Speed Up the Java Development Process

Java Generics

and Collections

O'REILLY®

Maurice Naftalin
& Philip Wadler

Java Generics and Collections
by Maurice Naftalin and Philip Wadler

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*safari.oreilly.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Editor: Mike Loukides
Production Services: Windfall Software
Indexers: Maurice Naftalin and Philip Wadler
Cover Designer: Karen Montgomery

Printing History:

October 2006: First Edition.

ISBN-10: 0-596-52775-6
ISBN-13: 978-0-596-52775-4
[M]

CHAPTER 5

Evolution, Not Revolution

One motto underpinning the design of generics for Java is *evolution, not revolution*. It must be possible to migrate a large, existing body of code to use generics gradually (evolution) without requiring a radical, all-at-once change (revolution). The generics design ensures that old code compiles against the new Java libraries, avoiding the unfortunate situation in which half of your code needs old libraries and half of your code needs new libraries.

The requirements for evolution are much stronger than the usual *backward compatibility*. With simple backward compatibility, one would supply both legacy and generic versions for each application; this is exactly what happens in C#, for example. If you are building on top of code supplied by multiple suppliers, some of whom use legacy collections and some of whom use generic collections, this might rapidly lead to a versioning nightmare.

What we require is that the *same* client code works with both the legacy and generic versions of a library. This means that the supplier and clients of a library can make completely independent choices about when to move from legacy to generic code. This is a much stronger requirement than backward compatibility; it is called *migration compatibility* or *platform compatibility*.

Java implements generics via erasure, which ensures that legacy and generic versions usually generate identical class files, save for some auxiliary information about types. It is possible to replace a legacy class file by a generic class file without changing, or even recompiling, any client code; this is called *binary compatibility*.

We summarize this with the motto *binary compatibility ensures migration compatibility*—or, more concisely, *erasure eases evolution*.

This section shows how to add generics to existing code; it considers a small example, a library for stacks that extends the Collections Framework, together with an associated client. We begin with the legacy stack library and client (written for Java before generics), and then present the corresponding generic library and client (written for Java with generics). Our example code is small, so it is easy to update to generics all in one go, but in practice the library and client will be much larger, and we may want to evolve them separately. This is aided by *raw types*, which are the legacy counterpart of parameterized types.

The parts of the program may evolve in either order. You may have a generic library with a legacy client; this is the common case for anyone that uses the Collections Framework in Java 5 with legacy code. Or you may have a legacy library with a generic client; this is the case where you want to provide generic signatures for the library without the need to rewrite the entire library. We consider three ways to do this: minimal changes to the source, stub files, and wrappers. The first is useful when you have access to the source and the second when you do not; we recommend against the third.

In practice, the library and client may involve many interfaces and classes, and there may not even be a clear distinction between library and client. But the same principles discussed here still apply, and may be used to evolve any part of a program independently of any other part.

5.1 Legacy Library with Legacy Client

We begin with a simple library of stacks and an associated client, as presented in Example 5.1. This is *legacy* code, written for Java 1.4 and its version of the Collections Framework. Like the Collections Framework, we structure the library as an interface `Stack` (analogous to `List`), an implementation class `ArrayStack` (analogous to `ArrayList`), and a utility class `Stacks` (analogous to `Collections`). The interface `Stack` provides just three methods: `empty`, `push`, and `pop`. The implementation class `ArrayStack` provides a single constructor with no arguments, and implements the methods `empty`, `push`, and `pop` using methods `size`, `add`, and `remove` on lists. The body of `pop` could be shorter—instead of assigning the value to the variable, it could be returned directly—but it will be interesting to see how the type of the variable changes as the code evolves. The utility class provides just one method, `reverse`, which repeatedly pops from one stack and pushes onto another.

The client allocates a stack, pushes a few integers onto it, pops an integer off, and then reverses the remainder into a fresh stack. Since this is Java 1.4, integers must be explicitly boxed when passed to `push`, and explicitly unboxed when returned by `pop`.

