7 Type Systems

7.3.5 Generics in C++, Java, and C#

Though templates were not officially added to C++ until 1990, when the language was almost ten years old, they were envisioned early in its evolution. C# generics, likewise, were planned from the beginning, though they actually didn't appear until the 2.0 release in 2004. By contrast, generics were deliberately omitted from the original version of Java. They were added to Java 5 (also in 2004) in response to strong demand from the user community.

C++ Templates

EXAMPLE 7.56
Generic `arbiter` class in C++

Figure C-7.6 defines a simple generic class in C++ that we have named an `arbiter`. The purpose of an `arbiter` object is to remember the "best instance" it has seen of some generic parameter class `T`. We have also defined a generic `chooser` class that provides an `operator()` method, allowing it to be called like a function. The intent is that the second generic parameter to `arbiter` should be a subclass of `chooser`, though as written the code does not enforce this. Given these definitions we might write

```
class case_sensitive : chooser<string> {
public:
    bool operator()(const string& a, const string& b) { return a < b; }
};
...
arbiter<string, case_sensitive> cs_names;       // declare new arbiter
cs_names.consider(new string("Apple"));
cs_names.consider(new string("aardvark"));
cout << *cs_names.best() << "\n";               // prints "Apple"
```

Alternatively, we might define a `case_insensitive` descendant of `chooser`, whereupon we could write

```
template<typename T>
class chooser {
public:
    virtual bool operator()(const T& a, const T& b) = 0;
};

template<typename T, typename C>
class arbiter {
    T* best_so_far;
    C comp;
public:
    arbiter() { best_so_far = nullptr; }
    void consider(T* t) {
        if (!best_so_far || comp(*t, *best_so_far)) best_so_far = t;
    }
    T* best() {
        return best_so_far;
    }
};
```

Figure 7.6 Generic arbiter in C++.

```
arbiter<string, case_insensitive> ci_names;     // declare new arbiter
ci_names.consider(new string("Apple"));
ci_names.consider(new string("aardvark"));
cout << *ci_names.best() << "\n";               // prints "aardvark"
```

Either way, the C++ compiler will create a new instance of the `arbiter` template every time we declare an object (e.g., `cs_names`) with a different set of generic arguments. Only at the point where we attempt to use such an object (e.g., by calling `consider`) will it check to see whether the arguments support all the required operations.

Because type checking is delayed until the point of use, there is nothing magic about the `chooser` class. If we neglected to define it, and then left it out of the header of `case_sensitive` (and similarly `case_insensitive`), the code would still compile and run just fine. ■

C++ templates are an extremely powerful facility. Template parameters can include not only types, but also values of ordinary (nongeneric) types, and nested template instances. Programmers can also define *specialized* templates that provide alternative implementations for certain combinations of arguments. These facilities suffice to implement recursion, giving programmers the ability, at least in principle, to compute arbitrary functions at compile time (in other words, templates are *Turing complete*). An entire branch of software engineering has grown up around so-called *template metaprogramming*, in which templates are used to persuade the C++ compiler to generate custom algorithms for special circumstances [AG05]. As a comparatively simple example, one can write a template that accepts a generic parameter `int n` and produces a sorting routine for *n*-element arrays in which all of the loops have been completely unrolled.

As described in Section 7.3.4 ("Implicit Instantiation"), C++ allows generic parameters to be *inferred* for generic functions, rather than specified explicitly. To identify the right version of a generic function (from among an arbitrary number of specializations), and to deduce the corresponding generic arguments, the compiler must perform a complicated, potentially recursive pattern-matching operation. This pattern matching is, in fact, quite similar to the type inference of ML-family languages, described in Section 7.4. It can, as noted in Sidebar 7.8, be cast as *unification*.

