Generics in the Java Programming Language

Gilad Bracha

July 5, 2004

Contents

1 Introduction

JDK 1.5 introduces several extensions to the Java programming language. One of these is the introduction of *generics*.

This tutorial is aimed at introducing you to generics. You may be familiar with similar constructs from other languages, most notably C++ templates. If so, you'll soon see that there are both similarities and important differences. If you are not familiar with look-a-alike constructs from elsewhere, all the better; you can start afresh, without unlearning any misconceptions.

Generics allow you to abstract over types. The most common examples are container types, such as those in the Collection hierarchy.

Here is a typical usage of that sort:

```
List myIntList = new LinkedList(); // 1
myIntList.add(new Integer(0)); // 2
Integer x = (Integer) myIntList.iterator().next(); // 3
```

The cast on line 3 is slightly annoying. Typically, the programmer knows what kind of data has been placed into a particular list. However, the cast is essential. The compiler can only guarantee that an Object will be returned by the iterator. To ensure the assignment to a variable of type Integer is type safe, the cast is required.

Of course, the cast not only introduces clutter. It also introduces the possibility of a run time error, since the programmer might be mistaken.

What if programmers could actually express their intent, and mark a list as being restricted to contain a particular data type? This is the core idea behind generics. Here is a version of the program fragment given above using generics:

```
List<Integer> myIntList = new LinkedList<Integer>(); // 1'
myIntList.add(new Integer(0)); //2'
Integer x = myIntList.iterator().next(); // 3'
```

Notice the type declaration for the variable myIntList. It specifies that this is not just an arbitrary List, but a List of Integer, written List<Integer>. We say that List is a generic interface that takes a *type parameter* - in this case, Integer. We also specify a type parameter when creating the list object.

The other thing to pay attention to is that the cast is gone on line 3'.

Now, you might think that all we've accomplished is to move the clutter around. Instead of a cast to Integer on line 3, we have Integer as a type parameter on line 1'. However, there is a very big difference here. The compiler can now check the type correctness of the program at compile-time. When we say that myIntList is declared with type List<Integer>, this tells us something about the variable myIntList, which holds true wherever and whenever it is used, and the compiler will guarantee it. In contrast, the cast tells us something the programmer thinks is true at a single point in the code.

The net effect, especially in large programs, is improved readability and robustness.

2 Defining Simple Generics

Here is a small excerpt from the definitions of the interfaces List and Iterator in package java.util:

```
public interface List<E> {
  void add(E x);
  Iterator<E> iterator();
}
public interface Iterator<E> {
  E next();
  boolean hasNext();
}
```

This should all be familiar, except for the stuff in angle brackets. Those are the declarations of the *formal type parameters* of the interfaces List and Iterator.

Type parameters can be used throughout the generic declaration, pretty much where you would use ordinary types (though there are some important restrictions; see section 7).

In the introduction, we saw *invocations* of the generic type declaration List, such as List<Integer>. In the invocation (usually called a *parameterized type*), all occurrences of the formal type parameter (E in this case) are replaced by the *actual type argument* (in this case, Integer).

You might imagine that List<Integer> stands for a version of List where E has been uniformly replaced by Integer:

```
public interface IntegerList {
  void add(Integer x)
  Iterator<Integer> iterator();
}
```

This intuition can be helpful, but it's also misleading.

It is helpful, because the parameterized type List<Integer> does indeed have methods that look just like this expansion.

It is misleading, because the declaration of a generic is never actually expanded in this way. There aren't multiple copies of the code: not in source, not in binary, not on disk and not in memory. If you are a C++ programmer, you'll understand that this is very different than a C++ template.

A generic type declaration is compiled once and for all, and turned into a single class file, just like an ordinary class or interface declaration.

Type parameters are analogous to the ordinary parameters used in methods or constructors. Much like a method has *formal value parameters* that describe the kinds of values it operates on, a generic declaration has formal type parameters. When a method is invoked, *actual arguments* are substituted for the formal parameters, and the method body is evaluated. When a generic declaration is invoked, the actual type arguments are substituted for the formal type parameters.

A note on naming conventions. We recommend that you use pithy (single character if possible) yet evocative names for formal type parameters. It's best to avoid lower

case characters in those names, making it easy to distinguish formal type parameters from ordinary classes and interfaces. Many container types use E, for element, as in the examples above. We'll see some additional conventions in later examples.

3 Generics and Subtyping

Let's test our understanding of generics. Is the following code snippet legal?

```
List<String> ls = new ArrayList<String>(); //1
List<Object> lo = ls; //2
```

Line 1 is certainly legal. The trickier part of the question is line 2. This boils down to the question: is a List of String a List of Object. Most people's instinct is to answer: "sure!".

Well, take a look at the next few lines:

```
lo.add(new Object()); // 3
String s = ls.get(0); // 4: attempts to assign an Object to a String!
```

Here we've aliased ls and lo. Accessing ls, a list of String, through the alias lo, we can insert arbitrary objects into it. As a result ls does not hold just Strings anymore, and when we try and get something out of it, we get a rude surprise.

