08장 서브루틴과 제어 추상화

【8.4.4】 C++, 자바와 C#에서의 제네릭

1990년대까지 템플릿을 공식적으로 C++에 추가하지 않았지만 그 언어가 나온 지 거의 10년이 됐을 때 C의 발전에 있어서 템플릿의 추가를 일찍 계획했다. 마찬가지로 C#의 제네릭을 초반부부터 계획했다. 하지만 2004년에 2.0 릴리즈까지 그것을 볼 수 없었다. 반대로 자바의 원래 버전에서 제네릭을 고의적으로 생략했다. 사용자 모임에서의 강한 요구에 응해 자바 5에 추가했다.

C++ 템플릿

예 8.65
C++에서의 제네릭 arbiter 클래스

©(심화학습에 있는) 그림 8.11은 `arbiter`라 이름 붙인 C++에서의 간단한 제네릭 클래스를 정의한다.

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

template<class T, class C>
class arbiter {
    T* best_so_far;
    C comp;
```

그림 8.11 | C++에서의 제네릭 arbiter(이어짐)

```
public:
    arbiter() { best_so_far = 0; }
    void consider(T* t) {
       if (!best_so_far || comp(*t, *best_so_far)) best_so_far = t;
    }
    T* best() {
       return best_so_far;
    }
};
```

그림 8.11 | C++에서의 제네릭 arbiter

arbiter 객체의 목적은 제네릭 매개변수 클래스 T의 관찰해왔던 "가장 좋은 인스턴스"를 기억하는 것이다. 또한 operator() 메소드를 제공하는 제네릭 chooser 클래스를 정의해 함수와 같이 그것을 호출하게 한다. 그 의도는 이를 강력히 주장하지는 않지만 arbiter의 두 번째 제네릭 매개변수가 chooser의 하위 클래스이어야 함이다. 이런 정의가 주어지면 다음과 같이 작성할 수 있다.

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

다른 방법으로 chooser의 case_insensitive의 후속 버전을 저장할 수 있다. 그래서 아래와 같이 작성한다.

```
arbiter<string, case_sensitive> ci_names;     // 새로운 arbiter를 선언
ci_names.consider(new string("Apple"));
ci_names.consider(new string("aardvark"));
cout << *ci_names.best() << "\n";             // "aardvark"를 출력
```

C++ 컴파일러는 다른 집합의 제네릭 인자를 가진 객체(예를 들어 cs_names)를 선언할 때마다 arbiter 템플릿의 새로운 인스턴스를 생성할 것이다. 그런 객체를 사용하려 할 때만(예를 들어 consider를 호출해) 그것은 인자가 모든 요구된 연산을 지원하는지를 검사할 것이다.

유형 검사를 사용 시점까지 지연시키기 때문에 chooser 클래스에 대해 신비한 뭔가

는 없다. 이를 정의하는 것을 간과한 다음 그것을 case_sensitive(그리고 case_inseneitive)의 헤더 밖에 남겨둔다 해도 그 코드는 여전히 컴파일되고 잘 실행될 것이다.

C++ 템플릿은 매우 강력한 편의 기능이다. 템플릿 매개변수는 유형뿐만 아니라 일반적인(제네릭이 아닌) 유형의 값과 중첩된 템플릿 선언도 포함할 수 있다. 프로그래머도 인자의 특정 조합을 위한 대안적인 구현을 제공하는 전문화된 템플릿을 정의할 수 있다. 이 편의 기능은 재귀를 구현하기에 충분해서 적어도 이론적으로는 프로그래머에게 컴파일 시에 임의의 함수를 계산할 수 있는 능력을 준다. 소프트웨어 공학의 전 부문은 소위 템플릿 메타프로그래밍에서 잔뼈가 굵다. 거기서는 특별한 상황에 대한 맞춤 알고리즘을 만들게 C++ 컴파일러를 설득하기 위해 템플릿을 사용한다[AG90]. 상대적으로 간단한 예로서 제네릭 매개변수 int n을 받아들이고 루프의 모든 것을 완전히 전개한 n개의 원소를 가진 배열에 대한 정렬 루틴을 만드는 템플릿을 작성할 수 있다.

예 8.66

C++ 템플릿의 인스턴트화 시 오류

불행히도 템플릿을 사용할 때마다의 인스턴트화는 두 가지 결점을 가지고 있다. 첫째, 수수께끼와 같은 오류 메시지를 만들어내는 경향이 있다. 다음과 같이 정의하고

