type `Object`. Note that the return type of `get` is also of type `Object`, and it is for this reason that the above pieces of code contain casts:

```
0163:       String tNewString = (String)tList.get(0);

0167:       Date tNewDate = (Date)tList.get(1);
```

5.2 Storing an `int` in an `ArrayList`

What happens if you want to store integer values in an `ArrayList`? Well, values of type `int` are not objects, and so, with Java 1.4 (and earlier), it is a little more tricky to store `int`s in a `List`. What you can do is to create an object of the class type `java.lang.Integer` from an `int` and use that. This process is known as *boxing*:

```
0169:       int tNumber = 123;
0170:       Integer tWrapper = new Integer(tNumber);
0171:       tList.add(tWrapper);
```

The same sort of shenanigans take place on retrieval, when *unboxing* takes place:

```
0172:       Integer tNewWrapper = (Integer)tList.get(2);
0173:       int tNewNumber = tNewWrapper.intValue();
0174:       System.out.println(tNewNumber);
```

You can do similar things for any of the primitive types, because each primitive type (e.g., `int`) has an associated class type called its *wrapper type* (e.g., `java.lang.Integer`).

5.3 Java 5 provides autoboxing and auto-unboxing

It is a tedious to have to write this code. And with Java 5 we do not have to: it permits us to write:

```
0175:       int tValue = 123;
0176:       tList.add(tValue);
0177:       int tNewValue = (Integer)tList.get(3);
0178:       System.out.println(tNewValue);
```

Behind the scenes, the boxing and unboxing still take place. So this is called *autoboxing* and *auto-unboxing*.

5.4 Avoiding overuse of autoboxing and auto-unboxing

It is probably not a good idea to overuse this new freedom. Consider the following program, and work out how much boxing/unboxing is happening:

```
0182: public class Integers                                    // Integers.java
0183: {
0184:    public static void main(String[] pArgs)
0185:    {
0186:       Integer tWrapper = 127;
0187:       Integer tNewWrapper = tWrapper + tWrapper;
0188:       System.out.println(tNewWrapper);
```

```
0189:       if (tWrapper<tNewWrapper)
0190:       {
0191:          System.out.println(tWrapper + " is less than " + tNewWrapper);
0192:       }
0193:       tNewWrapper = 127;
0194:       if (tWrapper==tNewWrapper)
0195:          System.out.println(tWrapper + " is equal to " + tNewWrapper);
0196:       else
0197:          System.out.println(tWrapper + " differs from " + tNewWrapper);
0198:       tWrapper = 128;
0199:       tNewWrapper = 128;
0200:       if (tWrapper==tNewWrapper)
0201:          System.out.println(tWrapper + " is equal to " + tNewWrapper);
0202:       else
0203:          System.out.println(tWrapper + " differs from " + tNewWrapper);
0204:    }
0205: }
```

The above program produces the following output:

```
0206: 254
0207: 127 is less than 254
0208: 127 is equal to 127
0209: 128 differs from 128
```

The last two lines of output need some explanation. Brett McLaughlin and David Flanagan have produced an excellent book on Java 5 ([21]). In it, they say that 'the Java specification indicates that certain primitives are always to be boxed into the same immutable wrapper objects. These objects are then cached and reused, with the expectation that these are commonly used objects. These special values are the `boolean` values `true` and `false`, all `byte` values, `short` and `int` values between -128 and 127, and any `char` in the range `\u0000` to `\u007F`.' So you may get surprising results from using the equality operator. McLaughlin and Flanagan conclude: 'you have to watch out for this, as it can result in some very tricky, hard-to-find bugs'.

6 Methods with a variable number of parameters

6.1 The problem

Suppose we want a method that performs some operation on a collection of values. We could get the caller of the method to store these values in an array and get that passed as an argument to the method. However, suppose instead we want to write a method that can handle any number of arguments.

This is possible in Java 5.

6.2 The details of the change

Suppose there is a method called `iEvaluate` that expects as arguments a `double` and any number of `int`s. Here are some examples of a call of this method:

```
0214:       double tResult = iEvaluate(2.7, 25, 2, -5, 42, -10);
0215:       System.out.println(tResult);
0216:       tResult = iEvaluate(4.2, 42);
0217:       System.out.println(tResult);
0218:       tResult = iEvaluate(4.2);
0219:       System.out.println(tResult);
```

And here is a possible `iEvaluate` method:

```
0221:    private static double iEvaluate(double pFactor, int... pValues)
0222:    {
0223:       int tSum = 0;
0224:       for (int tValue : pValues)
0225:          tSum += tValue;
0226:       return tSum/pFactor;
0227:    }
```

The `int...` signifies that the remaining arguments are `int`s. Within the method, this parameter behaves like an array and so the method can use a foreach loop.

7 The printf method

7.1 A new class for formatted printing

Java 5 introduces a new class called `java.util.Formatter`. The API documentation for this class is at [31]. It says: 'Formatted printing for the Java language is heavily inspired by C's printf. Although the format strings

are similar to C, some customizations have been made to accommodate the Java language and exploit some of its features. Also, Java formatting is more strict than C's; for example, if a conversion is incompatible with a flag, an exception will be thrown. In C inapplicable flags are silently ignored. The format strings are thus intended to be recognizable to C programmers but not necessarily completely compatible with those in C.'