EXAMPLE 7.57

Template function bodies moved to a `.cc` file

Unfortunately, per-use instantiation of templates has several significant drawbacks. First, it requires that the compiler have access to the template's source code at the point in the program where instantiation occurs. In the code of Figure C-7.6, the `arbiter` class includes complete definitions of its methods. This is entirely appropriate for small, simple classes, even in a header (`.h`) file. If the code were significantly more complex, we might wish to put only the declaration of the generic class in our header file (call it `arbiter.h`), and defer the method definitions to a separate `arbiter.cc` file:

```
// arbiter.h:

template<typename T, typename C>
class arbiter {
    T* best_so_far;
    C comp;
public:
    arbiter();
    void consider(T* t);
    T* best();
};

// arbiter.cc (imagine that these methods were long and complicated):

template<class T, class C>
arbiter<T,C>::arbiter() { best_so_far = nullptr; }

template<class T, class C>
void arbiter<T,C>::consider(T* t) {
    if (!best_so_far || comp(*t, *best_so_far)) best_so_far = t;
}

template<class T, class C>
T* arbiter<T,C>::best() { return best_so_far; }
```

Compilation units that have access to the `.h` file will still compile successfully, but now machine code for the `arbiter` methods will never be instantiated, because no actual use of an `arbiter` object appears in the file (`arbiter.cc`) that contains the source code. The likely symptom will be "missing symbol" errors from the linker.

C++ provides a partial solution to this problem, in the form of *explicit instantiation*. If we anticipate the need for case-sensitive and case-insensitive `string` arbiters, we can define the appropriate `chooser` classes in `arbiter.h`, and then instantiate corresponding `arbiter` classes in `arbiter.cc`:

```
template class arbiter<string, case_sensitive>;
template class arbiter<string, case_insensitive>;
```

Of course, explicit instantiation works only if the implementor of a template's `.cc` file knows what instantiations will eventually be required. If this cannot be anticipated, the bodies will need to remain in the `.h` file, regardless of their complexity. But then a second problem arises: if the same template is instantiated with the same arguments in 20 different compilation units, the compiler will end up compiling the same code 20 times. Most modern linkers are smart enough to keep only one copy of the machine code for a repeatedly instantiated template, but we will have wasted not only the cost of repeated scanning and parsing, but of semantic analysis, optimization, and code generation as well.

EXAMPLE 7.58
`extern` templates in C++11

C++11 provides a partial solution to this second problem, in the form of `extern` template declarations. If the templated class declaration and method definitions of Example C-7.57 were included in their entirety in `arbiter.h`, and we then needed a case-sensitive `arbiter` in each of 20 `.cc` files, we could write

```
extern template class arbiter<string, case_sensitive>;
```

in all but one of the files, instructing the compiler *not* to generate machine code for that `arbiter`, but rather to assume that an appropriate implementation would be generated elsewhere (presumably in the 20th file, where the `extern` keyword would be omitted), and would thus be available at link time.

EXAMPLE 7.59
Instantiation-time errors in C++ templates

Historically, the final and perhaps the most frustrating problem with per-use instantiation was its tendency to result in inscrutable error messages. Continuing our running example, if we define

```
class foo {                                // line 40 of source
public:
    bool operator()(const string& a, const unsigned int b) {
        // wrong type for second parameter, from arbiter's point of view
        return a.length() < b;
    }
};
```

and then say

```
arbiter<string, foo> oops;
...
oops.consider(new string("Apple"));        // line 75 of source
```

the GNU C++ compiler (version 12.2) will respond with

```
best.cc: In instantiation of ‘void arbiter<T, C>::consider(T*)
[with T = std::__cxx11::basic_string<char>; C = foo]’:
best.cc:75:18:   required from here
best.cc:28:33: error: no match for call to ‘(foo)
    (std::__cxx11::basic_string<char>&, std::__cxx11::basic_string<char>&)’
   28 |         if (!best_so_far || comp(*t, *best_so_far)) best_so_far = t;
      |                             ~~~~^~~~~~~~~~~~~~~~~
best.cc:42:10: note: candidate: ‘bool foo::operator()(const std::string&,
    unsigned int)’
   42 |     bool operator()(const string& a, const unsigned int b) {
      |          ^~~~~~~
best.cc:42:57: note:   no known conversion for argument 2
    from ‘std::__cxx11::basic_string<char>’ to ‘unsigned int’
   42 |     bool operator()(const string& a, const unsigned int b) {
      |                                      ~~~~~~~~~~~~~~~~~~~^
```