```
class foo {
public:
    bool operator()(const string& a, const unsigned int b)
       // arbiter의 관점에서는 두 번째 매개변수에 대한 잘못된 유형임
       { return a.length() < b; }
};
```

그러고 나서 다음과 같이 말한다면

```
arbiter<string, foo> oops;
...
oops.consider(new string("Apple"));     // 소스의 65행
```

"65행: foo의 operator() 메소드는 유형 string&를 가진 두 인자를 취할 필요가 있음"을 포함한 오류 메시지를 받기를 희망할지도 모른다. 대신 GNU C++ 컴파일러는 아래와 같이 응답한다.

```
simple_best.cc: In member function 'void arbiter<T, C>::consider(T*)
    [with T = std::string, C = foo]':
simple_best.cc:65: instantiated from here
simple_best.cc:21: error: no match for call to '(foo)
    (std::basic_string<char, std::char_traits<char>,
    std::allocator<char> >&, std::basic_string<char,
    std::char_traits<char>, std::allocator<char> >&)"
```

(21행은 ©심화학습에 있는 그림 8.11에서 메소드 consider의 몸체다)

Sun의 C++ 컴파일러도 마찬가지로 도움이 되지 않는다.

```
"simple_best.cc", line 21: Error: Cannot cast from std::basic_string<char,
    std::char_traits<char>, std::allocator<char>> to const int.
"simple_best.cc", line 65: Where: While instantiating
    "arbiter<std::basic_string<char, std::char_traits<char>,
    std::allocator<char>>, foo>::consider(std::basic_string<char,
    std::char_traits<char>, std::allocator<char>>*)".
"simple_best.cc", line 65: Where: Instantiated from non-template code.
```

여기서 문제는 근본적이다. 즉, 나쁜 컴파일러 설계 때문이 아니다. 언어가 유형 검사를 하기 전에 템플릿을 확장하게 요구하기 때문에 그 확장을 반영하지 않고 메시지를 생성하기 매우 어렵다.

사용 시마다 인스턴트화의 두 번째 결점은 "코드 팽창"에 대한 경향이다. 별개 컴파일의 존재 때문에 다른 컴파일 단위에서 동일한 인자를 가지고 동일한 템플릿을 인스턴트화하는 것을 인식하는 것은 어려울 수 있다. 20개 조각에서 컴파일된 프로그램은 자주 사용하는 템플릿 인스턴스의 사본을 20개 가지고 있다.

자바 제네릭

자바의 초기 버전에서는 의도적으로 제네릭을 뺐다. 별개의 제네릭 매개변수 유형을 가진 컨테이너를 인스턴트화하기보다는 자바 프로그래머는 컨테이너 내의 모든 객체가 다른 모든 클래스의 부모 클래스인 표준 기본 클래스 `Object`의 유형임을 가정하는 규정을 따랐다. 컨테이너의 사용자는 어떤 유형의 객체라도 안에 위치시킬 수 있다. 하지만 객체를 제거할 때 원래 유형을 다시 주장하는 변환이 필요하다. 자바의 객체가 자가 기술적이고 변환이 실행 시간 검사를 이용하기 때문에 어떤 위험도 중첩하고 있지는 않다.

C++에서의 템플릿 사용보다 훨씬 더 단순하지만 이런 프로그램 규정은 3가지 중요한 결점을 가지고 있다. (1) 컨테이너의 사용자는 변환으로 코드를 어지럽혀야 한다. 이 변환은 많은 사람이 혼란스럽다고 하거나 심미적으로 선호하지 않는다. (2) 컨테이너의 사용에 있어서 오류는 컴파일 시 오류 메시지가 아닌 실행 시간에 `ClassCastException`으로서 스스로를 나타낸다. (3) 변환은 실행 시간에 오버헤드를 초래한다. 자바가 순수 성능이 아닌 수식의 명확성을 강조한다는 사실은 문제 (1)과 (2)를 가장 심각하게 고려할 것이고 이는 자바 5에서의 언어 확장을 위한 자바 Community Process(역자 주: JCP라고 하는 자바 개발자 모임 사이트) 제안의 주제가 됐다. 채택된 해결책은 브라하 등의 GJ(Generic Java) 연구에 기반한다[BOSW98].

예 8.67

자바에서의 제네릭 arbiter 클래스

©(심화학습에 있는) 그림 8.12는 앞서 다룬 arbiter 클래스의 자바 5 버전이다.