7.2 Some examples of the use of `printf`

Here are some examples of the use of `printf`:

```
0242:       System.out.printf("%3d %1.2f%n%d%n",
0243:                         tAge, tHeight, tNumberOfChildren);
0244:       String tFormat =
0245:             "Age is %d, height is %f%nNo. of children is %d%n";
0246:       System.out.printf(tFormat, tAge, tHeight, tNumberOfChildren);
0247:       Calendar tNow = Calendar.getInstance();
0248:       System.out.printf("%tT%n", tNow);
0249:       System.out.printf("%tY-%tm-%td%n", tNow, tNow, tNow);
0250:       System.out.printf("%tH:%<tM:%<tS%n", tNow);
```

The above code produces output like the following:

```
0253:  42 1.85
0254: 2
0255: Age is 42, height is 1.850000
0256: No. of children is 2
0257: 08:03:39
0258: 2004-08-12
0259: 08:03:39
```

The `printf` method used above is from the `java.io.PrintStream` package.

8 Methods with covariant return types

8.1 The problem

Suppose we have a class that can represent two-dimensional shapes:

```
0260: public class Shape                                          // Shape.java
0261: {
0262:    private int iX, iY;
0263:    public Shape(int pX, int pY)
0264:    {
0265:       iX = pX; iY = pY;
0266:    }
0267:    public int getX()
0268:    {
0269:       return iX;
0270:    }
0271:    public int getY()
0272:    {
0273:       return iY;
0274:    }
0275:    public Shape translate(int pX, int pY)
0276:    {
0277:       return new Shape(iX + pX, iY + pY);
0278:    }
0279:    public boolean equals(Object pObject)
0280:    {
0281:       return iX==((Shape)pObject).iX && iY==((Shape)pObject).iY;
0282:    }
0283:    public String toString()
0284:    {
0285:       return iX + ":" + iY;
0286:    }
0287: }
```

You will see that this class has no `setXXX` methods. So, when an object of this class has been created, it cannot then be changed. The objects of this class are said to be *immutable*. (This is also true for objects of some of the classes of Java's APIs such as the class `java.lang.String`.)

However, the class does provide a method called `translate`. When applied to some Shape, it produces a new Shape that is (pX,pY) away from the first Shape:

```
Shape tShape = new Shape(100, 200);
Shape tNewShape = tShape.translate(1, 2);
```

Suppose we now want a type for a special two-dimensional shape that is a circle. We could represent circles by using a class called `Circle` that is derived from `Shape`. Besides the `iX` and `iY` fields that `Shape`s have, `Circle`s also need an `iRadius` field:

```
0288: public class Circle extends Shape                              // Circle.java
0289: {
0290:    private int iRadius;
0291:    public Circle(int pX, int pY, int pRadius) {
0292:       super(pX, pY);
0293:       iRadius = pRadius;
0294:    }
0295:    public int getRadius()
0296:    {
0297:       return iRadius;
0298:    }
0299:    public Shape translate(int pX, int pY)
0300:    {
0301:       return new Circle(getX() + pX, getY() + pY, iRadius);
0302:    }
0303:    public boolean equals(Object pObject)
0304:    {
0305:       return super.equals(pObject) &&
0306:                 iRadius==((Circle)pObject).iRadius;
0307:    }
0308:    public String toString()
0309:    {
0310:       return super.toString() + ":" + iRadius;
0311:    }
0312: }
```

Although `Circle`'s `translate` method creates a `Circle` object, in Java 1.4 (and earlier) the method has to have a return type of `Shape`:

```
0299:    public Shape translate(int pX, int pY)
```

Because this `translate` method has a return type of `Shape`, a client of `Circle` will have to use a cast if it wants to get a `Circle` from a call of this method:

```
0331:               Circle tCircle = new Circle(x, y, radius);
0332:               Circle tNewCircle = (Circle)tCircle.translate(1, 2);
0333:               System.out.println(tNewCircle.getRadius());
```

Suppose instead you declare `translate` with a return type of `Circle`:

```
0367:    public Circle translate(int pX, int pY)
```

If you compile this with a Java 1.4 compiler, you will get the compilation error:

```
0381: Circle.java:12: translate(int,int) in Circle cannot
0382: override translate(int,int) in Shape; attempting to use
0383: incompatible return type
0384: found   : Circle
0385: required: Shape
0386:    public Circle translate(int pX, int pY)
0387:                  ^
```

This is saying that the method is an incorrect overriding of `Shape`'s `translate` method.

8.2 Java 5 introduces covariant return types

With Java 5 this has changed: Java 5 permits *covariant return types*. This means that Java 5 does not insist that the return type of an overriding method is the same type as the return type of the method in the superclass: it can be a subclass of that return type.

So, in Java 5, `Circle`'s `translate` method can have the header:

```
0367:    public Circle translate(int pX, int pY)
```

and the call of this method can then be simplified to:

```
0406:               Circle tCircle = new Circle(x, y, radius);
0407:               Circle tNewCircle = tCircle.translate(1, 2);
0408:               System.out.println(tNewCircle.getRadius());
```

8.3 Changes to the `clone` method

This also brings one benefit to classes that provide a `clone` method that overrides `Object`'s `clone` method: such classes no longer have to declare their `clone` method with a return type of `Object`.

Suppose a program contains the code:

```
Shape tShape = new Shape(100, 200);
Shape tShapeSame = tShape;
```

then both `Shape` variables point to the same object. Suppose `Shape` provides a `clone` method that overrides `Object`'s `clone` method. Prior to Java 5, you would declare this as:

```
public Object clone()
{
   ...
}
```

and the caller would have to cast the result of the call of `clone`:

```
Shape tShape = new Shape(100, 200);
Shape tShapeCopy = (Shape)tShape.clone();
```

In Java 5, `Shape` can declare its `clone` method using:

```
public Shape clone()
{
   ...
}
```

and the call can then be simplified to:

```
Shape tShape = new Shape(100, 200);
Shape tShapeCopy = tShape.clone();
```

9 Enumeration types

9.1 The problem

Some objects of the real world that we may need to represent within a program are associated with a small set of possible values. There are usually names attached to each of the possible values that the object takes. For example, if we are writing a program representing a piece of four part harmony, we may want to refer to the four parts called *soprano*, *alto*, *tenor* and *bass*.

