

2nd Edition
Coverage of 1.5

Solutions and Examples for Java Developers



Java[™] Cookbook[™]

O'REILLY[®]

Ian F. Darwin

Data Structuring with Generics, foreach, and Enumerations (JDK 1.5)

8.0 Introduction

JDK 1.5 introduced two new concepts, Generics and AutoBoxing/Unboxing, and two new language features, “foreach” and typesafe enumerations; these features are covered in this chapter.

Generics allow a class to be tailored for a variety of argument/return types at compile time. It provides similar functionality to the Templates mechanism that has been in C++ for some years. When used with the Collections classes (see Chapter 7), Generics significantly increase type safety and remove the requirement for downcasting every object that is retrieved from a Collection (either directly or via an Iterator).

Recipe 5.3 discussed the “Wrapper” classes; these classes provide Objects that represent primitive values. Autoboxing automates the conversion from primitive to Object and vice versa.

The new “foreach” mechanism does not give Java a new keyword; the “for” keyword is still used. However, the syntax is changed slightly to resemble the “for value in list” construct in languages such as the Unix Bourne Shell and the Awk scripting language. This makes it much easier to use Collections, often eliminating the need to obtain an Iterator.

Finally, the Typesafe Enumerations feature provides a mechanism for dealing with a small list of discrete values, such as months, colors, and the like. It combines the enum syntax from Java’s predecessor, C, with the Typesafe Enumeration design pattern, giving JDK 1.5 the best of both worlds.

These represent, for the most part, changes to the compiler and the class libraries, rather than changes to the underlying Java Virtual Machine; the changes were implemented by having the compiler map from new constructions to calls on classes, rather than by perturbing the JVM. Because of this, in order to maintain backward compatibility (so that the 1.5 compiler can compile 1.4-level code), these new mechanisms are not enabled by default. So get very used to typing the magical incantation:

```
javac -source 1.5 file.java
```

For without this, the examples in this chapter will universally fail to compile. You have been warned!

You do not, at least, need any special options to the *java* command that invokes the JVM to interpret your class file because, as mentioned, the changes are in the compiler, not in the JVM itself. This does not mean that most of this chapter's material can be compiled on 1.5 and run on 1.4; because of the need for extra methods in the Java runtime, you can't run most of the generated code under 1.4.

See Also

The compiler techniques used in compiling these new constructs in a backward-compatible way include erasure and bridging, topics discussed in an article by O'Reilly author William Grosso, which can be read online at <http://today.java.net/pub/a/today/2003/12/02/explorations.html>.

8.1 Using Generic Collections

Problem

You want to store your data in one of the *Collection* classes defined in Chapter 7 but have it treated as though it were homogeneous.

Solution

Use the JDK 1.5 Generic Types mechanism, and declare the *Collection* with reference to the given type. The type name appears in angle brackets after the declaration and instantiation. For example, to declare an *ArrayList* for holding *String* object references:

```
List<String> myList = new ArrayList<String>();
```

Discussion

When you instantiate a *Collection* (or any other class using Generic Types), the class appears to be instantiated with the type given in angle brackets becoming the type of arguments passed in, values returned, and so on. Recipe 8.3 provides some details on the implementation. As an example, consider the code in Example 8-1, which creates and uses an *ArrayList* specialized to contain *String* objects.

Example 8-1. ArrayListGenericDemo.java

```
import java.util.*;

public class ArrayListGenericDemo {
    public static void main(String[] args) {
        ArrayList<String> data = new ArrayList<String>();
        data.add("hello");
        data.add("goodbye");
    }
}
```

Example 8-1. *ArrayListGenericDemo.java* (continued)

```
// data.add(new Date()); This won't compile!

Iterator<String> it = data.iterator();
while (it.hasNext()) {
    String s = it.next();
    System.out.println(s);
}
}
```

As you know from the `ArrayList` example in Recipe 7.3, prior to Generics, the references obtained from a `Collection` or `Iterator` would have to be downcasted to their specific type, often after testing with the `instanceof` operator. A key benefit of Generic Types is that they obviate this testing and downcasting by doing more work at compile time.

You can still instantiate classes such as `ArrayList` without using a specific type. In this case, they behave as in 1.4—that is, the objects returned from a `Collection` or `Iterator` are typed as `java.lang.Object` and must be downcasted before use.