LLVM’s `clang` front end (version 14.0) is only a little more helpful:

```
best.cc:28:29: error: no matching function for call to object of type 'foo'
        if (!best_so_far || comp(*t, *best_so_far)) best_so_far = t;
                            ^~~~
best.cc:75:10: note: in instantiation of member function
'arbiter<std::string, foo>::consider' requested here
    oops.consider(new string("Apple"));         // line 75 of source
         ^
best.cc:42:10: note: candidate function not viable: no known conversion
from 'std::string' to 'const unsigned int' for 2nd argument
    bool operator()(const string& a, const unsigned int b) {
         ^
```

The problem here is fundamental; it’s not poor compiler design. Because the language requires that templates be “expanded out” before they are type checked, it is extraordinarily difficult to generate messages without reflecting that expansion. ■

To facilitate more helpful messages (and also to increase the expressive power of template specialization), C++20 introduced the notion of *concepts*, which allow the programmer to specify constraints on template parameters—and the compiler to check those constraints at instantiation time. Dozens of concepts and related *requirements* are defined in the standard library, and rich notation is available in which to define additional concepts.

EXAMPLE 7.60
Insisting on a `chooser`

Perhaps the simplest constraint we might add to our `arbiter` type insists that type parameter `C` be derived from `chooser<T>`:

```
template<typename T, typename C>
    requires std::derived_from<C, chooser<T>>
class arbiter { ...
```

With this change in place, `clang` says

```
best.cc:74:5: error: constraints not satisfied for class template 'arbiter'
[with T = std::string, C = foo]
    arbiter<string, foo> oops;
    ^~~~~~~~~~~~~~~~~~~~
best.cc:18:14: note: because 'std::derived_from<foo, chooser<std::string> >'
evaluated to false
    requires std::derived_from<C, chooser<T>>
             ^
```

Note that the error message is now associated with the instantiation of `arbiter` at line 74, rather than its use at line 75. ■

EXAMPLE 7.61
Insisting on a predicate

But this is overkill. It rules out cases in which we provide a perfectly usable comparator type `C` that isn't actually derived from `chooser<T>`. To allow more general comparators, we might write

```
template<typename T, typename C>
    requires std::predicate<C, T, T>
class arbiter { ...
```

Here `predicate<C, T, T>` requires that `C` be an invocable object that takes two parameters of type `T` and returns a value whose type is (or can be coerced to be) `bool`. With this revised constraint, error messages still occur at the point of instantiation but we are no longer limited to strict descendants of the `chooser` type. ■

EXAMPLE 7.62
Defining a new named concept

Unfortunately, the error message for a failed instantiation of Example C-7.61 (a message we haven't shown) now fills most of a page, diving into details of the definition of `std::predicate`. To improve this message, we can define a concept that captures exactly what we need:

```
template<typename T, typename C>
concept Compares =
    requires(C c, T a, T b) { {c(a, b)} -> std::convertible_to<bool>; };

template<typename T, typename C>
    requires Compares<T, C>
class arbiter { ...
```

Here we have defined `Compares` to insist that for any `C` object `c` and any `T` objects `a` and `b`, the expression `c(a, b)` is well formed and has a value whose type is (or can be coerced to be) `bool`. Now our error message is quite nice:

```
best.cc:74:5: error: constraints not satisfied for class template 'arbiter'
[with T = std::string, C = foo]
    arbiter<string, foo> oops;
    ^~~~~~~~~~~~~~~~~~~~
best.cc:19:14: note: because 'Compares<foo, std::string>' evaluated to false
    requires Compares<C, T>
             ^
```

```
best.cc:15:32: note: because 'c(a, b)' would be invalid: no matching
function for call to object of type 'foo'
    requires(C c, T a, T b) { {c(a, b)} -> std::convertible_to<bool>; };
                               ^
```

Java Generics

Generics were deliberately omitted from the original version of Java. Rather than instantiate containers with different generic parameter types, Java programmers followed a convention in which all objects in a container were assumed to be of the standard base class `Object`, from which all other classes are descended. Users of a container could place any type of object inside. When removing an object, an explicit conversion (what Java calls a `cast`) could be used to reassert the original type. No danger was involved, because objects in Java are self-descriptive, and conversions employ run-time checks.