5.2 Generic Library with Generic Client

Next, we update the library and client to use generics, as presented in Example 5.2. This is *generic* code, written for Java 5 and its version of the Collections Framework. The interface now takes a type parameter, becoming `Stack<E>` (analogous to `List<E>`), and so does the implementing class, becoming `ArrayStack<E>` (analogous to `ArrayList<E>`), but no type parameter is added to the utility class `Stacks` (analogous to `Collections`). The type `Object` in the signatures and bodies of `push` and `pop` is replaced by the type parameter `E`. Note that the constructor in `ArrayStack` does not require a type parameter. In the utility class, the `reverse` method becomes a generic method with argument and result of type `Stack<T>`. Appropriate type parameters are added to the client, and boxing and unboxing are now implicit.

In short, the conversion process is straightforward: just add a few type parameters and replace occurrences of `Object` by the appropriate type variable. All differences between

Example 5.1. Legacy library with legacy client

```
l/Stack.java:
  interface Stack {
    public boolean empty();
    public void push(Object elt);
    public Object pop();
  }

l/ArrayStack.java:
  import java.util.*;
  class ArrayStack implements Stack {
    private List list;
    public ArrayStack() { list = new ArrayList(); }
    public boolean empty() { return list.size() == 0; }
    public void push(Object elt) { list.add(elt); }
    public Object pop() {
      Object elt = list.remove(list.size()-1);
      return elt;
    }
    public String toString() { return "stack"+list.toString(); }
  }

l/Stacks.java:
  class Stacks {
    public static Stack reverse(Stack in) {
      Stack out = new ArrayStack();
      while (!in.empty()) {
        Object elt = in.pop();
        out.push(elt);
      }
      return out;
    }
  }

l/Client.java:
  class Client {
    public static void main(String[]  args) {
      Stack stack = new ArrayStack();
      for (int i = 0; i<4; i++) stack.push(new Integer(i));
      assert stack.toString().equals("stack[0, 1, 2, 3]");
      int top = ((Integer)stack.pop()).intValue();
      assert top == 3 && stack.toString().equals("stack[0, 1, 2]");
      Stack reverse = Stacks.reverse(stack);
      assert stack.empty();
      assert reverse.toString().equals("stack[2, 1, 0]");
    }
  }
```

the legacy and generic versions can be spotted by comparing the highlighted portions of the two examples. The implementation of generics is designed so that the two versions generate essentially equivalent class files. Some auxiliary information about the types may differ, but the actual bytecodes to be executed will be identical. Hence, executing the legacy and generic versions yields the same results. The fact that legacy and generic sources yield identical class files eases the process of evolution, as we discuss next.

5.3 Generic Library with Legacy Client

Now let's consider the case where the library is updated to generics while the client remains in its legacy version. This may occur because there is not enough time to convert everything all at once, or because the library and client are controlled by different organizations. This corresponds to the most important case of backward compatibility, where the generic Collections Framework of Java 5 must still work with legacy clients written against the Collections Framework in Java 1.4.

In order to support evolution, whenever a parameterized type is defined, Java also recognizes the corresponding unparameterized version of the type, called a *raw type*. For instance, the parameterized type `Stack<E>` corresponds to the raw type `Stack`, and the parameterized type `ArrayStack<E>` corresponds to the raw type `ArrayStack`.

Every parameterized type is a subtype of the corresponding raw type, so a value of the parameterized type can be passed where a raw type is expected. Usually, it is an error to pass a value of a supertype where a value of its subtype is expected, but Java does permit a value of a raw type to be passed where a parameterized type is expected—however, it flags this circumstance by generating an *unchecked conversion* warning. For instance, you can assign a value of type `Stack<E>` to a variable of type `Stack`, since the former is a subtype of the latter. You can also assign a value of type `Stack` to a variable of type `Stack<E>`, but this will generate an unchecked conversion warning.

To be specific, consider compiling the generic source for `Stack<E>`, `ArrayStack<E>`, and `Stacks` from Example 5.2 (say, in directory `g`) with the legacy source for `Client` from Example 5.1 (say, in directory `l`). Sun's Java 5 compiler yields the following message:

```
% javac g/Stack.java g/ArrayStack.java g/Stacks.java l/Client.java
Note: Client.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

The unchecked warning indicates that the compiler cannot offer the same safety guarantees that are possible when generics are used uniformly throughout. However, when the generic code is generated by updating legacy code, we know that equivalent class files are produced from both, and hence (despite the unchecked warning) running a legacy client with the generic library will yield the same result as running the legacy client with the legacy library. Here we assume that the only change in updating the library was to introduce generics, and that no change to the behavior was introduced, either on purpose or by mistake.

Example 5.2. Generic library with generic client