```
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;
    }
}
```

그림 8.12 | 자바의 제네릭 arbiter

©(심화학습에 있는) 그림 8.11과 몇 가지 점에서 차이가 난다. 첫째, 자바는 제네릭의 모든 인스턴스가 동일한 코드를 공유할 수 있게 요구한다. 다른 것보다 이는 주어진 인스턴스가 사용하는 Chooser 클래스를 생성자 매개변수로 지정해야 한다는 것을 의미한다. 그것은 제네릭 매개변수가 될 수 없다(C++에서 생성자 매개변수를 사용할 수 있다. 하지만 자바에서 그것은 의무적이다). 둘째, 자바는 모든 특정 인스턴스화와 별개로 arbiter 클래스에 대한 코드가 명백히 유형적으로 안전적이다. 그러므로 comp를 반드시 Chooser가 되게 선언해야 한다. 그래서 그것이 better 메소드를 제공함을 안다. 이는 어떤 종류의 Chooser가 필요한가, 즉 comp의 선언에서 제네릭 매개변수는 무엇이어야 하는가(그리고 어울리는 생성자 매개변수는 어떤 유형이어야 하는가)라는 문제를 제기한다.

가장 명백한 선택은 Chooser<T>다. 이는 다음과 같이 작성하게 해준다.

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

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

예 8.68

자바의 제네릭 매개변수에 관한 예측할 수 없는 요인과 한계

하지만 다음과 같이 정의하고 있다고 가정해보자.

```
static class CaseInsensitive implements Chooser<Object> { // 유형을 기억하자!
    public Boolean better(Object a, Object b) {
       return a.toString().compareToIgnoreCase(b.toString()) < 1;
    }
}
```

클래스 Object는 (일반적으로 디버깅을 목적으로 사용되는) toString 메소드를 정의한다. 그래서 이 선언은 유효하다. 게다가 모든 String이 Object기 때문에 CaseInsensitive.better에 어떤 한 쌍의 문자열이라도 전달하고 유효한 응답을 얻을 수 있어야 한다. 불행히도 Chooser<String>과 어울리는 것으로 Chooser<Object>를 받아들이지 않을 것이다. 이것은 ©(심화학습에 있는) 그림 8.12에서 2번 사용한 <? super T> 유형 매개변수 배후의 이론적 근거다. 그것은 자바 컴파일러를 임의의 유형 매개변수("?")가 그 유형이 T의 부모 클래스인 한 Chooser의 제네릭 매개변수로 받아들일 수 있음을 알려준다.

