More Generics

How do I really use wildcards?

Consider

```
public class Stack<E> {
public Stack();
public void push(E e);
public E pop();
public boolean isEmpty(); }
```

Suppose we want to add a method that takes a sequence of elements and pushes them all onto the stack.

First Try ….

```
public void pushAll(Iterable<E> src) {
for (E e : src)
push(e);
}
```

- This method compiles cleanly, but it isn't entirely satisfactory.
- If the element type (E) of the Iterable exactly matches that of the stack, it works fine.

But suppose you have a Stack<Number> and you invoke push(intVal), where intVal is of type Integer.

This works, because Integer is a subtype of Number.

So logically, it seems that this should work, too:

```
Stack<Number> numberStack = new Stack<Number>();
Iterable<Integer> integers = new …..;
numberStack.pushAll(integers);
```

If you try it, however, you'll get this error message because parameterized types are invariant:

StackTest.java: pushAll(Iterable<Number>) in Stack<Number>cannot be applied to (Iterable<Integer>)
numberStack.pushAll(integers);

The type of the input parameter to pushAll should not be "Iterable of E" but "Iterable of some subtype of E,"

and there is a wildcard type that means precisely that:

Iterable<?extends E>

(*subtype* is defined so that every type is a subtype of itself, even though it does not extend itself.)

Let’s modify pushAll to use this type:

```
public void pushAll(Iterable<? extends E> src) {
for (E e : src)
push(e);
}
```

With this change, not only does Stack compile cleanly, but so does the client code that wouldn’t compile with the original pushAll declaration.

Because Stack and its client compile cleanly, you know that everything is typesafe!

Now suppose you want to write a popAll method to go with pushAll

- The popAll method pops each element off the stack and adds the elements to the given collection.

- Here's how a first attempt at writing the popAll method might look:…

```
public void popAll(Collection<E> dst) {
while (!isEmpty())
dst.add(pop());
}
```

Again, this compiles cleanly and works fine if the element type of the destination collection exactly matches that of the stack.

But …. Suppose you have a Stack<Number> and variable of type Object.

If you pop an element from the stack and store it in the variable, it compiles and runs without error.

So shouldn't you be able to do this, too?

```
Stack<Number> numberStack = new Stack<Number>();
Collection<Object> objects =  new ….;
numberStack.popAll(objects);
```

If you try to compile this client code against the version of popAll above, you'll get an error very similar to the one that we got with our first version of pushAll:

Collection<Object> is not a subtype of Collection<Number>.

Once again, wildcard types provide a way out.

The type of the input parameter to popAll should not be “collection of E” but “collection of some supertype of E”

(where supertype is defined such that E is a supertype of itself).

Again, there is a wildcard type that means precisely that:

Collection<? super E>

Let’s modify popAll to use it:

```
public void popAll(Collection<? super E> dst) {
while (!isEmpty())
dst.add(pop());
}
```

With this change, both Stack and the client code compile cleanly.

In summary

If an input parameter is both a producer and a consumer, then wildcard types will do you no good:

You need an exact type match, which is what you get without any wildcards.

In summary

- if a parameterized type represents a T producer, use <? extends T>

- In our Stack example, pushAll's src parameter produces E instances for use by the Stack, so the appropriate type for src is

Iterable<? extends E>

(Covariant)

In summary

if it represents a T consumer, use <? super T>

popAll's dst parameter consumes E instances from the Stack, so the appropriate type for dst is

Collection<? super E>

(Contravariant)