Though dramatically simpler than the use of templates in C++, this programming convention has three significant drawbacks: (1) users of containers must litter their code with conversions, which many people find distracting or aesthetically distasteful; (2) errors in the use of a container manifest themselves as `ClassCastExceptions` at run time, rather than as compile-time error messages; (3) the error checking of the conversions incurs overhead at run time. Given Java's emphasis on clarity of expression, rather than pure performance, problems (1) and (2) were considered the most serious, and became the subject of a Java Community Process proposal for a language extension in Java 5. The solution adopted is based on the GJ (Generic Java) work of Bracha et al. [BOSW98].

EXAMPLE 7.63
Generic `Arbiter` class in Java

Figure C-7.7 contains a Java version of our `arbiter` class. It differs from the C++ code of Figure C-7.6 in several important ways. First, Java requires that the code for each generic class be manifestly (self-obviously) type safe, independent of any particular instantiation. This means that the type of field `comp`—and in particular, the fact that it provides a `better` method—must be statically declared. As a result, the `Chooser` to be used by a given `Arbiter` instance must be specified as a constructor parameter; it cannot be a generic parameter. (We could have used a constructor parameter in C++; in Java it is mandatory.) For both field `comp` and constructor parameter `c`, we are then faced with the question: what should be the generic parameter of `Chooser`?

The most obvious choice (*not* the one adopted in Figure C-7.7) would be `Chooser<T>`. This would allow us to write

```
class CaseSensitive implements Chooser<String> {
    public boolean better(String a, String b) {
        return a.compareTo(b) < 1;
    }
}
```

```
interface Chooser<T> {
    public boolean better(T a, T b);
}

class Arbiter<T> {
    T bestSoFar;
    Chooser<? super T> comp;

    public Arbiter(Chooser<? super T> c) {
        comp = c;
    }
    public void consider(T t) {
        if (bestSoFar == null || comp.better(t, bestSoFar)) bestSoFar = t;
    }
    public T best() {
        return bestSoFar;
    }
}
```

Figure 7.7 Generic arbiter in Java.

```
...
Arbiter<String> csNames = new Arbiter<String>(new CaseSensitive());
csNames.consider(new String("Apple"));
csNames.consider(new String("aardvark"));
System.out.println(csNames.best());            // prints "Apple"
```
■

EXAMPLE 7.64
Wildcards and bounds on Java generic parameters

Suppose, however, we were to define

```
class CaseInsensitive implements Chooser<Object> {   // note type!
    public boolean better(Object a, Object b) {
        return a.toString().compareToIgnoreCase(b.toString()) < 1;
    }
}
```

Class `Object` defines a `toString` method (usually used for debugging purposes), so this declaration is valid. Moreover since every `String` is an `Object`, we ought to be able to pass any pair of strings to `CaseInsensitive.better` and get a valid response. Unfortunately, `Chooser<Object>` is not acceptable as a match for `Chooser<String>`. If we typed

```
Arbiter<String> ciNames = new Arbiter<String>(new CaseInsensitive());
```

the compiler would complain. The fix (as shown in Figure C-7.7) is to declare both `comp` and `c` to be of type `<? super T>` instead. This informs the Java compiler that an arbitrary type argument ("`?`") is acceptable as the generic parameter of our `Chooser`, so long as that type is an ancestor of `T`.

```
interface Chooser {
    public boolean better(Object a, Object b);
}

class Arbiter {
    Object bestSoFar;
    Chooser comp;

    public Arbiter(Chooser c) {
        comp = c;
    }
    public void consider(Object t) {
        if (bestSoFar == null || comp.better(t, bestSoFar)) bestSoFar = t;
    }
    public Object best() {
        return bestSoFar;
    }
}
```

Figure 7.8 **Arbiter in Java after type erasure.** Conversions will be inserted by the compiler on calls that return an `Object` or that expect an `Object` to support a particular method.