```
g/Stack.java:
  interface Stack<E> {
    public boolean empty();
    public void push(E elt);
    public E pop();
  }

g/ArrayStack.java:
  import java.util.*;
  class ArrayStack<E> implements Stack<E> {
    private List<E> list;
    public ArrayStack() { list = new ArrayList<E>(); }
    public boolean empty() { return list.size() == 0; }
    public void push(E elt) { list.add(elt); }
    public E pop() {
      E elt = list.remove(list.size()-1);
      return elt;
    }
    public String toString() { return "stack"+list.toString(); }
  }

g/Stacks.java:
  class Stacks {
    public static <T> Stack<T> reverse(Stack<T> in) {
      Stack<T> out = new ArrayStack<T>();
      while (!in.empty()) {
        T elt = in.pop();
        out.push(elt);
      }
      return out;
    }
  }

g/Client.java:
  class Client {
    public static void main(String[] args) {
      Stack<Integer> stack = new ArrayStack<Integer>();
      for (int i = 0; i<4; i++) stack.push(i);
      assert stack.toString().equals("stack[0, 1, 2, 3]");
      int top = stack.pop();
      assert top == 3 && stack.toString().equals("stack[0, 1, 2]");
      Stack<Integer> reverse = Stacks.reverse(stack);
      assert stack.empty();
      assert reverse.toString().equals("stack[2, 1, 0]");
    }
  }
```

If we follow the suggestion above and rerun the compiler with the appropriate switch enabled, we get more details:

```
% javac -Xlint:unchecked g/Stack.java g/ArrayStack.java \
%    g/Stacks.java l/Client.java
l/Client.java:4: warning: [unchecked] unchecked call
to push(E) as a member of the raw type Stack
      for (int i = 0; i<4; i++) stack.push(new Integer(i));
                                          ^
l/Client.java:8: warning: [unchecked] unchecked conversion
found   : Stack
required: Stack<E>
      Stack reverse = Stacks.reverse(stack);
                                    ^
l/Client.java:8: warning: [unchecked] unchecked method invocation:
<E>reverse(Stack<E>) in Stacks is applied to (Stack)
      Stack reverse = Stacks.reverse(stack);
                                    ^
3 warnings
```

Not every use of a raw type gives rise to a warning. Because every parameterized type is a subtype of the corresponding raw type, but not conversely, passing a parameterized type where a raw type is expected is safe (hence, no warning for getting the result from `reverse`), but passing a raw type where a parameterized type is expected issues a warning (hence, the warning when passing an argument to `reverse`); this is an instance of the Substitution Principle. When we invoke a method on a receiver of a raw type, the method is treated as if the type parameter is a wildcard, so getting a value from a raw type is safe (hence, no warning for the invocation of `pop`), but putting a value into a raw type issues a warning (hence, the warning for the invocation of `push`); this is an instance of the Get and Put Principle.

Even if you have not written any generic code, you may still have an evolution problem because others have generified their code. This will affect everyone with legacy code that uses the Collections Framework, which has been generified by Sun. So the most important case of using generic libraries with legacy clients is that of using the Java 5 Collections Framework with legacy code written for the Java 1.4 Collections Framework.

In particular, applying the Java 5 compiler to the legacy code in Example 5.1 also issues unchecked warnings, because of the uses of the generified class `ArrayList` from the legacy class `ArrayStack`. Here is what happens when we compile legacy versions of all the files with the Java 5 compiler and libraries:

```
% javac -Xlint:unchecked l/Stack.java l/ArrayStack.java \
%    l/Stacks.java l/Client.java
l/ArrayStack.java:6: warning: [unchecked] unchecked call to add(E)
as a member of the raw type java.util.List
    public void push(Object elt)  list.add(elt);
                                          ^
1 warning
```

Here the warning for the use of the generic method add in the legacy method push is issued for reasons similar to those for issuing the previous warning for use of the generic method push from the legacy client.

It is poor practice to configure the compiler to repeatedly issue warnings that you intend to ignore. It is distracting and, worse, it may lead you to ignore warnings that require attention—just as in the fable of the little boy who cried wolf. In the case of pure legacy code, such warnings can be turned off by using the -source 1.4 switch:

```
% javac -source 1.4 l/Stack.java l/ArrayStack.java \
%    l/Stacks.java l/Client.java
```

This compiles the legacy code and issues no warnings or errors. This method of turning off warnings is only applicable to true legacy code, with none of the features introduced in Java 5, generic or otherwise. One can also turn off unchecked warnings by using annotations, as described in the next section, and this works even with features introduced in Java 5.

5.4 Legacy Library with Generic Client

It usually makes sense to update the library before the client, but there may be cases when you wish to do it the other way around. For instance, you may be responsible for maintaining the client but not the library; or the library may be large, so you may want to update it gradually rather than all at once; or you may have class files for the library, but no source.

In such cases, it makes sense to update the library to use parameterized types in its method signatures, but not to change the method bodies. There are three ways to do this: by making minimal changes to the source, by creating stub files, or by use of wrappers. We recommend use of minimal changes when you have access to source and use of stubs when you have access only to class files, and we recommend against use of wrappers.

5.4.1 Evolving a Library using Minimal Changes

The minimal changes technique is shown in Example 5.3. Here the source of the library has been edited, but only to change method signatures, not method bodies. The exact changes required are highlighed in boldface. This is the recommended technique for evolving a library to be generic when you have access to the source.

To be precise, the changes required are:

- Adding type parameters to interface or class declarations as appropriate (for interface Stack<E> and class ArrayStack<E>)
- Adding type parameters to any newly parameterized interface or class in an extends or implements clause (for Stack<E> in the implements clause of ArrayStack<E>),
- Adding type parameters to each method signature as appropriate (for push and pop in Stack<E> and ArrayStack<E>, and for reverse in Stacks)

- Adding an unchecked cast to each return where the return type contains a type parameter (for pop in ArrayStack<E>, where the return type is E)—without this cast, you will get an error rather than an unchecked warning
- Optionally adding annotations to suppress unchecked warnings (for ArrayStack<E> and Stacks)

It is worth noting a few changes that we do *not* need to make. In method bodies, we can leave occurrences of Object as they stand (see the first line of pop in ArrayStack), and we do not need to add type parameters to any occurrences of raw types (see the first line of reverse in Stacks). Also, we need to add a cast to a return clause only when the return type is a type parameter (as in pop) but not when the return type is a parameterized type (as in reverse).

With these changes, the library will compile successfully, although it will issue a number of unchecked warnings. Following best practice, we have commented the code to indicate which lines trigger such warnings:

```
% javac -Xlint:unchecked m/Stack.java m/ArrayStack.java m/Stacks.java
m/ArrayStack.java:7: warning: [unchecked] unchecked call to add(E)
as a member of the raw type java.util.List
    public void push(E elt)  list.add(elt);   // unchecked call
                                     ^
m/ArrayStack.java:10: warning: [unchecked] unchecked cast
found   : java.lang.Object
required: E
      return (E)elt;  // unchecked cast
                ^
m/Stacks.java:7: warning: [unchecked] unchecked call to push(T)
as a member of the raw type Stack
        out.push(elt);  // unchecked call
                ^
m/Stacks.java:9: warning: [unchecked] unchecked conversion
found   : Stack
required: Stack<T>
      return out;  // unchecked conversion
             ^
4 warnings
```

To indicate that we expect unchecked warnings when compiling the library classes, the source has been annotated to suppress such warnings.

```
@SuppressWarnings("unchecked");
```

(The suppress warnings annotation does not work in early versions of Sun's compiler for Java 5.) This prevents the compiler from crying wolf—we've told it not to issue unchecked warnings that we expect, so it will be easy to spot any that we *don't* expect. In particular, *Example 5.3. Evolving a library using minimal changes*

```
m/Stack.java:
  interface Stack<E> {
    public boolean empty();
    public void push(E elt);
    public E pop();
  }

m/ArrayStack.java:
  @SuppressWarnings("unchecked")
  class ArrayStack<E> implements Stack<E> {
    private List list;
    public ArrayStack() { list = new ArrayList(); }
    public boolean empty() { return list.size() == 0; }
    public void push(E elt) { list.add(elt); }  // unchecked call
    public E pop() {
      Object elt = list.remove(list.size()-1);
      return (E)elt;  // unchecked cast
    }
    public String toString() { return "stack"+list.toString(); }
  }