```
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;
    }
}
```

그림 8.13 | 자바에서 유형 말소 후의 arbiter. 코드의 이 부분에서 어떤 변환도 필요하지 않다(하지만 사용을 위한 주요 텍스트에서는 발견할 수 있다).

super 키워드는 유형 매개변수의 하한을 지정한다. extends 키워드와 정확히 반대다. 상한을 지정하기 위해 예 8.37에서 이를 사용했다. 또한 상한과 하한은 제네릭을 인스턴스화하기 위해 사용할 수 있는 유형의 집합을 확장할 수 있게 해준다. 일반적인 규칙으로서 T 메소드를 호출할 때마다 extends T를 사용한다. T 메소드를 호출할 필요가 있을 때마다 super T를 사용한다. 하지만 수신자가 좀 더 일반적인 무언가를 받아들이려고 하는지를 염두에 두지 않는다. (심화학습에 있는) 그림 8.12에서 사용된 제한된 선언이 주어져있으면 다음은 컴파일되고 아무 문제없이 실행될 것이다.

유형 말소

예 8.69

유형 말소와 묵시적인 변환

자바에서는 유형 말소에 의해 제네릭를 정의한다. 컴파일러는 효과적으로 모든 제네릭 매개변수와 인자 목록을 삭제하고 Object와 관련된 유형 매개변수의 모든 발생을 대체하고 제네릭 메소드에서 객체를 반환하는 곳마다 구체화된 유형으로 돌아가는 변환을 삽입한다. (심화학습에 있는) 그림 8.12의 유형 말소 버전은 (심화학습에 있는) 그림 8.13에 나와 있다. 이 코드 부분에서는 어떤 변환도 필요치 않다. 하지만 best의 모든 사용에서 컴파일러는 묵시적인 변환을 삽입할 것이다. 사실

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

코드는

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

를 묵시적으로 대체할 것이다. 또한 Chooser<String> 인터페이스와 어울리기 위해 CaseSensitive(심화학습에 있는 예 8.67)의 정의를 사실상 다음으로 대체해야 할 것이다.

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

예 8.70

자바 5에서 unchecked 경고

제네릭이 아닌 버전의 코드와 비교한 유형 말소의 이점은 프로그래머가 변환을 작성하지 않아도 된다는 것이다. 추가적으로 컴파일러는 대부분의 경우에서 유형 말소 코드가 결코 ClassCastException을 실행 시간에 발생시키지 않음을 증명할 수 있다. 이미 존재하는 코드와의 상호호환성을 목적으로 프로그래머가 제네릭 컬렉션에 제네릭이 아닌 컬렉션을 대입했을 때 예외가 주로 발생한다.