Some programming languages provide a means for declaring types that represent a small number of values. As this is usually done by enumerating (i.e., listing) the values that belong to the type, such types are called *enumeration types*, *enumerated types* or *enum types*. The first programming language to provide enum types was Pascal. And Modula-2, Ada, C, C++ and C# have followed. Curiously, there is not yet an agreed way of providing enum types: all of these languages provide enum types in different ways. The notable exception from the above list of languages has been Java: however, enum types have been introduced into Java with the release of Java 5.

9.2 Java 5 has enumeration types

In Java 5, an enum type is produced using an *enum declaration*. This lists all the constants that denote the values of the enum type. Here is an example:

```
0432:    private enum Day
0433:    {
0434:       Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday
0435:    }
```

Note: although the designers of the enhanced for statement thought it inappropriate to add a new keyword such as `foreach` to the language, those introducing enum types have added the keyword `enum`.

One of the documents supporting the introduction of enum types is available at [28]. It says: 'The `enum` declaration is a special kind of class declaration. An *enum type* has public, self-typed members for each of the named *enum constants*. All enum classes have high-quality `toString`, `hashCode` and `equals` methods.'

Here is a program that uses the type `Day`. Hidden in this code is a call of the `toString` method:

```
0430: public class SimpleDayProg                              // SimpleDayProg.java
0431: {
0432:    private enum Day
0433:    {
0434:       Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday
0435:    }
0436:    public static void main(String[] pArgs)
0437:    {
0438:       Day tToday = Day.Wednesday;   System.out.println(tToday);
0439:    }
0440: }
```

This program outputs:

```
0441: Wednesday
```

9.3 The methods provided by `java.lang.Enum`

Above it was mentioned that each enum type has `toString`, `hashCode` and `equals` methods. This is because each enum type extends the type `java.lang.Enum` (which is new in Java 5). The API documentation for this class is at [30]. Besides `toString`, `hashCode` and `equals`, the class `java.lang.Enum` also declares methods called `compareTo`, `valueOf`, `values` and `ordinal`.

Here is some code illustrating the use of some of these methods:

```
0452:       Scanner tScanner = new Scanner(System.in);
0453:       String tInputString = tScanner.next();
0454:       Day tInputDay = Day.valueOf(tInputString);
0455:       switch (tInputDay)
0456:       {
0457:          case Sunday:   System.out.println("wash car");   break;
0458:          case Saturday: System.out.println("do nothing"); break;
0459:          default:       System.out.println("go to work"); break;
0460:       }
0461:       for (Day tDay : Day.values())
0462:       {
0463:          System.out.println(tDay + " " + tDay.ordinal());
0464:       }
```

This code produces the following dialogue:

```
0500: Sunday
0501: wash car
0502: Sunday 0
0503: Monday 1
0504: Tuesday 2
0505: Wednesday 3
0506: Thursday 4
0507: Friday 5
0508: Saturday 6
0509: Friday Thursday
0510: Thursday Thursday true
0511: Tuesday -1
0512: Wednesday 0
0513: Thursday 1
0514: Tuesday -1
0515: Wednesday 0
0516: Thursday 1
0517: Sunday is a 6 letter day
0518: Monday is a 6 letter day
0519: Friday is a 6 letter day
0520: Sunday is a 6 letter day
0521: Monday is a 6 letter day
0522: Friday is a 6 letter day
```

9.4 The classes `EnumSet` and `EnumMap`

You can use the new class `java.util.EnumSet` to create a set of enums and the new class `java.util.EnumMap` to create a map that uses enums as keys.

Here is some code illustrating two uses of `EnumSet`:

```
0472:       Day tToday = Day.Wednesday;
0473:       Day[] tDayValues = Day.values();
0474:       for (int tDayNumber = Day.Tuesday.ordinal();
0475:                tDayNumber<=Day.Thursday.ordinal(); tDayNumber++)
0476:       {
0477:          Day tDay = tDayValues[tDayNumber];
0478:          System.out.println(tDay + " " + tDay.compareTo(tToday));
0479:       }
0480:       for (Day tDay : EnumSet.range(Day.Tuesday, Day.Thursday))
0481:       {
0482:          System.out.println(tDay + " " + tDay.compareTo(tToday));
0483:       }
0484:       for (Day tDay : Day.values())
0485:       {
0486:          if (tDay.toString().length()==6)
0487:          {
0488:             System.out.println(tDay + " is a 6 letter day");
0489:          }
0490:       }
0491:       for (Day tDay : Day.values())
0492:       {
0493:          if (EnumSet.of(Day.Sunday, Day.Monday, Day.Friday).contains(tDay))
0494:          {
0495:             System.out.println(tDay + " is a 6 letter day");
0496:          }
0497:       }
```

The above code produces the output:

```
0511: Tuesday -1
0512: Wednesday 0
0513: Thursday 1
0514: Tuesday -1
0515: Wednesday 0
0516: Thursday 1
0517: Sunday is a 6 letter day
0518: Monday is a 6 letter day
0519: Friday is a 6 letter day
0520: Sunday is a 6 letter day
0521: Monday is a 6 letter day
0522: Friday is a 6 letter day
```

9.5 Providing fields and methods in an enum type

As well as listing the values of the enum type, in Java 5 an enum declaration can also supply fields and methods that can be used on values of the enum type. Above the enum declaration was declared inside a class declaration: when enum declarations become more complicated, they are probably best stored in a separate file.