m/Stacks.java:
  @SuppressWarnings("unchecked")
  class Stacks {
    public static <T> Stack<T> reverse(Stack<T> in) {
      Stack out = new ArrayStack();
      while (!in.empty()) {
        Object elt = in.pop();
        out.push(elt);  // unchecked call
      }
      return out;  // unchecked conversion
    }
  }
```

once we’ve updated the library, we should not see any unchecked warnings from the client. Note as well that we’ve suppressed warnings on the library classes, but *not* on the client.

The only way to eliminate (as opposed to suppress) the unchecked warnings generated by compiling the library is to update the entire library source to use generics. This is entirely reasonable, as unless the entire source is updated there is no way the compiler can check that the declared generic types are correct. Indeed, unchecked warnings are warnings—rather than errors—largely because they support the use of this technique. Use this technique only if you are sure that the generic signatures are in fact correct. The best practice is to use this technique only as an intermediate step in evolving code to use generics throughout.

Example 5.4. Evolving a library using stubs

```
s/Stack.java:
  interface Stack<E> {
    public boolean empty();
    public void push(E elt);
    public E pop();
  }

s/StubException.java:
  class StubException extends UnsupportedOperationException {}

s/ArrayStack.java:
  class ArrayStack<E> implements Stack<E> {
    public boolean empty() { throw new StubException(); }
    public void push(E elt) { throw new StubException(); }
    public E pop() { throw new StubException(); }
    public String toString() { throw new StubException(); }
  }

s/Stacks.java:
  class Stacks {
    public static <T> Stack<T> reverse(Stack<T> in) {
      throw new StubException();
    }
  }
```

5.4.2 Evolving a Library using Stubs

The stubs technique is shown in Example 5.4. Here we write stubs with generic signatures but no bodies. We compile the generic client against the generic signatures, but run the code against the legacy class files. This technique is appropriate when the source is not released, or when others are responsible for maintaining the source.

To be precise, we introduce the same modifications to interface and class declarations and method signatures as with the minimal changes technique, except we completely delete all executable code, replacing each method body with code that throws a `StubException` (a new exception that extends `UnsupportedOperationException`).

When we compile the generic client, we do so against the class files generated from the stub code, which contain appropriate generic signatures (say, in directory `s`). When we run the client, we do so against the original legacy class files (say, in directory `l`).

```
% javac -classpath s g/Client.java
% java -ea -classpath l g/Client
```

Again, this works because the class files generated for legacy and generic files are essentially identical, save for auxiliary information about the types. In particular, the generic signatures that the client is compiled against match the legacy signatures (apart from auxiliary information about type parameters), so the code runs successfully and gives the same answer as previously.

5.4.3 Evolving a Library using Wrappers

The wrappers technique is shown in Example 5.5. Here we leave the legacy source and class files unchanged, and provide a generic wrapper class that accesses the legacy class via delegation. We present this technique mainly in order to warn you *against* its use—it is usually better to use minimal changes or stubs.

This techique creates a parallel hierarchy of generic interfaces and wrapper classes. To be precise, we create a new interface `GenericStack` corresponding to the legacy interface `Stack`, we create a new class `GenericWrapperClass` to access the legacy implementation `ArrayStack`, and we create a new class `GenericStacks` corresponding to the legacy convenience class `Stacks`.

The generic interface `GenericStack` is derived from the legacy interface `Stack` by the same method used in the previous sections to update the signatures to use generics. In addition, a new method `unwrap` is added, that extracts the legacy implementation from a wrapper.

The wrapper class `GenericStackWrapper<E>` implements `GenericStack<E>` by delegation to a `Stack`. The constructor takes an instance that implements the legacy interface `Stack`, which is stored in a private field, and the `unwrap` method returns this instance. Because delegation is used, any updates made to the underlying legacy stack will be seen through the generic stack view offered by the wrapper.

The wrapper implements each method in the interface (`empty`, `push`, `pop`) by a call to the corresponding legacy method; and it implements each method in `Object` that is overridden in the legacy class (`toString`) similarly. As with minimal changes, we add an unchecked cast to the return statement when the return type contains a type parameter (as in `pop`); without this cast you will get an error rather than an unchecked warning.

A single wrapper will suffice for multiple implementations of the same interface. For instance, if we had both `ArrayStack` and `LinkedStack` implementations of `Stack`, we could use `GenericStackWrapper<E>` for both.

The new convenience class `GenericStacks` is implemented by delegation to the legacy class `Stacks`. The generic `reverse` method unwraps its argument, calls the legacy `reverse` method, and wraps its result.

Required changes to the client in Example 5.5 are shown in boldface.

Wrappers have a number of disadvantages compared to minimal changes or stubs. Wrappers require maintaining two parallel hierarchies, one of legacy interfaces and classes and one of generic interfaces and classes. Conversion by wrapping and unwrapping between these can become tedious. If and when the legacy classes are generified properly, further work will be required to remove the redundant wrappers.

Example 5.5. Evolving a library using wrappers

