==========
Polymorphism and Generics

Recall from chapter 3:

ad hoc polymorphism: fancy name for overloading

subtype polymorphism in OO languages allows code to do the "right thing" when a ref of parent type refers to an object of child type
implemented with vtables (to be discussed in chapter 9)

parametric polymorphism
type is a parameter of the code, implicitly or explicitly

implicit (true)

language implementation figures out what code requires of object
at compile-time, as in ML or Haskell
at run-time, as in Lisp, Smalltalk, or Python
lets you apply operation to object only if object has
everything the code requires

explicit (***generics***)
programmer specifies the type parameters explcitly
mostly what I want to talk about today

Generics found in Clu, Ada, Modula-3, C++, Eiffel, Java 5, C# 2.0, ...

C++ calls its generics ***templates***.
Allow you, for example, to create a single stack abstraction, and instantiate it for stacks of integers, stacks of strings, stacks of employee records, ...

```
template <class T>
class stack {
   T[100] contents;
   int tos = 0;    // first unused location
public:
   T pop();
   void push(T);
   ...
}
...
stack<double> S;
```

I could, of course, do

```
class stack {
   void*[100] contents;
   int tos = 0;
public:
   void* pop();
   void push(void *);
   ...
}
```

But then I have to use type casts all over the place. Inconvenient and, in C++, unsafe.

Lots of containers (stacks, queues, sets, strings, mappings, ...) in the C++ standard library.

Similarly rich libraries exist for Java and C#.
(Also for Python and Ruby, but those use implicit parametric polymorphism with run-time checking.)

Some languages (e.g. Ada and C++) allow things other than types to be passed as template arguments:

```
template <class T, int N>
class stack {
   T[N] contents;
   int tos = 0;
public:
   T pop();
   void push(T);
   ...
}
...
stack<double, 100> S;
```

Implementation

C# generics do run-time instantiation (***reification***). When you say `stack<foo>`, the run-time system invokes the JIT compiler and generates the appropriate code. Doesn't box native types if it doesn't need to -- more efficient.

Java doesn't do run-time instantiation. Internally everything is `stack<Object>`. You avoid the casts in the source code, but you have to pay for boxing of native types. And since the designers were unwilling (for backward compatibility reasons) to modify the VM, you're stuck with the casts in the generated code (automatically inserted by the compiler) -- even though the compiler knows they're going to succeed -- because the JVM won't accept the byte code otherwise: it will think it's unsafe. Also, because everything is an `Object` internally, reflection doesn't work.

The Java implementation strategy is known as ***erasure*** -- the type parameters are simply erased by the compiler. One more disadvantage: you can't say `new T()`, where T is generic parameter, because Java doesn't know what to create.

C++ does compile-time instantiation; more below.
C# may do compile-time instantiation when it can, as an optimization.

Constraints

The problem:

If I'm writing a sorting routine, how do I insist that the elements in the to-be-sorted list support a `less_than()` method or < operator?

If I'm writing a hash table, how do I insist that the keys support a `hash()` method?

If I'm writing output formatting code, how do I insist that objects support a `to_string()` method?

Related question:

Do I (can I) type-check the generic code, independent of any particular instantiation, or do I type-check the instantiations independently?

Tradeoffs are nicely illustrated by comparing Java, C#, and C++.

C++ is very flexible: every instantiation is independently type-checked. Constraints are implicit: if we try to instantiate a template for a type that doesn't support needed operations, the instance won't type-check. This has led, historically, to ***really*** messy error messages.

Most other languages type-check the generic itself, so you don't get any instantiation-specific error messages. To support this, they require that the operations supported by generic parameter types be explicitly listed. Java and C# leverage interfaces for this purpose.
C++20 adds ***concepts***, which provide an (optional, for backward compatibility) superset of the functionality found in Java and C#.

Java example:

```
public static <T implements Comparable<T>> void sort(T A[]) {
   ...
   if (A[i].compareTo(A[j]) >= 0) ...
   ...
}
...
Integer[] myArray = new Integer[50];
...
sort(myArray);
```

Note that Java puts the type parameters right in front of the return type of the function, rather than in a preceding "template" clause.
`Comparable` is a standard library interface that includes the `compareTo()` method. Wrapper class `Integer` implements `Comparable<Integer>`.

C# syntax is similar:

```
static void sort<T>(T[] A) where T : IComparable {
   ...
   if (A[i].compareTo(A[j]) >= 0) ...
   ...
}
...
int[] myArray = new int[50];
...
sort(myArray);
```

C# puts the type parameter between the function name and the parameter list and the constraints after the parameter list. Java won't let you use `int` as a generic parameter, but C# is happy to; it creates a custom version of sort for `int`s.

(pre-20) C++ doesn't require that constraints be explicit.

```
template <class T>
void sort(T A[], int A_size) {...
```

(C++ can't figure out the size of an array, so you have to pass it in. Alternatively you could make it another generic parameter.)

As noted in the book, bad things happen if a parameter "accidentally" supports a needed operation, in the "wrong way". If we instantiate sort on an array of C strings (`char*`s), for example, we get sorting by location in memory, not lexicographic order (C++ string objects compare lexicographically).