The `super` keyword specifies a *lower bound* on a type parameter. It is the symmetric opposite of the `extends` keyword, which we used in Example 7.39 to specify an *upper bound*. Together, upper and lower bounds allow us to broaden the set of types that can be used to instantiate generics. As a general rule, we use `extends T` whenever a method returns a `T` object (on which we need to be able to invoke `T` methods); we use `super T` whenever we expect to pass a `T` object as a parameter, but don't mind if the receiver is willing to accept something more general. Given the bounded declarations of Figure C-7.7, our use of `CaseInsensitive` will compile and run just fine:

```
Arbiter<String> ciNames = new Arbiter<String>(new CaseInsensitive());
ciNames.consider(new String("Apple"));
ciNames.consider(new String("aardvark"));
System.out.println(ciNames.best());            // prints "aardvark"
```

Type Erasure

EXAMPLE 7.65
Type erasure and implicit conversions

Generics in Java are defined in terms of *type erasure*: the compiler effectively deletes every generic parameter and argument list, replaces every occurrence of a type parameter with `Object`, and inserts conversions back to concrete types wherever objects are returned from generic methods. The erased equivalent of Figure C-7.7 appears in Figure C-7.8. No conversions are required in this portion of the code. On any use of `best`, however, the compiler would insert an implicit conversions. The statement

```
String winner = csNames.best();
```

will, in effect, be implicitly replaced with

```
String winner = (String) csNames.best();
```

Also, in order to match the `Chooser<String>` interface, our definition of `CaseSensitive` (Example C-7.63) will in effect be replaced with

```
class CaseSensitive implements Chooser {
    public boolean better(Object a, Object b) {
        return ((String) a).compareTo((String) b) < 1;
    }
}
```

The advantage of type erasure over the nongeneric (`Object`-based) version of the code is that the programmer doesn't have to write the conversions. In addition, the compiler is able to verify in most cases that the erased code will never generate a `ClassCastException` at run time. The exceptions occur primarily when, for the sake of interoperability with preexisting code, the programmer assigns a generic collection into a nongeneric collection:

EXAMPLE 7.66

`Unchecked` warnings in Java

```
Arbiter<String> csNames = new Arbiter<String>(new CaseSensitive());
Arbiter alias = csNames;                // nongeneric
alias.consider(Integer.valueOf(3));     // unsafe
```

DESIGN & IMPLEMENTATION

7.11 Why erasure?

Erasure in Java has several surprising consequences. For one, we can't invoke `new T()`, where `T` is a type parameter: the compiler wouldn't know what kind of object to create. Similarly, Java's *reflection* mechanism, which allows a program to examine and reason about the concrete type of an object at run time, knows nothing about generics: `csNames.getClass().toString()` returns `"class Arbiter"`, not `"class Arbiter<String>"`. Why would the Java designers introduce a mechanism with such significant limitations? The answer is backward compatibility or, more precisely, *migration* compatibility, which requires complete interoperability of old and new code.

More so than most previous languages, Java encourages the assembly of working programs, often on the fly, from components written independently by many different people in many different organizations. The Java designers felt it was critical not only that old (nongeneric) programs be able to run with new (generic) libraries, but also that new (generic) programs be able to run with old (nongeneric) libraries. In addition, they took the position that the Java virtual machine, which interprets Java bytecode in the typical implementation, could not be modified. While one can take issue with these goals, once they are accepted erasure becomes a natural solution.

The compiler will issue an "unchecked" warning on the third line of this example, because we have invoked method `consider` on a "raw" (nongeneric) `Arbiter` without explicitly converting the arguments. In this case the warning is clearly warranted: `alias` *shouldn't* be passed an `Integer`. Other examples can be quite a bit more subtle. It should be emphasized that the warning simply indicates the lack of *static* checking; any type errors that actually occur will still be caught at run time. ■

EXAMPLE 7.67
Java generics and built-in types

Note, by the way, that the use of erasure, and the insistence that every instance of a given generic be able to share the same code, means that type arguments in Java must all be descended from `Object`. While `Arbiter<Integer>` is a perfectly acceptable type, `Arbiter<int>` is not. ■

C# Generics

Though generics were omitted from C# version 1, the language designers always intended to add them, and the .NET Common Language Infrastructure (CLI) was designed from the outset to provide appropriate support. As a result, C# 2.0 was able to employ an implementation based on *reification* rather than erasure. Reification creates a different concrete type every time a generic is instantiated with different arguments. Reified types are visible to the reflection library (`csNames.GetType().ToString()` returns `"Arbiter`1[System.Double]"`), and it is perfectly acceptable to call `new T()` if `T` is a type parameter with a zero-argument constructor (a constraint to this effect is required). Moreover where the Java compiler must generate implicit type conversions to satisfy the requirements of the virtual machine (which knows nothing of generics) and to ensure type-safe interaction with legacy code (which might pass a parameter or return a result of an inappropriate type), the C# compiler can be sure that such checks will never be needed, and can therefore leave them out. The result is faster code.