The Java compiler will prevent this from happening of course. Line 2 will cause a compile time error.

In general, if Foo is a subtype (subclass or subinterface) of Bar, and G is some generic type declaration, it is **not** the case that G<Foo> is a subtype of G<Bar>. This is probably the hardest thing you need to learn about generics, because it goes against our deeply held intuitions.

The problem with that intuition is that it assumes that collections don't change. Our instinct takes these things to be immutable.

For example, if the department of motor vehicles supplies a list of drivers to the census bureau, this seems reasonable. We think that a List<Driver> is a List<Person>, assuming that Driver is a subtype of Person. In fact, what is being passed is a **copy** of the registry of drivers. Otherwise, the census bureau could add new people who are not drivers into the list, corrupting the DMV's records.

In order to cope with this sort of situation, it's useful to consider more flexible generic types. The rules we've seen so far are quite restrictive.

4 Wildcards

Consider the problem of writing a routine that prints out all the elements in a collection. Here's how you might write it in an older version of the language:

```
void printCollection(Collection c) {
  Iterator i = c.iterator();
  for (k = 0; k < c.size(); k++) {
    System.out.println(i.next());
}}
```

And here is a naive attempt at writing it using generics (and the new for loop syntax):

```
void printCollection(Collection<Object> c) {
  for (Object e : c) {
    System.out.println(e);
}}
```

The problem is that this new version is much less useful than the old one. Whereas the old code could be called with any kind of collection as a parameter, the new code only takes Collection<Object>, which, as we've just demonstrated, is *not* a supertype of all kinds of collections!

So what *is* the supertype of all kinds of collections? It's written Collection<?> (pronounced "collection of unknown") , that is, a collection whose element type matches anything. It's called a *wildcard type* for obvious reasons. We can write:

```
void printCollection(Collection<?> c) {
  for (Object e : c) {
    System.out.println(e);
}}
```

and now, we can call it with any type of collection. Notice that inside printCollection(), we can still read elements from c and give them type Object. This is always safe, since whatever the actual type of the collection, it does contain objects. It isn't safe to add arbitrary objects to it however:

```
Collection<?> c = new ArrayList<String>();
c.add(new Object()); // compile time error
```

Since we don't know what the element type of c stands for, we cannot add objects to it. The add() method takes arguments of type E, the element type of the collection. When the actual type parameter is ?, it stands for some unknown type. Any parameter we pass to add would have to be a subtype of this unknown type. Since we don't know what type that is, we cannot pass anything in. The sole exception is null, which is a member of every type.

On the other hand, given a List<?>, we **can** call get() and make use of the result. The result type is an unknown type, but we always know that it is an object. It is

therefore safe to assign the result of get() to a variable of type Object or pass it as a parameter where the type Object is expected.

4.1 Bounded Wildcards

Consider a simple drawing application that can draw shapes such as rectangles and circles. To represent these shapes within the program, you could define a class hierarchy such as this:

```
public abstract class Shape {
  public abstract void draw(Canvas c);
}
public class Circle extends Shape {
  private int x, y, radius;
  public void draw(Canvas c) { ... }
}
public class Rectangle extends Shape {
  private int x, y, width, height;
  public void draw(Canvas c) { ... }
}
```

These classes can be drawn on a canvas:

```
public class Canvas {
  public void draw(Shape s) {
    s.draw(this);
  }
}
```

Any drawing will typically contain a number of shapes. Assuming that they are represented as a list, it would be convenient to have a method in Canvas that draws them all:

```
public void drawAll(List<Shape> shapes) {
  for (Shape s: shapes) {
    s.draw(this);
  }
}
```

Now, the type rules say that drawAll() can only be called on lists of exactly Shape: it cannot, for instance, be called on a List<Circle>. That is unfortunate, since all the method does is read shapes from the list, so it could just as well be called on a List<Circle>. What we really want is for the method to accept a list of *any* kind of shape:

```
public void drawAll(List<? extends Shape> shapes) { ... }
```

There is a small but very important difference here: we have replaced the type List<Shape> with List<? extends Shape>. Now drawAll() will accept lists of any subclass of Shape, so we can now call it on a List<Circle> if we want.

List<? **extends** Shape> is an example of a *bounded wildcard.* The ? stands for an unknown type, just like the wildcards we saw earlier. However, in this case, we know that this unknown type is in fact a subtype of Shape[1]. We say that Shape is the *upper bound* of the wildcard.

There is, as usual, a price to be paid for the flexibility of using wildcards. That price is that it is now illegal to write into shapes in the body of the method. For instance, this is not allowed:

```
public void addRectangle(List<? extends Shape> shapes) {
   shapes.add(0, new Rectangle()); // compile-time error!
}
```

You should be able to figure out why the code above is disallowed. The type of the second parameter to shapes.add() is ? **extends** Shape - an unknown subtype of Shape. Since we don't know what type it is, we don't know if it is a supertype of Rectangle; it might or might not be such a supertype, so it isn't safe to pass a Rectangle there.