Constraints have historically been specified explicitly in C++ only by convention:
make a function parameter inherit from a standard base class
e.g., `sort<foo>(SortableVector<foo> A)`, where `SortableVector<T>` inherits from both `Vector<T>` and `Comparator<T>`
provide required operations as generic parameters
e.g. `sort<foo, comparator<foo>>(foo* A, int len)`
provide required operations as ordinary parameters
e.g. `sort<foo>(foo* A, int len, bool (*less_than<foo>)())`
make sort the `operator()` of a class for which `less_than()` is a constructor argument
e.g. `sort = new Sorter(bool (*less_than<foo>)())`,
where `Sorter` has an `operator()`

Concepts change this:

```
template <typename T>
concept Comparable = requires(T a, T b) { a < b; };

template <Comparable T>
void sort(T A[], int A_size) {...
```

Implicit Instantiation

Several languages, including C++, Java, and C#, will instantiate generic functions (not classes) as you need them, using roughly the same resolution mechanism used for overloading. (Actually, in C++ it requires unification, because of the generality of generic parameters, including nested templates and specialization.)

Interaction with Subtype Polymorphism

These two play nicely together. If I derive queue from `list` I want subclasses. But I may also want generics: derive queue<T> from `list<T>`.

The subtle part is ***conformance*** of argument and return types.
Suppose I want to be able to sort things in Java that don't implement `Comparable` themselves. I could make the comparator be a constructor argument instead of a generic argument (the 4th by-convention option in C++ above):

```
interface Comparator<T> {
   public Boolean ordered(T a, T b);
}

class Sorter<T> {
   Comparator<T> comp;

   public Sorter(Comparator<T> c) {  // constructor
      comp = c;
   }

   public void sort(T A[]) {
      ...
      if (comp.ordered(A[i], A[j])) ...
      ...
   }
}

class IntComp implements Comparator<Integer> {
   public Boolean ordered(Integer a, Integer b) {
      return a < b;
   }
}

Sorter<Integer> s = new Sorter<Integer>(new IntComp());
s.sort(myArray);
```

This works fine, but it breaks if I try

```
class ObjComp implements Comparator<Object> {
   public Boolean ordered(Object a, Object b) {
      return a.toString().compareTo(b.toString()) < 0;
   }
}

Sorter<Integer> s = new Sorter<Integer>(new ObjComp());
s.sort(myArray);
```

The call to new causes the compiler to generate a type clash message, because we're passing a `Comparator<Object>` rather than a `Comparator<Integer>`. This is fixed in Java using ***type wildcards***:

```
class Sorter<T> {
   Comparator<? super T> comp;

   public Sorter(Comparator<? super T> c) {
      comp = c;
   }
```

In general, you use `<? super T>` when you expect to pass objects into a method that might be willing to take something more general (and you never expect to accept such objects in return).

There's also `<? extends T>` syntax, for when you expect something back out of a context that might in fact give you something more specific (and you never expect to pass such objects in).

In effect, these `super` and `extends` keywords serve to control type compatibility for generics. Given a generic `Foo<T>`, we must ask: if C is derived from P (and thus C can be used in any context that expects a P), can `Foo<C>` be used in any context that expects `Foo<P>`?
If so, we say `Foo<T>` is ***covariant*** in T.
Typically happens in the case where T objects are returned from Foo methods, but never passed into them as parameters.
For example, I can probably pass a `Generator<C>` object to anybody who expects a `Generator<P>` object:
- They expect to use the generator to conjure up P objects.
- If the generator gives them a C instead, they're happy.

Conversely (and more commonly), there are times when a `Foo<P>` object can be used in a context that expects `Foo<C>`.
When this happens, we say `Foo<T>` is ***contravariant*** in T.
Typically happens when T objects are passed to `Foo` methods, but never returned from them.
For example, I can probably pass a `Printer<P>` object to anybody who expects a `Printer<C>`.
- They're only going to give it C objects.
- Since the printer is willing to take a P object, it's happy.

C# makes covariance and contravariance a property of the generic itself. You specify `<in T>` and `<out T>` in the declaration of the generic to indicate contravariance and covariance, respectively. This restricts what can be done with T inside the generic.

Java gives you more flexibility (and arguably more confusion) by allowing you to annotate ***uses*** of the generic -- in effect saying "I promise (and the compiler should verify) that I never call methods that pass objects the wrong way).
If you're always going to pass objects in, you're using contravariance, and you say `<? super T>`
If you're always going to accept objects as return values, you're using covariance, and you say `<? extends T>`.

covariance:
C is a P --> `Foo<C>` is a `Foo<P>` `C# out` `Java extends`
contravariance:
C is a P --> `Foo<P>` is a `Foo<C>` `C# in` `Java super`
invariance:
C is a P but `Foo<C>` and `Foo<P>` are incomparable

More on this on the PLP companion, section 7.3.2.
The Wikipedia page on covariance & contravariance has even more detail.