EXAMPLE 7.68
Sharing generic implementations in C#

Of course the C# compiler is free to merge the implementations of any generic instantiations whose code would be the same. Such sharing is significantly easier in C# than it is in C++, because implementations typically employ just-in-time compilation, which delays the generation of machine code until immediately prior to execution, when it's clear whether an identical instantiation already exists somewhere else in the program. In particular, `MyType<Foo>` and `MyType<Bar>` will share code whenever `Foo` and `Bar` are both classes, because C# employs a reference model for variables of class type. ■

EXAMPLE 7.69
C# generics and built-in types

Like C++, C# allows generic arguments to be value types (built-ins or `struct`s), not just classes. We are free to create an object of class `MyType<int>`; we do not have to "wrap" it as `MyType<Integer>`, the way we would in Java. `MyType<int>` and `MyType<double>` would generally not share code, but both would run significantly faster than `MyType<Integer>` or `MyType<Double>`, because they wouldn't incur the dynamic memory allocation required to create a wrapper object, the garbage collection required to reclaim it, or the indirection overhead required to access the data inside. ■

Like Java, C# allows only types as generic parameters, and insists that generics be manifestly type safe, independent of any particular instantiation. It generates

```
interface Chooser<in T> {
    bool better(T a, T b);
}

class Arbiter<T> {
    T bestSoFar;
    Chooser<T> comp;
    bool initialized;

    public Arbiter(Chooser<T> c) {
        comp = c;
        bestSoFar = default(T);
        initialized = false;
    }
    public void Consider(T t) {
        if (!initialized || comp.better(t, bestSoFar)) bestSoFar = t;
        initialized = true;
    }
    public T Best() {
        return bestSoFar;
    }
}
```

Figure 7.9 Generic arbiter in C#.

reasonable error messages if we try to instantiate a generic with an argument that doesn't meet the constraints of the corresponding generic parameter, or if we try, inside the generic, to invoke a method that the constraints don't guarantee will be available.

EXAMPLE 7.70
Generic `Arbiter` class in C#

A C# version of our `Arbiter` class appears in Figure C-7.9. One small difference with respect to Figure C-7.7 can be seen in the `Arbiter` constructor, which must explicitly initialize field `bestSoFar` to `default(T)`. We can leave this out in Java because variables of class type are implicitly initialized to `null`, and type parameters in Java are all classes. In C# `T` might be a built-in or a `struct`, both of which require explicit initialization. ■

EXAMPLE 7.71
Contravariance in the `Arbiter` interface

A more interesting difference from Figure C-7.7 appears in the definitions of the `Chooser` interface, the `comp` member of class `Arbiter`, and the `c` parameter of the `Arbiter` constructor. In Java, we used explicit lower bounds (`? super T`) on `comp` and `c` to indicate that any `Chooser<S>`, where `S` is a superclass of `T`, would be acceptable. While C# allows us to specify upper bounds in the form of type constraints (we did so in the `sort` routine of Example 7.40), it has no direct equivalent of lower bounds. It does, however, support the related notions of *covariance* and *contravariance*. We have exploited this support in Figure C-7.9, where it appears not as bounds on the `Chooser` passed to a newly created `Arbiter`, but as an `in` modifier on the generic parameter of the `Chooser` interface itself.

The declaration `interface Chooser<in T>` indicates that objects of class `T` will be used only as input parameters to methods of the interface. Suppose now that S is a superclass of T. Since T provides all the methods of S, any method that expects an input of class S will also accept an input of class T. This means that in any context in which all we do is provide T objects as inputs to a Chooser, we can use a "less choosy" Chooser that merely expects S inputs. In other words, Chooser<T> is a superclass of Chooser<S>. Represented graphically,

```
T → S  ⇒  Chooser<S> → Chooser<T>
```

where the → symbol, pronounced "is a," indicates that the item on the left inherits from the item on the right. Chooser<T> is said to be "*contra*variant in T" because the relationship between S and T is reversed when wrapping them in a Chooser. ■