Bounded wildcards are just what one needs to handle the example of the DMV passing its data to the census bureau. Our example assumes that the data is represented by mapping from names (represented as strings) to people (represented by reference types such as Person or its subtypes, such as Driver). Map<K,V> is an example of a generic type that takes two type arguments, representing the keys and values of the map.

Again, note the naming convention for formal type parameters - K for keys and V for values.

```
public class Census {
  public static void
     addRegistry(Map<String, ? extends Person> registry) { ...}
}
...
Map<String, Driver> allDrivers = ...;
Census.addRegistry(allDrivers);
```

5 Generic Methods

Consider writing a method that takes an array of objects and a collection and puts all objects in the array into the collection.

Here is a first attempt:

```
static void fromArrayToCollection(Object[] a, Collection<?> c) {
  for (Object o : a) {
    c.add(o); // compile time error
}}
```

By now, you will have learned to avoid the beginner's mistake of trying to use Collection<Object> as the type of the collection parameter. You may or may not

[1] It could be Shape itself, or some subclass; it need not literally extend Shape.

have recognized that using Collection<?> isn't going to work either. Recall that you cannot just shove objects into a collection of unknown type.

The way to do deal with these problems is to use *generic methods.* Just like type declarations, method declarations can be generic - that is, parameterized by one or more type parameters.

```
static <T> void fromArrayToCollection(T[] a, Collection<T> c) {
  for (T o : a) {
    c.add(o); // correct
}}
```

We can call this method with any kind of collection whose element type is a supertype of the element type of the array.

```
Object[] oa = new Object[100];
Collection<Object> co = new ArrayList<Object>();
fromArrayToCollection(oa, co);// T inferred to be Object
String[] sa = new String[100];
Collection<String> cs = new ArrayList<String>();
fromArrayToCollection(sa, cs);// T inferred to be String
fromArrayToCollection(sa, co);// T inferred to be Object
Integer[] ia = new Integer[100];
Float[] fa = new Float[100];
Number[] na = new Number[100];
Collection<Number> cn = new ArrayList<Number>();
fromArrayToCollection(ia, cn);// T inferred to be Number
fromArrayToCollection(fa, cn);// T inferred to be Number
fromArrayToCollection(na, cn);// T inferred to be Number
fromArrayToCollection(na, co);// T inferred to be Object
fromArrayToCollection(na, cs);// compile-time error
```

Notice that we don't have to pass an actual type argument to a generic method. The compiler infers the type argument for us, based on the types of the actual arguments. It will generally infer the most specific type argument that will make the call type-correct.

One question that arises is: when should I use generic methods, and when should I use wildcard types? To understand the answer, let's examine a few methods from the Collection libraries.

```
interface Collection<E> {
  public boolean containsAll(Collection<?> c);
  public boolean addAll(Collection<? extends E> c);
}
```

We could have used generic methods here instead:

```
interface Collection<E> {
  public <T> boolean containsAll(Collection<T> c);
  public <T extends E> boolean addAll(Collection<T> c);
  // hey, type variables can have bounds too!
}
```

However, in both containsAll and addAll, the type parameter T is used only once. The return type doesn't depend on the type parameter, nor does any other argument to the method (in this case, there simply is only one argument). This tells us that the type argument is being used for polymorphism; its only effect is to allow a variety of actual argument types to be used at different invocation sites. If that is the case, one should use wildcards. Wildcards are designed to support flexible subtyping, which is what we're trying to express here.

Generic methods allow type parameters to be used to express dependencies among the types of one or more arguments to a method and/or its return type. If there isn't such a dependency, a generic method should not be used.

It is possible to use both generic methods and wildcards in tandem. Here is the method Collections.copy():

```
class Collections {
  public static  <T> void copy(List<T> dest, List<? extends T> src){...}
}
```

Note the dependency between the types of the two parameters. Any object copied from the source list, src, must be assignable to the element type T of the destination list, dst. So the element type of src can be any subtype of T - we don't care which. The signature of copy expresses the dependency using a type parameter, but uses a wildcard for the element type of the second parameter.

We could have written the signature for this method another way, without using wildcards at all:

```
class Collections {
  public static  <T, S extends T>
        void copy(List<T> dest, List<S> src){...}
}
```

This is fine, but while the first type parameter is used both in the type of dst and in the bound of the second type parameter, S, S itself is only used once, in the type of src - nothing else depends on it. This is a sign that we can replace S with a wildcard. Using wildcards is clearer and more concise than declaring explicit type parameters, and should therefore be preferred whenever possible.

Wildcards also have the advantage that they can be used outside of method signatures, as the types of fields, local variables and arrays. Here is an example.