```
// Don't do this---use of wrappers is not recommended!

l/Stack.java, l/Stacks.java, l/ArrayStack.java:
  // As in Example 5.1

w/GenericStack.java:
  interface GenericStack<E> {
    public Stack unwrap();
    public boolean empty();
    public void push(E elt);
    public E pop();
  }

w/GenericStackWrapper.java:
  @SuppressWarnings("unchecked")
  class GenericStackWrapper<E> implements GenericStack<E> {
    private Stack stack;
    public GenericStackWrapper(Stack stack) { this.stack = stack; }
    public Stack unwrap() { return stack; }
    public boolean empty() { return stack.empty(); }
    public void push(E elt) { stack.push(elt); }
    public E pop() { return (E)stack.pop(); }  // unchecked cast
    public String toString() { return stack.toString(); }
  }

w/GenericStacks.java:
  class GenericStacks {
    public static <T> GenericStack<T> reverse(GenericStack<T> in) {
      Stack rawIn = in.unwrap();
      Stack rawOut = Stacks.reverse(rawIn);
      return new GenericStackWrapper<T>(rawOut);
    }
  }

w/Client.java:
  class Client {
    public static void main(String[] args) {
      GenericStack<Integer> stack
        = new GenericStackWrapper<Integer>(new ArrayStack());
      for (int i = 0; i<4; i++) stack.push(i);
      assert stack.toString().equals("stack[0, 1, 2, 3]");
      int top = stack.pop();
      assert top == 3 && stack.toString().equals("stack[0, 1, 2]");
      GenericStack<Integer> reverse = GenericStacks.reverse(stack);
      assert stack.empty();
      assert reverse.toString().equals("stack[2, 1, 0]");
    }
  }
```

Wrappers also present deeper and subtler problems. If the code uses object identity, problems may appear because the legacy object and the wrapped object are distinct. Further, complex structures will require multiple layers of wrappers. Imagine applying this technique to a stack of stacks! You would need to define a two-level wrapper, that wraps or unwraps each second-level stack as it is pushed onto or popped from the top-level stack. Because wrapped and legacy objects are distinct, it may be hard or even impossible to always ensure that the wrapped objects view all changes to the legacy objects.

The design of Java generics, by ensuring that legacy objects and generic objects are the same, avoids all of these problems with wrappers. The design of generics for C# is very different: legacy classes and generic classes are completely distinct, and any attempt to combine legacy collections and generic collections will bump into the difficulties with wrappers discussed here.

5.5 Conclusions

To review, we have seen both generic and legacy versions of a library and client. These generate equivalent class files, which greatly eases evolution. You can use a generic library with a legacy client, or a legacy library with a generic client. In the latter case, you can update the legacy library with generic method signatures, either by minimal changes to the source or by use of stub files.

The foundation stone that supports all this is the decision to implement generics by erasure, so that generic code generates essentially the same class files as legacy code—a property referred to as binary compatibility. Usually, adding generics in a natural way causes the legacy and generic versions to be binary compatible. However, there are some corner cases where caution is required; these are discussed in Section 8.4.

It is interesting to compare the design of generics in Java and in C#. In Java, generic types do not carry information about type parameters at run time, whereas arrays do contain information about the array element type at run time. In C#, both generic types and arrays contain information about parameter and element types at run time. Each approach has advantages and disadvantages. In the next chapter, we will discuss problems with casting and arrays that arise because Java does not reify information about type parameters, and these problems do not arise in C#. On the other hand, evolution in C# is much more difficult. Legacy and generic collection classes are completely distinct, and any attempt to combine legacy collections and generic collections will encounter the difficulties with wrappers discussed earlier. In contrast, as we've seen, evolution in Java is straightforward.