```
Arbiter<String> csNames = new Arbiter<String>(new CaseSensitive());
Arbiter alias = csNames;              // 제네릭이 아닌
alias.consider(new Integer(3));       // 안전하지 않은
```

컴파일러는 이 예의 2행에 대해 "검사하지 않은" 경고를 발생시킬 것이다. 명시적인 인자의 변환 없이 "가공하지 않은"(제네릭이 아닌) `Arbiter`의 메소드 `consider`를 호출해왔기 때문이다. 이 경우에서 `alias`에게 `Integer`를 전달하면 안 된다는 경고를 명확하게 말해준다. 다른 예는 좀 더 미묘하다. 그 경고는 단지 **정적** 검사의 결여, 즉 실제로 발생하는 유형 오류는 어느 것이나 여전히 실행 시간에 잡아야 할 것임을 뜻할 뿐임을 역설한다.

예 8.71

자바 5 제네릭과 고유형

그런데 말소의 사용과 주어진 제네릭의 모든 인스턴스가 동일한 코드를 공유할 수 있어야 한다는 주장이 자바의 유형 인자가 모두 `Object`의 자식 클래스임을 의미함을 알자. `Arbiter<Integer>`가 완벽히 받아들일 수 있는 유형이지만 `Arbiter<int>`는 아니다.

설계와 구현

왜 말소인가?

자바의 말소는 몇 가지 놀라운 결과를 가진다. 첫째로 T가 유형 매개변수이면 new T()를 호출할 수 없다. 컴파일러는 어떤 종류의 객체를 만들지 알 수 없다. 비슷한 예로 프로그램이 실행 시간에 객체의 구체화된 유형에 대해 검사하고 추론하게 하는 자바의 반영 방법은 제네릭에 대해 어떤 것도 알지 못한다. 예를 들어 csNames.getClass().toString()은 "class Arbiter〈String〉"이 아닌 "class Arbiter"를 반환한다. 왜 자바 설계자는 그런 중요한 제한과 관련된 방법을 도입하지 않는 것인가? 답은 하위 호환성이나 좀 더 정확히 이전 코드와 새 코드의 완벽한 상호호환성을 필요로 하는 이주 호환성이다.

대부분의 정확한 언어보다 더 그렇게 자바는 무수히 다른 조직의 무수히 다른 사람이 독립적으로 작성한 구성 요소에서 동작하는 프로그램을 조립하는 것을 장려한다. 자바 설계자는 (제네릭이 아닌) 이전 프로그램을 새로운 (제네릭) 라이브러리를 이용해서도 실행할 수 있어야 하고 새로운 (제네릭) 프로그램을 이전 (제네릭이 아닌) 라이브러리를 이용해서도 실행할 수 있어야 한다고 느꼈다. 게다가 전형적인 구현에서 자바의 바이트 코드를 해석하는 자바 가상 기계를 변경할 수 없는 입장이었다. 이런 목표에 이의를 제기했지만 일단 그것을 받아들이자 말소는 자연스러운 해결책이 되었다.

C#의 제네릭

C# 버전 1에서 제네릭을 생략했었지만 언어 설계자는 항상 제네릭을 넣으려고 했었고 처음부터 적절한 지원을 제공하기 위해 .NET 공통 언어 기반 구조(CLI, Common Language Infrastructure)를 설계했다. 결과적으로 C# 2.0은 말소가 아닌 구체화(reification)에 기반한 구현을 채택할 수 있었다. 구체화는 제네릭을 다른 인자로 인스턴스화할 때마다 별개의 구체화된 유형을 생성한다. 구체화된 유형은 반영 라이브러리

(csNames.GetType().ToString()은 "Arbiter'1[System.Double]"을 반환한다)를 볼 수 있고 T가 인자가 없는 생성자를 가진 유형 매개변수면 new T()를 호출하는 것을 완벽히 수용할 수 있다(이 효과에 대한 제약 조건이 필요함). 게다가 자바 컴파일러가 (제네릭의 아무것도 알지 못하는) 가상 기계의 요구 사항을 만족시키기 위해서나 (매개변수를 전달하거나 부적절한 유형의 결과를 반환하는) 기존 코드와의 유형적으로 안전한 상호작용을 보증하기 위해 묵시적으로 형 변환을 생성시켜야 해야 하지만 C# 컴파일러는 그런 검사를 결코 필요로 하지 않을 것임을 확신할 수 있고 그래서 그것을 제거할 수 있다. 그 결과는 더 빠른 코드다.

예 8.72

C#에서 제네릭 구현의 공유

물론 C# 컴파일러는 동일한 코드를 가질 코드의 구현을 합치는 것에 자유롭다. 이런 공유는 C++보다 C#에서 훨씬 더 쉽다. 구현이 일반적으로 동일한 구현이 프로그램 내의 어디선가 이미 존재할지 아닌지를 명확할 때 실행하기 바로 직전까지 기계 코드의 생성을 지연시키는 적시(just-in-time) 컴파일을 채용하기 때문이다. 특히 C#이 클래스형의 변수에 대해 참조 모델을 채용하기 때문에 MyType〈Foo〉와 MyType〈Bar〉는 Foo와 Bar가 모두 클래스일 때마다 코드를 공유할 것이다.

예 8.73

C#의 제네릭과 고유형

C++와 마찬가지로 C#은 제네릭 인자가 단지 클래스가 아닌 값 유형(고유형이나 struct)이 될 수 있게 한다. 클래스 MyType<int>의 객체를 생성하는 데 문제가 없다. 즉, 그것을 자바에서 했던 것처럼 MyType<Integer>로 "래핑(wrap)"할 필요가 없다. MyType<int>와 MyType<double>은 일반적으로 코드를 공유하지 않을 것이다. 하지만 둘 모두 MyType<Integer>나 MyType<Double>보다 훨씬 더 빨리 실행될 것이다. 래퍼(wrapper) 객체를 생성하기 위해 필요한 동적 메모리 할당, 그것을 회수하기 위한 가배지 컬렉션이나 자료에 접근하기 위해 필요한 간접 오버헤드를 초래하지 않기 때문이다.

자바와 같이 C#은 제네릭 매개변수로 유형만을 허용하고 제네릭이 특정 인스턴스화와 독립적으로 명백히 유형적으로 안전하기를 주장한다. 그것은 대응하는 제네릭 매개변수의 제약을 만족시키지 않는 인자를 가진 제네릭을 인스턴스화하려 하거나 제네릭 내에서 제약 조건이 이용할 수 있을 것이라 보증하지 않는 메소드를 실행하려고 하면 적당한 오류 메시지를 발생시킨다.

예 8.74

C#의 제네릭 arbiter 클래스

앞선 Arbiter 클래스의 C# 버전은 ©(심화학습에 있는) 그림 8.14에서 확인할 수 있다. ©(심화학습에 있는) 그림 8.12와의 약간의 차이점 하나는 Arbiter 생성자에서 나타난다. 이는 명시적으로 항목 bestSoFar을 default(T)로 초기화해야 한다. 클래스 유형의 변수를 묵시적으로 null로 초기화하고 자바의 유형 매개변수가 모두 클래스이기 때문에 자바에서는 이를 뺄 수 있다. 하지만 C#에서 T는 고유형이나 struct가 될 수 있다. 이 둘 모두 명시적인 초기화를 필요로 한다.

©(심화학습에 있는) 그림 8.12와의 좀 더 흥미로운 차이점은 인터페이스가 아닌 대리인으로서 Chooser의 정의와 매개변수와 항목 선언에서 하한(super 키워드의 사용)의 결

여다. 이 쟁점은 서로 연관이 있다. C#은 유형 제약 조건으로 상한을 지정하게 한다. 예 8.38의 sort 루틴에서 그렇게 했다. 하지만 자바의 하한과 직접적인 동일한 것은 없다. Arbiter 예에서는 Chooser가 (©심화학습에 있는 그림 8.12에서 better라고 이름 붙인) 단 하나의 메소드를 가진다는 사실을 이용함으로써 문제를 회피할 수 있었다.

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

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

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

그림 8.14 | C#의 제네릭 arbiter

8.3.1절에서 언급했듯이 C#의 대리인은 제1종 서브루틴이다. ©(심화학습에 있는) 그림 8.14에서 대리인 선언은 C#의 Chooser 객체가 포인터가 아닌 클로저인 것을 제외하고 대략 아래 C의 선언(두 개의 T 인자를 받아서 논리형을 반환하는 함수로의 포인터)과 유사하다.

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

그것은 8.3.1절의 설계와 구현 보충설명에서 설명했듯이 정적 함수, 특정 객체의 메소드(이 경우 그 객체의 항목에 접근할 수 있음)나 익명의 중첩된 함수(주위 유효 범위에서 변수로의 접근이 제한 없이 가능함)를 참조할 수 있다. 특정 경우에서 Chooser를 대리인이 되게 정의하는 것은 클래스 상속 계층에 상관없이 Arbiter 생성자에게 적절한 함수를 전달하게 한다. 다음과 같이 선언할 수 있고