Returning to our shape drawing problem, suppose we want to keep a history of drawing requests. We can maintain the history in a static variable inside class Shape, and have drawAll() store its incoming argument into the history field.

```
static List<List<? extends Shape>> history =
new ArrayList<List<? extends Shape>>();
public void drawAll(List<? extends Shape> shapes) {
   history.addLast(shapes);
   for (Shape s: shapes) {
      s.draw(this);
   }}
```

Finally, again let's take note of the naming convention used for the type parameters. We use T for type, whenever there isn't anything more specific about the type to distinguish it. This is often the case in generic methods. If there are multiple type parameters, we might use letters that neighbor T in the alphabet, such as S. If a generic method appears inside a generic class, it's a good idea to avoid using the same names for the type parameters of the method and class, to avoid confusion. The same applies to nested generic classes.

6 Interoperating with Legacy Code

Until now, all our examples have assumed an idealized world, where everyone is using the latest version of the Java programming language, which supports generics.

Alas, in reality this isn't the case. Millions of lines of code have been written in earlier versions of the language, and they won't all be converted overnight.

Later, in section 10, we will tackle the problem of converting your old code to use generics. In this section, we'll focus on a simpler problem: how can legacy code and generic code interoperate? This question has two parts: using legacy code from within generic code, and using generic code within legacy code.

6.1 Using Legacy Code in Generic Code

How can you use old code, while still enjoying the benefits of generics in your own code?

As an example, assume you want to use the package com.Fooblibar.widgets. The folks at Fooblibar.com [2] market a system for inventory control, highlights of which are shown below:

```
package com.Fooblibar.widgets;
public interface Part { ...}
public class Inventory {
/**
 * Adds a new Assembly to the inventory database.
 * The assembly is given the name name, and consists of a set
 * parts specified by parts. All elements of the collection parts
 * must support the Part interface.
**/
  public static void addAssembly(String name, Collection parts) {...}
  public static Assembly getAssembly(String name) {...}
}
public interface Assembly {
  Collection getParts(); // Returns a collection of Parts
}
```

Now, you'd like to add new code that uses the API above. It would be nice to ensure that you always called addAssembly() with the proper arguments - that is, that

[2] Fooblibar.com is a purely fictional company, used for illustration purposes. Any relation to any real company or institution, or any persons living or dead, is purely coincidental.

the collection you pass in is indeed a Collection of Part. Of course, generics are tailor made for this:

```
package com.mycompany.inventory;
import com.Fooblibar.widgets.*;
public class Blade implements Part {
...
}
public class Guillotine implements Part {
}
public class Main {
  public static void main(String[] args) {
    Collection<Part> c = new ArrayList<Part>();
    c.add(new Guillotine()) ;
    c.add(new Blade());
    Inventory.addAssembly("thingee", c);
    Collection<Part> k = Inventory.getAssembly("thingee").getParts();
}}
```

When we call addAssembly, it expects the second parameter to be of type Collection. The actual argument is of type Collection<Part>. This works, but why? After all, most collections don't contain Part objects, and so in general, the compiler has no way of knowing what kind of collection the type Collection refers to.

In proper generic code, Collection would always be accompanied by a type parameter. When a generic type like Collection is used without a type parameter, it's called a *raw type.*

Most people's first instinct is that Collection really means Collection<Object>. However, as we saw earlier, it isn't safe to pass a Collection<Part> in a place where a Collection<Object> is required. It's more accurate to say that the type Collection denotes a collection of some unknown type, just like Collection<?>.

But wait, that can't be right either! Consider the call to getParts(), which returns a Collection. This is then assigned to k, which is a Collection<Part>. If the result of the call is a Collection<?>, the assignment would be an error.

In reality, the assignment is legal, but it generates an *unchecked warning*. The warning is needed, because the fact is that the compiler can't guarantee its correctness. We have no way of checking the legacy code in getAssembly() to ensure that indeed the collection being returned is a collection of Parts. The type used in the code is Collection, and one could legally insert all kinds of objects into such a collection.

So, shouldn't this be an error? Theoretically speaking, yes; but practically speaking, if generic code is going to call legacy code, this has to be allowed. It's up to you, the programmer, to satisfy yourself that in this case, the assignment is safe because the contract of getAssembly() says it returns a collection of Parts, even though the type signature doesn't show this.

So raw types are very much like wildcard types, but they are not typechecked as stringently. This is a deliberate design decision, to allow generics to interoperate with pre-existing legacy code.

Calling legacy code from generic code is inherently dangerous; once you mix generic code with non-generic legacy code, all the safety guarantees that the generic

type system usually provides are void. However, you are still better off than you were without using generics at all. At least you know the code on your end is consistent.

At the moment there's a lot more non-generic code out there then there is generic code, and there will inevitably be situations where they have to mix.

If you find that you must intermix legacy and generic code, pay close attention to the unchecked warnings. Think carefully how you can justify the safety of the code that gives rise to the warning.

What happens if you still made a mistake, and the code that caused a warning is indeed not type safe? Let's take a look at such a situation. In the process, we'll get some insight into the workings of the compiler.