EXAMPLE 7.72
Covariance

In other situations, objects of a generic type may only be *produced* by the methods of an interface. Consider, for example, the notion of an iterator, as provided by C#'s IEnumerator<T> interface. Method Current of this interface returns an object of class T; no method takes a T object as input. In the C# standard library, the interface is declared as

```
public interface IEnumerator<out T> ...
```

Now suppose again that S is a superclass of T. In any context in which all we do is extract S objects from an IEnumerator, we can use a more specific IEnumerator that gives us T objects instead. In other words, IEnumerator<S> is a superclass of IEnumerator<T>. Graphically,

```
T → S  ⇒  IEnumerator<T> → IEnumerator<S>
```

Here IEnumerator<T> is said to be "*co*variant in T" because the relationship between S and T is preserved when wrapping them in an IEnumerator. In many interfaces, of course, generic parameters appear as both inputs and outputs of methods. For such an interface Foo, there is no subclassing relationship: Foo<T> is said to be "*in*variant in T." ■

EXAMPLE 7.73
Chooser as a delegate

Returning to the Arbiter example, there is actually a simpler way to write our code in C#. Because the Chooser interface has only a single method, we can express it as a *delegate* instead:

```
delegate bool Chooser<T>(T a, T b);
```

Then in method Arbiter.Consider, we can call the delegate directly as comp(t, bestSoFar). Our new Chooser is roughly analogous to the C declaration

```
typedef _Bool (*Chooser)(T a, T b);
```

(pointer to function of two T arguments, returning a Boolean), except that a C# Chooser object is a closure, not a pointer: it can refer to a static function, a method of a particular object (in which case it has access to the object's fields), or an anonymous nested function (in which case it has access, with unlimited extent, to variables in the surrounding scope). In our particular case, defining Chooser to be a delegate allows us to pass any appropriate function to the Arbiter constructor, without regard to the class inheritance hierarchy. We can declare

```
static bool CaseSensitive(String a, String b) {
    return String.CompareOrdinal(a, b) < 1;
    // use Unicode order, in which upper-case letters come first
}
static bool CaseInsensitive(Object a, Object b) {
    return String.Compare(a.ToString(), b.ToString(), false) < 1;
}
```

and then say

```
Arbiter<String> csNames =
    new Arbiter<String>(new Chooser<String>(CaseSensitive));
csNames.Consider("Apple");
csNames.Consider("aardvark");
Console.WriteLine(csNames.Best());              // prints "Apple"

Arbiter<String> ciNames =
    new Arbiter<String>(new Chooser<String>(CaseInsensitive));
ciNames.Consider("Apple");
ciNames.Consider("aardvark");
Console.WriteLine(ciNames.Best());              // prints "aardvark"
```

The compiler is perfectly happy to instantiate `CaseInsensitive` as a `Chooser<String>`, because `String`s can be passed as `Object`s. ■

CHECK YOUR UNDERSTANDING

48. Why was it difficult, historically, to produce high-quality error messages for misuses of C++ templates? How do the *concepts* of C++20 address this problem?
49. What is the purpose of explicit instantiation in C++? What is the purpose of `extern` templates?
50. What is *template metaprogramming*?
51. Explain the difference between *upper bounds* and *lower bounds* in Java type constraints. Which of these does C# support?
52. What is *type erasure*? Why is it used in Java?
53. Under what circumstances will a Java compiler issue an "unchecked" generic warning?
54. Why must fields of generic parameter type be explicitly initialized in C#?
55. For what two main reasons are C# generics often more efficient than comparable code in Java?
56. Summarize the notions of *covariance* and *contravariance* in generic types.

57. How does a C# *delegate* differ from an interface with a single method (e.g., the C++ `chooser` of Figure C-7.6)? How does it differ from a function pointer in C?

7 Type Systems

7.7 Exercises

7.27 C++ has no direct analogue of the `extends X` and `super X` clauses of Java. Why not?

7.28 Write a simple abstract `ordered_set<T>` class (an *interface*) whose methods include `void insert(T val)`, `void remove (T val)`, `bool lookup (T val)`, and `bool is_empty()`, together with a language-appropriate iterator, as described in Section 6.5.3. Using this abstract class as a base, build a simple `list_set` class that uses a sorted linked list internally. Try this exercise in C++, Java, and C#. Note that you will need constraints on `T` in Java and C#. You may also want them in C++. Discuss the differences among your implementations.