Suppose the file `Day.java` contains:

```
0523: public enum Day                                                  // Day.java
0524: {
0525:    Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday;
0526:    public String toString()
0527:    {
0528:       return super.toString().substring(0, 3);
0529:    }
0530:    private static Day[] iDayValues = Day.values();
0531:    public static Day getFirst()
0532:    {
0533:       return iDayValues[0];
0534:    }
0535:    public boolean isLast()
0536:    {
0537:       return this==iDayValues[iDayValues.length - 1];
0538:    }
0539:    public Day getNext()
0540:    {
0541:       int tNewPos = ordinal() + 1;
0542:       if (tNewPos==iDayValues.length)
0543:          return iDayValues[0];
0544:       else
0545:          return iDayValues[tNewPos];
0546:    }
0547: }
```

The above enum type could then be used in the following way:

```
0548: public class DayProg                                          // DayProg.java
0549: {
0550:    public static void main(String[] pArgs)
0551:    {
0552:       System.out.println("getFirst: " + Day.getFirst());
0553:       Day tToday = Day.Wednesday;
0554:       System.out.println("tToday.isLast: " + tToday.isLast());
0555:       System.out.println("Day.Saturday.isLast: " + Day.Saturday.isLast());
0556:       for (Day tDay : Day.values())
0557:       {
0558:          System.out.print(tDay.getNext() + " ");
0559:       }
0560:       System.out.println();
0561:    }
0562: }
```

The above code outputs:

```
0563: getFirst: Sun
0564: tToday.isLast: false
0565: Day.Saturday.isLast: true
0566: Mon Tue Wed Thu Fri Sat Sun
```

9.6 Associating values with each enum value

One final twist: each value of an enum type can have associated values in which case the enum type needs an appropriate constructor (which must be marked as `private`). Here is an example:

```
0567: public enum ShopDay                                          // ShopDay.java
0568: {
0569:    Sunday   (new Time("10:00"), new Time("16:00")),
0570:    Monday   (new Time("08:00"), new Time("17:30")),
0571:    Tuesday  (new Time("08:00"), new Time("17:30")),
0572:    Wednesday(new Time("08:00"), new Time("17:30")),
0573:    Thursday (new Time("08:00"), new Time("17:30")),
0574:    Friday   (new Time("08:00"), new Time("17:30")),
0575:    Saturday (new Time("09:00"), new Time("12:30"));
0576:    private Time iStartTime;
0577:    private Time iFinishTime;
0578:    private ShopDay(Time pStartTime, Time pFinishTime)
0579:    {
0580:       iStartTime = pStartTime;
0581:       iFinishTime = pFinishTime;
0582:    }
0583:    public Time getStartTime()
0584:    {
0585:       return iStartTime;
```

```
0586:    }
0587:    public Time getFinishTime()
0588:    {
0589:       return iFinishTime;
0590:    }
0591: }

0592: public class Time                                          // Time.java
0593: {
0594:    private String iString;
0595:    public Time(String pString)
0596:    {
0597:       iString = pString;
0598:    }
0599:    public String toString()
0600:    {
0601:       return iString;
0602:    }
0603: }

0604: public class ShopDayProg                                  // ShopDayProg.java
0605: {
0606:    public static void main(String[] pArgs)
0607:    {
0608:       for (ShopDay tShopDay : ShopDay.values())
0609:       {
0610:          System.out.println(tShopDay + " " + tShopDay.getStartTime() +
0611:                                        " " + tShopDay.getFinishTime());
0612:       }
0613:    }
0614: }
```

The above program produces the output:

```
0615: Sunday 10:00 16:00
0616: Monday 08:00 17:30
0617: Tuesday 08:00 17:30
0618: Wednesday 08:00 17:30
0619: Thursday 08:00 17:30
0620: Friday 08:00 17:30
0621: Saturday 09:00 12:30
```

9.7 Is an enum type a class type?

With most programming languages, a unique integer is associated with each enum value. If a variable that is of enum type is assigned a value, the appropriate integer value is stored in the variable.

However, the literature about Java 5 refers to an enum declaration as being a *special kind of class declaration*. It is probably the case that each enum value has an associated object. So, when an enum variable is assigned an enum value, a pointer to the appropriate enum object is assigned to the enum variable. Because of this, the `==` operator produces the same result as the `equals` method.

9.8 Differences from C#

Here are the two main differences from C#:

1. In C#, it is possible to use the relational operators (<, <=, > and >=) with enum values, but this is not possible in Java 5. So, although C# permits:

   ```
   tDay1<=tDay2
   ```

   in Java 5 this has to be written as:

   ```
   tDay1.compareTo(tDay2)<=0
   ```

2. In C#, it is not possible to declare constructors, fields and methods in an enum declaration.

9.9 More about enumeration types

See Item 21 of [2], page 13 of [24] or pages 485 to 496 of [6] for details about how to mimic enumeration types in Java 1.4 (or earlier). See [28], [29], [30], [21], [18] and [23] for other information about the enumeration types of Java 5.