6.2 Erasure and Translation

```
public String loophole(Integer x) {
    List<String> ys = new LinkedList<String>();
    List xs = ys;
    xs.add(x); // compile-time unchecked warning
    return ys.iterator().next();
}
```

Here, we've aliased a list of strings and a plain old list. We insert an Integer into the list, and attempt to extract a String. This is clearly wrong. If we ignore the warning and try to execute this code, it will fail exactly at the point where we try to use the wrong type. At run time, this code behaves like:

```
public String loophole(Integer x) {
  List ys = new LinkedList;
  List xs = ys;
  xs.add(x);
  return (String) ys.iterator().next(); // run time error
}
```

When we extract an element from the list, and attempt to treat it as a string by casting it to String, we will get a ClassCastException. The exact same thing happens with the generic version of loophole().

The reason for this is, that generics are implemented by the Java compiler as a front-end conversion called *erasure*. You can (almost) think of it as a source-to-source translation, whereby the generic version of loophole() is converted to the non-generic version.

As a result, **the type safety and integrity of the Java virtual machine are never at risk, even in the presence of unchecked warnings**.

Basically, erasure gets rid of (or *erases*) all generic type information. All the type information betweeen angle brackets is thrown out, so, for example, a parameterized type like List<String> is converted into List. All remaining uses of type variables are replaced by the upper bound of the type variable (usually Object). And, whenever the resulting code isn't type-correct, a cast to the appropriate type is inserted, as in the last line of loophole.

The full details of erasure are beyond the scope of this tutorial, but the simple description we just gave isn't far from the truth. It's good to know a bit about this, especially if you want to do more sophisticated things like converting existing APIs to use generics (see section 10), or just want to understand why things are the way they are.

6.3 Using Generic Code in Legacy Code

Now let's consider the inverse case. Imagine that Fooblibar.com chose to convert their API to use generics, but that some of their clients haven't yet. So now the code looks like:

```
package com.Fooblibar.widgets;
public interface Part { ...}
public class Inventory {
/**
 * Adds a new Assembly to the inventory database.
 * The assembly is given the name name, and consists of a set
 * parts specified by parts. All elements of the collection parts
 * must support the Part interface.
**/
 public static void addAssembly(String name, Collection<Part> parts) {...}
 public static Assembly getAssembly(String name) {...}
}
public interface Assembly {
 Collection<Part> getParts(); // Returns a collection of Parts
}
```

and the client code looks like:

```
package com.mycompany.inventory;
import com.Fooblibar.widgets.*;
public class Blade implements Part {
...
}
public class Guillotine implements Part {
}
public class Main {
 public static void main(String[] args) {
   Collection c = new ArrayList();
   c.add(new Guillotine()) ;
   c.add(new Blade());
   Inventory.addAssembly("thingee", c); // 1: unchecked warning
   Collection k = Inventory.getAssembly("thingee").getParts();
}}
```

The client code was written before generics were introduced, but it uses the package com.Fooblibar.widgets and the collection library, both of which are using generic types. All the uses of generic type declarations in the client code are raw types.

Line 1 generates an unchecked warning, because a raw Collection is being passed in where a Collection of Parts is expected, and the compiler cannot ensure that the raw Collection really is a Collection of Parts.

As an alternative, you can compile the client code using the source 1.4 flag, ensuring that no warnings are generated. However, in that case you won't be able to use any of the new language features introduced in JDK 1.5.

7 The Fine Print

7.1 A Generic Class is Shared by all its Invocations

What does the following code fragment print?

```
List <String> l1 = new ArrayList<String>();
List<Integer> l2 = new ArrayList<Integer>();
System.out.println(l1.getClass() == l2.getClass());
```

You might be tempted to say false, but you'd be wrong. It prints true, because all instances of a generic class have the same run-time class, regardless of their actual type parameters.

Indeed, what makes a class generic is the fact that it has the same behavior for all of its possible type parameters; the same class can be viewed as having many different types.

As consequence, the static variables and methods of a class are also shared among all the instances. That is why it is illegal to refer to the type parameters of a type declaration in a static method or initializer, or in the declaration or initializer of a static variable.

7.2 Casts and InstanceOf

Another implication of the fact that a generic class is shared among all its instances, is that it usually makes no sense to ask an instance if it is an instance of a particular invocation of a generic type:

```
Collection cs = new ArrayList<String>();
if (cs instanceof Collection<String>) { ...} // illegal
```

similarly, a cast such as

```
Collection<String> cstr = (Collection<String>) cs; // unchecked warning
```

gives an unchecked warning, since this isn't something the run time system is going to check for you.

The same is true of type variables

```
<T> T badCast(T t, Object o) {return (T) o; // unchecked warning
    }
```

Type variables don't exist at run time. This means that they entail no performance overhead in either time nor space, which is nice. Unfortunately, it also means that you can't reliably use them in casts.