7.29 Building on the previous exercise, implement higher-level `union<T>`, `intersection<T>`, and `difference<T>` functions that operate on ordered sets. Note that these should not be members of the `ordered_set<T>` class, but rather stand-alone functions: they should be independent of the details of `list_set` or any other particular `ordered_set`. So, for example, `union(A, B, C)` should verify that `A` is empty, and then add to it all the elements found in `B` or `C`. Explain, for each of C++, Java, and C#, how to handle the comparison of elements.

7.30 Continuing Example C-7.63, the call

```
csNames.consider(null);
```

will generate a run-time exception, because `String.compareTo` is not designed to take `null` arguments.

(a) Modify Figure C-7.7 to guard against this possibility by including a predicate `public Boolean valid(T a);` in the `Chooser<T>` interface, and by modifying `consider` to make an appropriate call to this predicate. Modify class `CaseSensitive` accordingly.

(b) Suggest how to make similar modifications to the C# `Arbiter` of Figure C-7.9 and Example C-7.70. How should you handle lower bounds when you need both `Better` and `Valid`?

7.31 (a) Modify your solution to Exercise 7.15 so that the comparison routine is an explicit generic parameter, reminiscent of the `chooser` of Figure C-7.6.

(b) Give an alternative solution in which the comparison routine is an extra parameter to `sort`.

7.32 Consider the C++ program shown in Figure C-7.10. Explain why the final call to `first_n` generates a compile-time error, but the call to `last_n` does not. (Note that `first_n` is generic but `last_n` is not.) Show how to modify the final call to `first_n` so that the compiler will accept it.

7.33 Consider the following code in C++:

```
template <typename T>
class cloneable_list : public list<T> {
public:
    cloneable_list<T>* clone() {
        auto rtn = new cloneable_list<T>();
        for (auto e : *this) {
            rtn->push_back(e);
        }
        return rtn;
    }
};

...
cloneable_list<foo> L;
...
cloneable_list<foo>* Lp = L.clone();
```

Here `*Lp` will be a “deep copy” of `L`, containing a copy of each `foo` object. Try to write equivalent code in Java. What goes wrong? How might you get around the problem?

```
#include <iostream>
#include <list>
using std::cout;
using std::list;

template<typename T> void first_n(list<T> p, int n) {
    for (typename list<T>::iterator li = p.begin(); li != p.end(); li++) {
        if (n-- <= 0) break;
        cout << *li << " ";
    }
    cout << "\n";
}

void last_n(list<int> p, int n) {
    for (list<int>::reverse_iterator li = p.rbegin(); li != p.rend(); li++) {
        if (n-- <= 0) break;
        cout << *li << " ";
    }
    cout << "\n";
}

class int_list_box {
    list<int> content;
public:
    int_list_box(list<int> l) { content = l; }
    operator list<int>() { return content; }
        // user-supplied operator for coercion/conversion
};

int main() {
    int i = 5;
    list<int> l;

    for (int i = 0; i < 10; i++) l.push_back(i);
    int_list_box b(l);

    first_n(l, i);      // works
    last_n(b, i);       // works (coerces b)
    first_n(b, i);      // static semantic error
}
```

Figure 7.10 **Coercion and generics in C++.** The compiler refuses to accept the final call to `first_n`.

7 Type Systems

7.8 Explorations

7.44 Learn more about *concepts* in C++, together with the earlier notions of *named requirements* and the "substitution failure is not an error" (SFINAE) idiom. Compare and contrast concepts with the constraint mechanisms of Java and C#.

7.45 Explore the support for generics in Scala, Eiffel, Ada, or some other programming language. Compare this support to that of C++, Java, and C#. What might account for the differences? Which approach(es) do you prefer? Why?

7.46 Explore more fully the concepts of *covariance* and *contravariance* in object-oriented languages, as exemplified by the `in` and `out` modifiers for generic parameters in C# 4.0. Discuss the connection between these concepts and the notions of upper and lower bounds on generic parameters (`? extends T` and `? super T` in Java).