10 Annotating code with metadata

10.1 Adding annotations to programs

Java 5 allows you to annotate elements of a program. Examples are the uses of `@NeedsFurtherTesting` and `@ToDo` in the following code:

```
0633:    @NeedsFurtherTesting
0634:    private static double iGetDouble(String pLabel)
0635:    {
0636:       System.out.print("Type in your " + pLabel + " number: ");
0637:       Scanner tScanner = new Scanner(System.in);
0638:       return tScanner.nextDouble();
0639:    }
0640:    @NeedsFurtherTesting
0641:    @ToDo(programmer="Barry Cornelius",
0642:          importance=ToDo.Importance.High,
0643:          deadline="2004-08-31")
0644:    private static double iFindSum(double pLHS, double pRHS)
0645:    {
0646:       return pLHS + pRHS;
0647:    }
```

Besides annotating method declarations (as shown above), a program can also annotate other declarations such as those of constructors, fields, enum constants, local variables, packages, parameters, classes, interfaces, annotation types and enum types.

These annotations need not directly affect the semantics of a program. Instead, tools can read these annotations and process them in some way.

Note: Javadoc comments and the `@deprecated` tag could now be replaced by annotations.

Note: in some situations, you need to provide stylised Java source code, code that always takes a particular form. By introducing an annotation, you could shorten the amount of code you write and get a tool to generate the real Java source code for you from your template. The Java 5 release provides such a tool: it is called `apt`. See [27] for more details.

10.2 Declaring annotation types

In order to use an annotation like `@NeedsFurtherTesting` or `@ToDo`, you will need to declare an *annotation type*. Here are some declarations for these two annotation types:

```
0649: public @interface NeedsFurtherTesting            // NeedsFurtherTesting.java
0650: {
0651: }

0652: public @interface ToDo                                          // ToDo.java
0653: {
0654:    public enum Importance {VeryHigh, High, Medium, Low};
0655:    public String programmer();
0656:    public Importance importance();
0657:    public String deadline();
0658: }
```

You can see that the declaration of an annotation type looks very much like the declaration of an interface type. See [21] for more information about annotation types.

10.3 The use of `java.lang.Override`

The APIs of Java 5 provide seven annotation types. If you look at the documentation of the Java 5 APIs, you will see that the package `java.lang` has annotation types called `Deprecated`, `Override` and `SuppressWarnings`; and the package `java.lang.annotation` has annotation types called `Documented`, `Inherited`, `Retention` and `Target`.

I just want to look at one of these: `java.lang.Override`. In the API documentation, this annotation type is documented in the following way:

```
java.lang
Annotation Type Override

@Target(value=METHOD)
@Retention(value=SOURCE)
public @interface Override

Indicates that a method declaration is intended to override a
method declaration in a superclass. If a method is annotated
with this annotation type but does not override a superclass
method, compilers are required to generate an error message.
```

Note: the above text indicates that the declaration of the annotation type `Override` is itself annotated with two annotations: the `Target` annotation indicates that the `Override` annotation can only be used to annotate method declarations, and the `Retention` annotation indicates that the `Override` annotation does not have to be passed into the `.class` file that the compiler is generating.

Suppose a class `Shape` includes the declarations:

```
0682:    public int hashcode()
0683:    {
0684:       return 0;
0685:    }
0686:    public String toString(String pString)
0687:    {
0688:       return iX + ":" + iY;
0689:    }
```

Here `hashcode` is not overriding `Object`'s `hashCode` method because `Shape` has misspelt `hashCode` as `hashcode`. Similarly, `toString` is not overriding `Object`'s `toString` method because `Object`'s `toString` method has no parameters. But the above code is legal: it is just that it is providing something different from what you probably intended. When you come to use these methods, you will get a result different from what you expected. For example:

```
0725:          Shape tShape = shapes[shapeNumber];
0726:          System.out.println(tShape);
```

will use the `toString` method from the class `Object`.

The above is so typical of what happens: it is so easy to get the wrong spelling or the wrong parameters. So, if you are using Java 5, my advice would be always to use an `@Override` annotation when overriding:

```
0753:    @Override
0754:    public int hashcode()
0755:    {
0756:       return 0;
0757:    }
0758:    @Override
0759:    public String toString(String pString)
0760:    {
0761:       return iX + ":" + iY;
0762:    }
```

If you compile this with the Java 5 compiler, it will produce two compilation errors saying:

```
method does not override a method from its superclass
```

Note: McLaughlin and Flanagan ([21]) say that `@Override` has to appear on a separate line. However, both of the following work with the Java 5 compiler:

```
@Override public int hashcode()
public @Override int hashcode()
```

In my opinion, it would be better to force you to document overriding. This is what is done in C# (and in Eiffel).

10.4 Annotations for Web Services

One of the fads during the last couple of years has been the creation of *Web Services* ([9, 10]). In its simplest form, this is the ability for the code of a method to use the HTTP protocol to call a method executed by a web server at a WWW site somewhere in the world.

The Java Community Process (JSR 181) has produced a set of annotation types for Web Services. So, when you want to say that you are providing a Web Service, you are able to write code like:

```
0764: import javax.jws.WebMethod;                          // HelloWorldService.java
0765: import javax.jws.WebService;
0766: @WebService
0767: public class HelloWorldService
0768: {
0769:    @WebMethod
0770:    public String helloWorld()
0771:    {
0772:       return "Hello World!";
0773:    }
0774: }
```

The `@WebService` annotation is saying that this class is providing methods that we would like programs running elsewhere to be able to execute. And the `@WebMethod` annotation is saying that this is one of the methods that we are happy for programs to execute.

The services that are offered by a Web Service are documented using an XML notation known as the *Web Services Description Language* or *WSDL* ([22]). The developers of JSR 181 ensured that its design makes it easy to write tools to produce WSDL documents from Java source code and vice versa. See [17] for more details of JSR 181.

10.5 Other annotations

The Java Community Process (JSR 250) has also produced 'annotations for common semantic concepts in the J2SE and J2EE platforms that apply across a variety of individual technologies. With the addition of JSR 175 ... we envision that various JSRs will use annotations to enable a declarative style of programming'. See [34] for more details of JSR 250.