7.3 Arrays

The component type of an array object may not be a type variable or a parameterized type, unless it is an (unbounded) wildcard type.You can declare array *types* whose element type is a type variable or a parameterized type, but not array *objects*.

This is annoying, to be sure. This restriction is necessary to avoid situations like:

```
List<String>[] lsa = new List<String>[10]; // not really allowed
Object o = lsa;
Object[] oa = (Object[]) o;
List<Integer> li = new ArrayList<Integer>();
li.add(new Integer(3));
oa[1] = li; // unsound, but passes run time store check
String s = lsa[1].get(0); // run-time error - ClassCastException
```

If arrays of parameterized type were allowed, the example above would compile without any unchecked warnings, and yet fail at run-time. We've had type-safety as a primary design goal of generics. In particular, the language is designed to guarantee that **if your entire application has been compiled without unchecked warnings using javac -source 1.5, it is type safe**.

However, you can still use wildcard arrays. Here are two variations on the code above. The first forgoes the use of both array objects and array types whose element type is parameterized. As a result, we have to cast explicitly to get a String out of the array.

```
List<?>[] lsa = new List<?>[10]; // ok, array of unbounded wildcard type
Object o = lsa;
Object[] oa = (Object[]) o;
List<Integer> li = new ArrayList<Integer>();
li.add(new Integer(3));
oa[1] = li; // correct
String s = (String) lsa[1].get(0); // run time error, but cast is explicit
```

IIn the next variation, we refrain from creating an array object whose element type is parameterized, but still use an array type with a parameterized element type. This is legal, but generates an unchecked warning. Indeed, the code is unsafe, and eventually an error occurs.

```
List<String>[] lsa = new List<?>[10]; // unchecked warning - this is unsafe!
Object o = lsa;
Object[] oa = (Object[]) o;
List<Integer> li = new ArrayList<Integer>();
li.add(new Integer(3));
oa[1] = li; // correct
String s = lsa[1].get(0); // run time error, but we were warned
```

Similarly, attempting to create an array object whose element type is a type variable causes a compile-time error:

```
<T> T[] makeArray(T t) {
   return new T[100]; // error
}
```

Since type variables don't exist at run time, there is no way to determine what the actual array type would be.

The way to work around these kinds of limitations is to use class literals as run time type tokens, as described in section 8.

8 Class Literals as Run-time Type Tokens

One of the changes in JDK 1.5 is that the class java.lang.Class is generic. It's an interesting example of using genericity for something other than a container class.

Now that Class has a type parameter T, you might well ask, what does T stand for? It stands for the type that the Class object is representing.

For example, the type of String.class is Class<String>, and the type of Serializable.class is Class<Serializable>. This can be used to improve the type safety of your reflection code.

In particular, since the newInstance() method in Class now returns a T, you can get more precise types when creating objects reflectively.

For example, suppose you need to write a utility method that performs a database query, given as a string of SQL, and returns a collection of objects in the database that match that query.

One way is to pass in a factory object explicitly, writing code like:

```
interface Factory<T> { T make();}
public <T> Collection<T> select(Factory<T> factory, String statement) {
      Collection<T> result = new ArrayList<T>();
      /* run sql query using jdbc */
      for (/* iterate over jdbc results */ ) {
         T item = factory.make();
         /* use reflection and set all of item's fields from sql results */
         result.add(item);
      }
      return result;
}
```

You can call this either as

```
select(new Factory<EmpInfo>(){ public EmpInfo make() {
                                    return new EmpInfo();
                                }}
     , "selection string");
```

or you can declare a class EmpInfoFactory to support the Factory interface

```
class EmpInfoFactory implements Factory<EmpInfo> {
    ...
    public EmpInfo make() { return new EmpInfo();}
}
```

and call it

```
select(getMyEmpInfoFactory(), "selection string");
```

The downside of this solution is that it requires either:

- the use of verbose anonymous factory classes at the call site, or
- declaring a factory class for every type used and passing a factory instance at the call site, which is somewhat unnatural.

It is very natural to use the class literal as a factory object, which can then be used by reflection. Today (without generics) the code might be written:

```
Collection emps = sqlUtility.select(EmpInfo.class, "select * from emps");
...
public static Collection select(Class c, String sqlStatement) {
    Collection result = new ArrayList();
    /* run sql query using jdbc */
    for ( /* iterate over jdbc results */ ) {
        Object item = c.newInstance();
        /* use reflection and set all of item's fields from sql results */
        result.add(item);
    }
    return result;
}
```

However, this would not give us a collection of the precise type we desire. Now that Class is generic, we can instead write

```
Collection<EmpInfo> emps =
                sqlUtility.select(EmpInfo.class, "select * from emps");
...
public static <T> Collection<T> select(Class<T>c, String sqlStatement) {
    Collection<T> result = new ArrayList<T>();
    /* run sql query using jdbc */
    for ( /* iterate over jdbc results */ ) {
        T item = c.newInstance();
        /* use reflection and set all of item's fields from sql results */
        result.add(item);
        }
    return result;
}
```

giving us the precise type of collection in a type safe way.