As a further example, consider the `Map` interface mentioned in Chapter 7. A `Map` requires `Keys` and `Values` in its `put()` method. A `Map`, therefore, has two parameterized types. To set up a `Map` whose keys are `Person` objects and whose values are `Address` objects (assuming these two classes exist in your application), you could define it as:

```
Map<Person, Address> addressMap = new HashMap<Person, Address>();
```

This `Map` would expect a `Person` as its key and an `Address` as its value in the `put()` method; the `get()` method would return an `Address` object. The `keySet()` method would return `Set<Person>`—a `Set` specialized for `Person` objects—and so on.

8.2 Using “foreach” Loops

Problem

You want a convenient means of accessing all the elements of an array or collection.

Solution

Use the JDK 1.5 “foreach” construction. For example:

```
for (String s : myList)
```

This form of `for` is always read as “foreach” and is referred to that way in the documentation and the compiler messages; the colon (`:`) is always pronounced as “in” so that the above statement is read as “foreach `String s` in `myList`.” The `String` named `s` will be given each value from `myList` (which is presumed to be declared as an array or `Collection` of `String` references).

Discussion

The foreach construction can be used on Java arrays and on collection classes. The compiler turns it into an iteration, typically using an Iterator object where Collection classes are involved. Example 8-2 shows foreach on an array; in a slightly longer example, Example 8-3 shows foreach on a Collection.

Example 8-2. ForeachArray.java

```
public class ForeachArray {
    public static void main(String args[]) {
        String[] data = { "Toronto", "Stockholm" };
        for (String s : data) {
            System.out.println(s);
        }
    }
}
```

Example 8-3. ForeachDemo.java

```
import java.util.Collection;
import java.util.List;
import java.util.ArrayList;

public class ForeachDemo {
    static void iterate(Collection<String> c) {
        for (String s : c)
            System.out.println(s);
    }
    public static void main(String args[]) {
        List<String> l = new ArrayList<String>();
        l.add("Toronto");
        l.add("Stockholm");
        iterate(l);
    }
}
```

8.3 Avoid Casting by Using Generics

Problem

You wish to define your own container classes using the Generic Type mechanism to avoid needless casting.

Solution

Define a class using `<TypeName>` where the type is declared and `TypeName` where it is used.

Discussion

Consider the very simple Stack class in Example 8-4. This class has been parameterized to take a type whose local name is `T`. This type `T` will be the type of the argument of the `push()` method, the return type of the `pop()` method, and so on. Because of this return type—more specific than the `Object` return type of the original Collections—the return value from `pop()` does not need to be downcasted. In 1.5, the Collections Framework classes have been modified similarly.

Example 8-4. MyStack.java

```
/** A lax Stack implementation. */
public class MyStack<T> {
    private int ix = 0;
    public final int MAX = 10;
    private T[] data = (T[])new Object[MAX];

    public void push(T obj) {
        data[ix++] = obj;
    }

    public boolean hasNext() {
        return ix > 0;
    }

    public boolean hasRoom() {
        return ix < MAX;
    }

    public T pop() {
        if (hasNext()) {
            return data[--ix];
        }
        throw new ArrayIndexOutOfBoundsException(-1);
    }
}
```

The association of a particular type is done at the time the class is instantiated. For example, to instantiate a `MyStack` specialized for holding `BankAccount` objects, one would need to code only the following:

```
MyStack<BankAccount> theAccounts = new MyStack<BankAccount>();
```

Note that if you do not provide a specific type, this class defaults to the most general behavior, that is, type `T` is treated as `java.lang.Object`. So this toy collection, like the real ones in `java.util`, will behave as they did in 1.4—accepting input arguments of any type, returning `java.lang.Object` from getter methods, and requiring downcasting—as their default, backward-compatible behavior. Example 8-5 shows a program that creates two instances of `MyStack`, one specialized for `Strings` and one left general. The general one, called `m2`, is loaded up with the same two `String` objects as `m1` but also includes a `Date` object. The printing code is now “broken”, as it will throw a

ClassCastException: a Date is not a String. I handle this case specially for pedantic purposes: it is illustrative of the kinds of errors you can get into when using nonparameterized container classes.