```
public static bool CaseSensitive(String a, String b) {
    return String.Compare(a, b) < 1;
}
public static bool CaseInsensitive(Object a, Object b) {
```

```
    return String.Compare(a.ToString(), b.ToString(), true) < 1;
}
```

다음과 같이 작성할 수 있다.

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

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

컴파일러가 `Chooser<String>`으로 `CaseInsensitive`를 인스턴스화하는 데 완전히 만족한다. 즉, `Strings`을 `Object`로 전달하기 때문이다.

✓ 확인문제

65. C++ 템플릿의 오용에 대한 고품질의 오류 메시지를 만들어내는 것이 어려운 이유는?

66. 템플릿 메타프로그래밍은 무엇인가?

67. 자바 유형 제약 조건에서 상한과 하한 간의 차이점을 설명하라. C#은 이 중 어떤 것을 지원하는가?

68. 유형 말소는 무엇인가? 자바에서 그것을 사용하는 이유는?

69. 자바 컴파일러는 어떤 상황에서 "검사하지 않은" 제네릭 경고를 발생하겠는가?

70. 두 가지 어떤 주요 이유 때문에 C# 제네릭은 C++ 템플릿보다 "코드 확장"을 더 거부하는가?

71. C#에서 제네릭 매개변수의 항목을 명시적으로 초기화해야 할 이유는 무엇인가?

72. C# 제네릭이 자바에서의 코드보다 좀 더 빠른 주요 이유 두 가지는?

73. C#의 대리인은 하나의 메소드를 가진 인터페이스(예를 들어 그림 8.11에 있는 C++의 `chooser`)와 어떻게 다른가? 그것은 C의 함수 포인터와 어떻게 다른가?