This technique of using class literals as run time type tokens is a very useful trick to know. It's an idiom that's used extensively in the new APIs for manipulating annotations, for example.

9 More Fun with Wildcards

In this section, we'll consider some of the more advanced uses of wildcards. We've seen several examples where bounded wildcards were useful when reading from a data structure. Now consider the inverse, a write-only data structure.

The interface Sink is a simple example of this sort.

```
interface Sink<T> {
  flush(T t);
}
```

We can imagine using it as demonstrated by the code below. The method writeAll() is designed to flush all elements of the collection coll to the sink snk, and return the last element flushed.

```
public static <T> T writeAll(Collection<T> coll, Sink<T> snk){
      T last;
      for (T t : coll) {
          last = t;
          snk.flush(last);
       }
       return last;
}
...
Sink<Object> s;
Collection<String> cs;
String str = writeAll(cs, s); // illegal call
```

As written, the call to writeAll() is illegal, as no valid type argument can be inferred; neither String nor Object are appropriate types for T, because the Collection element and the Sink element must be of the same type.

We can fix this by modifying the signature of writeAll() as shown below, using a wildcard.

```
public static <T> T writeAll(Collection<? extends T>, Sink<T>){...}
...
String str = writeAll(cs, s); // call ok, but wrong return type
```

The call is now legal, but the assignment is erroneous, since the return type inferred is Object because T matches the element type of s, which is Object.

The solution is to use a form of bounded wildcard we haven't seen yet: wildcards with a *lower bound*. The syntax ? **super** T denotes an unknown type that is a supertype of T[3]. It is the dual of the bounded wildcards we've been using, where we use ? **extends** T to denote an unknown type that is a subtype of T.

```
public static <T> T writeAll(Collection<T> coll, Sink<? super T> snk){...}
...
String str = writeAll(cs, s); // Yes!
```

Using this syntax, the call is legal, and the inferred type is String, as desired.

[3]Or T itself. Remember, the supertype relation is reflexive.

Now let's turn to a more realistic example. A java.util.TreeSet<E> represents a tree of elements of type E that are ordered. One way to construct a TreeSet is to pass a Comparator object to the constructor. That comparator will be used to sort the elements of the TreeSet according to a desired ordering.

```
TreeSet(Comparator<E> c)
```

The Comparator interface is essentially:

```
interface Comparator<T> {
  int compare(T fst, T snd);
}
```

Suppose we want to create a TreeSet<String> and pass in a suitable comparator, We need to pass it a Comparator that can compare Strings. This can be done by a Comparator<String>, but a Comparator<Object> will do just as well. However, we won't be able to invoke the constructor given above on a Comparator<Object>. We can use a lower bounded wildcard to get the flexibility we want:

```
TreeSet(Comparator<? super E> c)
```

This allows any applicable comparator to be used.

As a final example of using lower bounded wildcards, lets look at the method Collections.max(), which returns the maximal element in a collection passed to it as an argument.

Now, in order for max() to work, all elements of the collection being passed in must implement Comparable. Furthermore, they must all be comparable to *each other.*

A first attempt at generifying this method signature yields

```
public static <T extends Comparable<T>>
      T max(Collection<T> coll)
```

That is, the method takes a collection of some type T that is comparable to itself, and returns an element of that type. This turns out to be too restrictive.

To see why, consider a type that is comparable to arbitrary objects

```
class Foo implements Comparable<Object> {...}
...
Collection<Foo> cf = ...;
Collections.max(cf); // should work
```

Every element of cf is comparable to every other element in cf, since every such element is a Foo, which is comparable to any object, and in particular to another Foo. However, using the signature above, we find that the call is rejected. The inferred type must be Foo, but Foo does not implement Comparable<Foo>.

It isn't necessary that T be comparable to **exactly** itself. All that's required is that T be comparable to one of its supertypes. This give us: [4]

[4]The actual signature of Collections.max() is more involved. We return to it in section 10

```
public static <T extends Comparable<? super T>>
    T max(Collection<T> coll)
```

This reasoning applies to almost any usage of Comparable that is intended to work for arbitrary types: You always want to use Comparable<? **super** T>.

In general, if you have an API that only uses a type parameter T as an argument, its uses should take advantage of lower bounded wildcards (? **super** T). Conversely, if the API only returns T, you'll give your clients more flexibility by using upper bounded wildcards (? **extends** T).

9.1 Wildcard Capture

It should be pretty clear by now that given

```
Set<?> unknownSet = new HashSet<String>();
...
/** Add an element t to a Set s */
public static <T> void addToSet(Set<T> s, T t) {...}
```

The call below is illegal.

```
addToSet(unknownSet, "abc"); // illegal
```

It makes no difference that the actual set being passed is a set of strings; what matters is that the expression being passed as an argument is a set of an unknown type, which cannot be guaranteed to be a set of strings, or of any type in particular.