11 Parameterized types and methods

11.1 The problem

The interface `List` and its supporting classes (such as `ArrayList`) can be used to manipulate lists of objects of any classes. We can have a list of strings, a list of people, a list of shapes, and so on. This is possible because the interface and the classes are written in terms of the class `Object`.

As was explained earlier, it is possible to add objects of many different classes to the same list. Most of the time we will not want to do this. If we do it inadvertently, the error will not be detected at compilation time as what we have written is allowed.

For example, suppose we have a `List`:

```
0781:         List tList = new ArrayList();
```

in which we are storing `Shapes`, i.e., we normally do something like:

```
0782:         Shape tShape = new Shape(100, 200);
0783:         tList.add(tShape);
```

or:

```
0784:         Circle tCircle = new Circle(100, 200, 10);
0785:         tList.add(tCircle);
```

where the classes `Shape` and `Circle` are as declared in Section 8 of this document.

We can retrieve elements as follows:

```
0786:         Shape tShapeGet0 = (Shape)tList.get(0);
0787:         System.out.println(tShapeGet0);
0788:         Shape tShapeGet1 = (Shape)tList.get(1);
0789:         System.out.println(tShapeGet1);
```

and this code produces the output:

```
0795: 100:200
0796: 100:200:10
```

Suppose our code inadvertently adds a value of some other type to this list:

```
0790:         tList.add("hello world");
```

Then when we come to retrieve this value:

```
0791:         Shape tShapeGet2 = (Shape)tList.get(2);
0792:         System.out.println(tShapeGet2);
```

our program will collapse with a `ClassCastException`:

```
0797: Exception in thread "main" java.lang.ClassCastException
0798:          at ListShape.main(ListShape.java:17)
```

Unless we detect this bug when testing the program, the program will embarassingly crash with a `ClassCastException` when someone uses the program.

11.2 Java 5 provides parameterized types

Java 5 permits the declaration of a *parameterized type* (also known as a *generic type*). This is a class type or an interface type that is written in terms of some unspecified type.

Some of the Java 5 APIs have parameterized types. In particular, the Collection classes have been rewritten so that they are parameterized types. For example, if you look at the Java 5 API documentation for the `ArrayList` class, you will see that it appears as:

```
Class ArrayList<E>
```

Similarly, the interface type `List` is now documented as:

```
Interface List<E>
```

When we come to use these types, we indicate the type we want to use instead of `E`:

```
0805:         List<Shape> tList = new ArrayList<Shape>();
```

The code of the `add` method of `List` (and `ArrayList`) has been altered. Instead of having the header:

```
public boolean add(Object o)
```

the `add` method is now documented as:

```
public boolean add(E o)
```

So, we can add elements as before:

```
0806:         Shape tShape = new Shape(100, 200);
0807:         tList.add(tShape);
0808:         Circle tCircle = new Circle(100, 200, 10);
0809:         tList.add(tCircle);
```

However, if our code inadvertently adds a value of the wrong type:

```
0814:         tList.add("hello world");
```

then that statement will fail to compile:

```
0819: ListShape.java:16: cannot find symbol
0820: symbol  : method add(java.lang.String)
0821: location: interface java.util.List<Shape>
0822:         tList.add("hello world");
0823:              ^
```

This is a lot better than the program crashing with a `ClassCastException`.

There is one other benefit. The `get` method's header has been changed from:

```
public Object get(int index)
```

to:

```
public E get(int index)
```

Because of this, we can retrieve a value without having to use a cast:

```
0810:         Shape tShapeGet0 = tList.get(0);
0811:         System.out.println(tShapeGet0);
0812:         Shape tShapeGet1 = tList.get(1);
0813:         System.out.println(tShapeGet1);
```

When a compiler compiles code containing the use of a parameterized type, *erasure* takes place. This means that the intermediate code generated for lines 805 to 813 is the same as that generated for lines 781 to 789.

11.3 Using a parameterized type for a parameter

The type `List<Shape>` can be used just like any other type. So we can produce methods that have this as their return type, or a method that has this as the type of one of its parameters:

```
0860:     private static void printList1(List<Shape> pList)
0861:     {
0862:        Iterator<Shape> tIterator = pList.iterator();
0863:        while (tIterator.hasNext())
0864:        {
0865:           Shape tShape = tIterator.next();
0866:           System.out.println("X is " + tShape.getX());
0867:        }
0868:     }
```

However, with Java 5, many of the uses of `Iterator` should be replaced by a foreach statement. So the above can be simplified to:

```
0869:     private static void printList2(List<Shape> pList)
0870:     {
0871:        for (Shape tShape : pList)
0872:        {
0873:           System.out.println("X is " + tShape.getX());
0874:        }
0875:     }
```

Things get interesting when you want to write a method that works for any homogeneous list: a list of shapes, a list of strings, and so on. The new notation:

```
List<?>
```

has to be used: it means *a `List` where each element is of some unknown type*. Here it is in action:

```
0876:     private static void printList3(List<?> pList)
0877:     {
0878:        Iterator<?> tIterator = pList.iterator();
0879:        while (tIterator.hasNext())
0880:        {
0881:           Object tObject = tIterator.next();
0882:           System.out.println(tObject);
0883:        }
0884:     }
```

Once again, this can be simplified to:

```
0885:     private static void printList4(List<?> pList)
0886:     {
0887:        for (Object tObject : pList)
0888:        {
0889:           System.out.println(tObject);
0890:        }
0891:     }
```

Suppose instead you want to provide a method that works only for a `List` where the element type is the type `Shape` or any of its subclasses. You may be tempted to use one of the methods given earlier, a method that has a parameter of type `List<Shape>`. However, a class that implements the interface `List<Circle>` cannot be used as an argument to this method. Instead we can use:

```
0892:    private static void printList5(List<? extends Shape> pList)
0893:    {
0894:       for (Shape tShape : pList)
0895:       {
0896:          System.out.println("X is " + tShape.getX());
0897:       }
0898:    }
```

This can be used with an object of a class that implements `List<Shape>`, an object of a class that implements `List<Circle>`, and so on.