Example 8-5. MyStackDemo.java

```
public class MyStackDemo {
    public static void main(String[] args) {
        MyStack<String> ms1 = new MyStack<String>();
        ms1.push("billg");
        ms1.push("scottm");

        while (ms1.hasNext()) {
            String name = ms1.pop();
            System.out.println(name);
        }

        // Old way of using Collections: not type safe.
        MyStack ms2 = new MyStack();
        ms2.push("billg");
        ms2.push("scottm");
        ms2.push(new java.util.Date());

        // Show that it is broken
        try {
            String bad = (String)ms2.pop();
            System.err.println("Didn't get expected exception!");
        } catch (ClassCastException ex) {
            System.out.println("Did get expected exception.");
        }

        // Removed the brokenness, print rest of it.
        while (ms2.hasNext()) {
            String name = (String)ms2.pop();
            System.out.println(name);
        }
    }
}
```

Because of this potential for error, the 1.5 compiler warns that you have unchecked raw types. Like the Deprecation warnings discussed in Recipe 1.9, by default, these warnings are not printed in detail. You must ask for them, with the rather lengthy option `-Xlint:unchecked`:

```
C:> javac -source 1.5 MyStackDemo.java
Note: MyStackDemo.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
C:> javac -source 1.5 -Xlint:unchecked MyStackDemo.java
MyStackDemo.java:14: warning: unchecked call to push(T) as a member of the raw
type MyStack
        ms2.push("billg");
            ^
```

```

MyStackDemo.java:15: warning: unchecked call to push(T) as a member of the raw
type MyStack
        ms2.push("scottm");
            ^
MyStackDemo.java:16: warning: unchecked call to push(T) as a member of the raw
type MyStack
        ms2.push(new java.util.Date());
            ^
3 warnings
C:>

```

8.4 Let Java Convert with AutoBoxing and AutoUnboxing

Problem

You are tired of typing code like `new Integer(i)` and `intObj.intValue()` to convert back and forth between primitives and Object Wrappers.

Solution

Use the JDK 1.5 compiler; it will AutoBox and AutoUnbox for you.

Discussion

There's a reason they call it automatic boxing: you don't have to do any work. The 1.5 compiler is finally able to figure out how to convert back and forth between primitives and their wrappers. Example 8-6 shows converting from a primitive `int` value to an `Integer` needed in a method call.

Example 8-6. AutoboxDemo.java

```

public class AutoboxDemo {
    public static void main(String[] args) {
        int i = 42;
        foo(i);
    }

    public static void foo(Integer i) {
        System.out.println("Object = " + i);
    }
}

```

This code compiles and runs on JDK 1.5 (but only with the `-source 1.5` option). Notice what happens when we omit that option:

```

C:\ian\javasrc\structure1.5>javac AutoboxDemo.java
AutoboxDemo.java:4: foo(java.lang.Integer) in AutoboxDemo cannot be applied to (int)
        foo(i);
            ^

```

```
1 error
```

```
C:\ian\javasrc\structure1.5>javac -source 1.5 AutoboxDemo.java
```

```
C:\ian\javasrc\structure1.5>java AutoboxDemo  
Object = 42
```

```
C:\ian\javasrc\structure1.5>
```

The resulting class file does not run in a JDK 1.4 implementation because it depends on a new method signature in the Integer class, notably `valueOf(int i)`.

8.5 Using Typesafe Enumerations

Problem

You need to manage a small list of discrete values within a program.

Solution

Use the JDK 1.5 enum mechanism.

Discussion

To enumerate means to list all the values. You often know that a small list of possible values is all that's wanted in a variable, such as the months of the year, the suits or ranks in a deck of cards, the primary and secondary colors, and so on. The C programming language provided an enum keyword:

```
enum { BLACK, RED, ORANGE } color;
```

Java has been criticized since the earliest releases for its lack of enumerations, which many developers have wished for. Many have had to develop custom classes to implement the “enumeration pattern.”

But C enumerations are not “typesafe”; they simply define constants that can be used in any integer context. For example, this code compiles without warning, even on gcc 3 with `-Wall` (all warnings), while a C++* compiler catches the error:

```
enum { BLACK, RED, ORANGE } color;  
enum { READ, UNREAD } state;  
  
/*ARGSUSED*/  
int main(int argc, char *argv[]) {  
    color = RED;  
    color = READ;  
    return 0;  
}
```

* For Java folks not that familiar with C/C++, C is the older, non-OO language; C++ is an OO derivative of C; and Java is in part a portable, more strongly typesafe derivative of C++.