Now, consider

```
class Collections {
...
    <T> public static Set<T> unmodifiableSet(Set<T> set) { ... }
}
...
Set<?> s = Collections.unmodifiableSet(unknownSet); // this works! Why?
```

It seems this should not be allowed; yet, looking at this specific call, it is perfectly safe to permit it. After all, unmodifiableSet() does work for any kind of Set, regardless of its element type.

Because this situation arises relatively frequently, there is a special rule that allows such code under very specific circumstances in which the code can be proven to be safe. This rule, known as *wildcard capture*, allows the compiler to infer the unknown type of a wildcard as a type argument to a generic method.

10 Converting Legacy Code to Use Generics

Earlier, we showed how new and legacy code can interoperate. Now, it's time to look at the harder problem of "generifying" old code.

If you decide to convert old code to use generics, you need to think carefully about how you modify the API.

You need to make certain that the generic API is not unduly restrictive; it must continue to support the original contract of the API. Consider again some examples from java.util.Collection. The pre-generic API looks like:

```
interface Collection {
  public boolean containsAll(Collection c);
  public boolean addAll(Collection c);
}
```

A naive attempt to generify it is:

```
interface Collection<E> {
  public boolean containsAll(Collection<E> c);
  public boolean addAll(Collection<E> c);
}
```

While this is certainly type safe, it doesn't live up to the API's original contract. The containsAll() method works with any kind of incoming collection. It will only succeed if the incoming collection really contains only instances of E, but:

- The static type of the incoming collection might differ, perhaps because the caller doesn't know the precise type of the collection being passed in, or perhaps because it is a Collection<S>,where S is a subtype of E.
- It's perfectly legitimate to call containsAll() with a collection of a different type. The routine should work, returning false.

In the case of addAll(), we should be able to add any collection that consists of instances of a subtype of E. We saw how to handle this situation correctly in section 5.

You also need to ensure that the revised API retains binary compatibility with old clients. This implies that the erasure of the API must be the same as the original, ungenerified API. In most cases, this falls out naturally, but there are some subtle cases. We'll examine one of the subtlest cases we've encountered, the method Collections.max(). As we saw in section 9, a plausible signature for max() is:

```
public static <T extends Comparable<? super T>>
      T max(Collection<T> coll)
```

This is fine, except that the erasure of this signature is

```
public static Comparable max(Collection coll)
```

which is different than the original signature of max():

```
public static Object max(Collection coll)
```

One could certainly have specified this signature for max(), but it was not done, and all the old binary class files that call Collections.max() depend on a signature that returns Object.

We can force the erasure to be different, by explicitly specifying a superclass in the bound for the formal type parameter T.

```
public static <T extends Object & Comparable<? super T>>
    T max(Collection<T> coll)
```

This is an example of giving *multiple bounds* for a type parameter, using the syntax T1& T2 ... & Tn. A type variable with multiple bounds is known to be a subtype of all of the types listed in the bound. When a multiple bound is used, the first type mentioned in the bound is used as the erasure of the type variable.

Finally, we should recall that max only reads from its input collection, and so is applicable to collections of any subtype of T.

This brings us to the actual signature used in the JDK:

```
public static <T extends Object & Comparable<? super T>>
    T max(Collection<? extends T> coll)
```

It's very rare that anything so involved comes up in practice, but expert library designers should be prepared to think very carefully when converting existing APIs.

Another issue to watch out for is *covariant returns*, that is, refining the return type of a method in a subclass. You should not take advantage of this feature in an old API. To see why, let's look at an example.

Assume your original API was of the form

```
public class Foo {
  public Foo create(){...}// Factory, should create an instance of whatever class it is declared in
}
public class Bar extends Foo {
  public Foo create(){...} // actually creates a Bar
}
```

Taking advantage of covariant returns, you modify it to:

```
public class Foo {
  public Foo create(){...} // Factory, should create an instance of whatever class it is declared in
}
public class Bar extends Foo {
  public Bar create(){...} // actually creates a Bar
}
```

Now, assume a third party client of your code wrote

```
public class Baz extends Bar {
  public Foo create(){...} // actually creates a Baz
}
```

The Java virtual machine does not directly support overriding of methods with different return types. This feature is supported by the compiler. Consequently, unless

the class Baz is recompiled, it will not properly override the create() method of Bar. Furthermore, Baz will have to be modified, since the code will be rejected as written - the return type of create() in Baz is not a subtype of the return type of create() in Bar.

11 Acknowledgements

Erik Ernst, Christian Plesner Hansen, Jeff Norton, Mads Torgersen, Peter von der Ahé and Philip Wadler contributed material to this tutorial.

Thanks to David Biesack, Bruce Chapman, David Flanagan, Neal Gafter, Örjan Petersson, Scott Seligman, Yoshiki Shibata and Kresten Krab Thorup for valuable feedback on earlier versions of this tutorial. Apologies to anyone whom I've forgotten.