11.4 Java 5 also has parameterized methods

Besides parameterized interface types and parameterized class types, Java 5 also permits parameterized methods. So the last two examples can be rewritten as:

```
0899:    private static <GType> void printList6(List<GType> pList)
0900:    {
0901:       for (GType tGType : pList)
0902:       {
0903:          System.out.println(tGType);
0904:       }
0905:    }
0906:    private static <GShape extends Shape> void printList7(List<GShape> pList)
0907:    {
0908:       for (GShape tGShape : pList)
0909:       {
0910:          System.out.println("X is " + tGShape.getX());
0911:       }
0912:    }
```

11.5 Defining our own parameterized type

Besides using the parameterized types that are in the APIs, we could define our own parameterized type. Suppose we want a collection object that remembers seven things: as soon as you add more than seven items, it forgets about something that has already been stored. Below there is a class called `LimitedMemory` that can be used to represent such an object.

It is assumed that all the objects that are to be stored in a `LimitedMemory` object are all of the same type. For this reason, it is written as a parameterized class type. Rather than use `E`, I have chosen to use the name `GType`.

One other thing: it is also assumed that each object has a value, and so each object is of a class that implements the `Valuable` interface. The code below introduces one type that meets these constraints; it is the type `Name`.

```
0914: public interface Valuable                                    // Valuable.java
0915: {
0916:    public int value();
0917: }

0918: public class Name implements Valuable                          // Name.java
0919: {
0920:    private String iString;
0921:    public Name(String pString)
0922:    {
0923:       iString = pString;
0924:    }
0925:    public String toString()
0926:    {
0927:       return iString;
0928:    }
0929:    public int value()
0930:    {
0931:       return iString.length();
0932:    }
0933: }
```

```
0934: import java.util.ArrayList;                              // LimitedMemory.java
0935: import java.util.Iterator;
0936: import java.util.Random;
0937: public class LimitedMemory <GType extends Valuable>
0938:       implements Iterable<GType>
0939: {
0940:    private static final int iCapacity = 7;
0941:    private ArrayList<GType> iArrayList;
0942:    private Random iRandom;
0943:    public LimitedMemory()
0944:    {
0945:       iArrayList = new ArrayList<GType>(iCapacity);
0946:       iRandom = new Random();
0947:    }
0948:    public void add(GType pGType)
0949:    {
0950:       if (iArrayList.size()<iCapacity)
0951:          iArrayList.add(pGType);
0952:       else
0953:          iArrayList.set(iRandom.nextInt(iCapacity), pGType);
0954:    }
0955:    public String toString()
0956:    {
0957:       return iArrayList.toString();
0958:    }
0959:    public int value()
0960:    {
0961:       int tSum = 0;
0962:       for (GType tGType : iArrayList)
0963:       {
0964:          tSum += tGType.value();
0965:       }
0966:       return tSum;
0967:    }
0968:    public Iterator<GType> iterator()
0969:    {
0970:       return iArrayList.iterator();
0971:    }
0972: }
```

Here are two fragments of code that use a `LimitedMemory` object:

```
0978:       LimitedMemory<Name> tLimitedMemory = new LimitedMemory<Name>();
0979:       tLimitedMemory.add(new Name("jim"));
0980:       tLimitedMemory.add(new Name("bert"));

1031:       iMyPrintln(tLimitedMemory);
1032:       System.out.println(" " + tLimitedMemory.value());
1033:       LimitedMemory<Name> tLimitedMemory2 = new LimitedMemory<Name>();
1034:       tLimitedMemory2.add(new Name("jim"));
1035:       tLimitedMemory2.add(new Name("bert"));
1036:       tLimitedMemory2.add(new Name("jill"));
1037:       System.out.println(iBiggestCost(tLimitedMemory, tLimitedMemory2));
```

The above code uses the following subsidiary methods:

```
1039:    private static void iMyPrintln(LimitedMemory<Name> pLimitedMemory)
1040:    {
1041:       System.out.print("+");
1042:       for (Name tName : pLimitedMemory)
1043:       {
1044:          System.out.print(tName + "+");
1045:       }
1046:       System.out.println();
1047:    }
1048:    private static
1049:       <GType1 extends Valuable, GType2 extends Valuable>
1050:          int iBiggestCost(
1051:             LimitedMemory<GType1> pLimitedMemory1,
1052:             LimitedMemory<GType2> pLimitedMemory2)
1053:    {
1054:       return Math.max(pLimitedMemory1.value(), pLimitedMemory2.value());
1055:    }
```

11.6 More about parameterized types and methods

See [5], [21], [1], [11], [12] and [16] for other information about parameterized types and methods.

12 Conclusions

In some ways, C# has been Sun's wake-up call. Although the Java Community Process had been working on generics long before C# appeared, in producing Java 5 Java has acquired several new language features that are already in C#. This catch-up has levelled the playing field.

Of course, there are disadvantages with making significant changes to a programming language. There is more for people to learn, and existing books and courses will need to be updated. And developers will not be able to avoid the changes for long as future APIs will use the new features.