To replicate this mistake in Java, one needs only to define a series of `final int` values; it will still not be typesafe. By typesafe I mean that you can not accidentally use values other than those defined for the given enumeration. The definitive statement on the “typesafe enumeration pattern” is probably the version defined in Item 21 of Joshua Bloch’s book *Effective Java* (Addison Wesley). Bloch was one of the authors of the Typesafe Enumeration specification for JDK 1.5, so you can be sure the book does a good job of implementing his pattern. These enums are implemented as Classes, subclassed (transparently, by the compiler) from the new class `java.lang.Enum`. Unlike C, and unlike the “series of final int” implementation, JDK 1.5 typesafe enumerations:

- Are printable (they print as the name, not as an underlying `int` implementation).
- Are almost as fast as `int` constants, but the code is more readable.
- Can be easily iterated over.
- Utilize a separate namespace for each enum type, so you don’t have to prefix each with some sort of constant name, like `ACCOUNT_SAVINGS`, `ACCOUNT_CHECKING`, etc.

Enum constants are not compiled into clients, giving you the freedom to reorder the constants within your enum without recompiling the client classes. Normally this works correctly and, even if you blow it and remove a constant that a client depends on, you’ll get an informative message instead of a cryptic crash. Additionally, an enum type is a class so it can, for example, implement arbitrary interfaces, and you can add arbitrary fields and methods to an enum class.

Compared to Bloch’s Typesafe Enum pattern in the book:

- JDK 1.5 enums are simpler to use and more readable (those in the book require a lot of methods, making them cumbersome to write).
- Enums can be used in switch statements.

So there are many benefits and few pitfalls.

The new enum keyword is at the same level as the keyword `class` in declarations. That is, an enum may be declared in its own file with `public` or default access. It may also be declared inside classes, much like nested or inner classes (see Recipe 9.6). *Media.java*, shown in Example 8-7, is a code sample showing the definition of a typesafe enum.

Example 8-7. Media.java

```
public enum Media {  
    book, music_cd, music_vinyl, movie_vhs, movie_dvd;  
}
```

Notice that an enum is a class; see what *javap* thinks of the Media class:

```
C:> javap Media  
Compiled from "Media.java"
```

```

public class Media extends java.lang.Enum{
    public static final Media book;
    public static final Media music_cd;
    public static final Media music_vinyl;
    public static final Media movie_vhs;
    public static final Media movie_dvd;
    public static final Media[] values();
    public static Media valueOf(java.lang.String);
    public Media(java.lang.String, int);
    public int compareTo(java.lang.Enum);
    public int compareTo(java.lang.Object);
    static {};
}
C:>

```

Product.java, shown in Example 8-8, is a code sample that uses the *Media* enum.

Example 8-8. Product.java

```

import com.darwinsys.util.Debug;

public class Product {
    String title;
    String artist;
    Media media;

    public Product(String artist, String title, Media media) {
        this.title = title;
        this.artist = artist;
        switch (media) {
            case book:
                Debug.println("media", title + " is a book");
                break;
            case music_cd:
                Debug.println("media", title + " is a CD");
                break;
            case music_vinyl:
                Debug.println("media", title + " is a relic");
                break;
            case movie_vhs:
                Debug.println("media", title + " is on tape");
                break;
            case movie_dvd:
                Debug.println("media", title + " is on DVD");
                break;
            default:
                Debug.println("media", "Warning: " + title +
                    " : Unknown media " + media);
                break;
        }
        this.media = media;
    }
}

```

In Example 8-9, `MediaFancy` shows how operations (methods) can be added to enumerations; the `toString()` method is overridden for the “book” value of this enum.

Example 8-9. MediaFancy.java

```
public enum MediaFancy {
    book {
        public String toString() { return "Book"; }
    },
    music_cd, music_vinyl, movie_vhs, movie_dvd;

    public static void main(String[] args) {
        MediaFancy[] data = { book, movie_dvd, music_vinyl };
        for (MediaFancy mf : data) {
            System.out.println(mf);
        }
    }
}
```

Running the `MediaFancy` program produces this output:

```
Book
movie_dvd
music_vinyl
```

That is, the `Book` values print in a “user-friendly” way compared to the default way the other values print. You’d want to extend this to all the values in an Enumeration.

Finally, `EnumList`, in Example 8-10, shows how to list all the possible values that a given enum can take on; simply iterate over the array returned by the class’s `values()` method.