There is one another disadvantage. Some of these language changes need support from classes that are new in Java 5: this means that source code written to use these new features need the Java 5 JRE (Java Runtime Environment) in order to execute. So, for the time being, it would be prudent to generate JDK 1.4 code when producing Java applets.

Although there are these snags, Java 5 will be a nicer language to work with. Most of the changes being introduced enables us to produce better quality code: either making the code easier to understand or less likely to crash.

13 And finally

The book by McLaughlin and Flanagan ([21]) is a delight: if you want to learn more about Java 5, this is currently the best book to get. It is packed full of useful information.

I'm also grateful for the comments on an earlier draft of this material given by an audience at the University of Durham. This document has improved as a result.

14 References

1. Maria Lucia Barron-Estrada and Ryan Stansifer, 'A Comparison of Generics in Java and C#', http://www.cs.fit.edu/~mbarron/papers/ACM03.pdf

2. Joshua Bloch, 'Effective Java', Addison Wesley, 2001, 0-201-31105-8.

3. Joshua Bloch, 'New Language Features for Ease of Development in the Java 2 Platform, Standard Edition 1.5: A Conversation with Joshua Bloch', http://java.sun.com/features/2003/05/bloch_qa.html

4. Joshua Bloch and Neal Gafter, 'Forthcoming Java Programming Language Features', http://www.cse.ogi.edu/class/cse529/slides/Tiger%20lang%20sldies%201_up.pdf

5. Gilad Bracha, 'Generics in the Java Programming Language', http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf

6. Barry Cornelius, 'Understanding Java', Addison Wesley, 2001, 0-201-71107-9.

7. Barry Cornelius, 'A Taste of C#', http://www.barrycornelius.com/papers/a.taste.of.csharp/

8. Barry Cornelius, 'Comparing .NET with Java', http://www.barrycornelius.com/papers/comparing.dotnet.with.java/

9. Barry Cornelius, 'Web Services using Axis', http://www.barrycornelius.com/papers/web.services.using.axis/

10. Barry Cornelius, 'Web Services using .NET', http://www.barrycornelius.com/papers/web.services.using.dotnet/

11. Bruce Eckel, 'Generics Aren't', http://mindview.net/WebLog/log-0050

12. Bruce Eckel, 'Latent Typing', http://mindview.net/WebLog/log-0051

13. Bruce Eckel, 'Java Issues & Directions', http://mindview.net/WebLog/log-0055

14. Bruce Haddon, ‘The Java Tiger Release’,
http://www.cs.colorado.edu/events/colloquia/docs/tiger.pdf

15. Benedict Heal, ‘Taming the Tiger’,
http://noops.prismtechnologies.com/talks/52/Event_075.ppt

16. Anders Hejlsberg, ‘Generics in C#, Java, and C++’,
http://www.artima.com/intv/genericsP.html

17. JSR 181, ‘Web Services Metadata for the Java Platform’,
http://jcp.org/en/jsr/detail?id=181

18. Jeff Langr, ‘Typesafe Enum: Using enum in J2SE 1.5 (Tiger)’,
http://www.langrsoft.com/articles/enum.html

19. John Lewis and William Loftus, ‘Java Software Solutions: Foundations of Program Design (3rd edition)’,
Addison Wesley, 2003, 0-201-78129-8.

20. John Lewis and William Loftus, ‘Java Software Solutions: Foundations of Program Design (4th edition)’,
Addison Wesley, 2004, 0-321-26979-9.

21. Brett McLaughlin and David Flanagan, ‘Java 1.5 Tiger: A Developer’s Notebook’,
O’Reilly, 2004, 0-596-00738-8.

22. Microsoft, ‘Web Services Description Language (WSDL) Explained’,
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnwebsrv/html/wsdlexplained.asp

23. Matt Quail, ‘Groking `Enum` (aka `Enum<E extends Enum<E>>`)’,
http://madbean.com/blog/2004/3/

24. Robert Stroud, ‘The Evolution of Java’,
http://www.ics.ltsn.ac.uk/events/jicc8/JICCstroud JDK1.5.pdf

25. Sun Microsystems, ‘Java 2 SDK, Standard Edition, Version 1.5.0 Summary of New Features and Enhancements’, http://java.sun.com/j2se/1.5.0/docs/relnotes/features.html

26. Sun Microsystems, ‘[Changes to the] Java Programming Language’,
http://java.sun.com/j2se/1.5.0/docs/guide/language/

27. Sun Microsystems, ‘Getting Started with the Annotation Processing Tool (apt)’,
http://java.sun.com/j2se/1.5.0/docs/guide/apt/GettingStarted.html

28. Sun Microsystems, ‘A Typesafe Enum Facility for the Java Programming Language’,
http://www.jcp.org/aboutJava/communityprocess/jsr/tiger/enum.html

29. Sun Microsystems, ‘Enums’,
http://java.sun.com/j2se/1.5.0/docs/guide/language/enums.html

30. Sun Microsystems, ‘API documentation for the `Enum` class’,
http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Enum.html

31. Sun Microsystems, ‘API documentation for the `Formatter` class’,
http://java.sun.com/j2se/1.5.0/docs/api/java/util/Formatter.html

32. Sun Microsystems, ‘API documentation for the `Scanner` class’,
http://java.sun.com/j2se/1.5.0/docs/api/java/util/Scanner.html

33. JSR 14, ‘Add Generic Types to the Java Programming Language’,
http://jcp.org/en/jsr/detail?id=14

34. JSR 250, ‘Common Annotations for the Java Platform’,
http://jcp.org/en/jsr/detail?id=250