Example 8-10. EnumList.java

```
/** Simple demo to print all the types of an enum. */
public class EnumList {
    public static void main(String[] args) {
        enum State { ON, OFF, UNKNOWN };
        for (State i : State.values()) {
            System.out.println(i);
        }
    }
}
```

The output of the `EnumList` program is, of course:

```
ON
OFF
UNKNOWN
```

8.6 Program: MediaInvoicer

This sketch of an invoicing program demonstrates several of these concepts. `MediaFactory` is a class that takes a `String` (or `int`) and returns the corresponding `Media` enum constant:

```
/*
 * MediaFactory - give out Media enumeration constants
 * @verion $Id: ch08,v 1.5 2004/05/04 20:11:57 ian Exp $
 */
public class MediaFactory {

    public static void main(String[] args) {

        System.out.println(MediaFactory.getMedia("Book"));
    }
    public static Media getMedia(String s) {
        return Enum.valueOf(Media.class, s.toLowerCase());
    }
    public static Media getMedia(int n){
        return Media.values()[n];
    }
}
```

This program uses the `valueOf()` method inherited from `java.lang.Enum` by all user-defined enumerations. The `MediaFactory` is used in the main class's `getInvoice()` method to get the correct enum constant for a given `String` read from the *invoices* file, a sample of which looks like this:

```
# Lines beginning with # are comments, ignored by the program
# I invoice# cust#
# M media item# quantity
M Book 2074 1
M MUSIC_VINYL 107 1
M MUSIC_CD 5102 5
M book 2100 1
```

As you can see, the data entry people have been careless about case, but the `MediaFactory` class turns the strings into lowercase before looking them up. The main program pulls these lines apart with a `StringTokenizer` and parses the ints with `Integer.parseInt()`.

The main class uses two helper classes, `Invoice` and `Item`. An invoice can have one or more line items in it, so the `Invoice` object contains an array of `Item` objects.

Example 8-11 is the code for the main `MediaInvoicer` class and the helper classes.

Example 8-11. MediaInvoicer.java

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.InputStream;
```

Example 8-11. MediaInvoicer.java (continued)

```
import java.io.PrintStream;
import java.util.ArrayList;
import java.util.List;
import java.util.StringTokenizer;

/**
 * MediaInvoicer - Simple applicatin of Media, MediaFactory &c.
 *
 * @author ian
 * @version $Id: ch08,v 1.5 2004/05/04 20:11:57 ian Exp $
 */
public class MediaInvoicer {

    public static void main(String[] args) throws IOException {
        MediaInvoicer mi = new MediaInvoicer(System.in);
        Invoice i = mi.getInvoice();
        i.print(System.out);
    }
    BufferedReader myFile;
    public MediaInvoicer(InputStream is) {
        myFile = new BufferedReader(new InputStreamReader(is));
    }

    public Invoice getInvoice() throws IOException {
        String line;
        List < Item > items = new ArrayList < Item > ();
        while ((line = myFile.readLine()) != null) {
            if (line.startsWith("#")) {
                continue;
            }
            StringTokenizer st = new StringTokenizer(line);
            st.nextToken();
            Media m = MediaFactory.getMedia(st.nextToken());
            int stock = Integer.parseInt(st.nextToken());
            int qty = Integer.parseInt(st.nextToken());
            Item tmp = new Item(m, stock, qty);
            items.add(tmp);
        }
        return new Invoice(1, 3,
            (Item[]) items.toArray(new Item[items.size()]));
    }

    /** Inner class for line order item */
    class Item {
        Media product;
        int stockNumber;
        int quantity;
        /**
         * @param product
         * @param stockNumber
         * @param quantity
         */
    }
}
```

Example 8-11. *MediaInvoicer.java* (continued)

```
    public Item(Media product, int stockNumber, int quantity) {
        super();
        this.product = product;
        this.stockNumber = stockNumber;
        this.quantity = quantity;
    }
    public String toString() {
        return "Item[" + product + " " + stockNumber + "];"
    }
}
/** Inner class for one invoice */
class Invoice {
    int orderNumber;
    int custNumber;
    Item[] items;

    public Invoice(int orderNumber, int custNumber, Item[] items) {
        super();
        this.orderNumber = orderNumber;
        this.custNumber = custNumber;
        this.items = items;
    }
    public void print(PrintStream ps) {
        ps.println("*** Invoice ***");
        ps.println("Customer: " + custNumber + " ");
        ps.println("Our order number: " + orderNumber);
        for (int i = 0; i < items.length; i++) {
            Item it = items[i];
            ps.println(it);
        }
    }
}
}
```

Running the program with the sample file shown in Example 8-11 produces this output:

```
*** Invoice ***
Customer number: 1
Our order number: 3
Item[book 2074]
Item[music_vinyl 107]
Item[music_cd 5102]
Item[book 2100]
```

This demonstrates a bit of the ease of use of Java enumerations—they print as themselves—and the use of a List customized to hold Item objects—the list named items in the `getInvoice()` method.