Java Generics

Handout written by Nick Parlante

In my opinion, Java generics are a mixed blessing. Some uses of generics are simple to understand and make the code cleaner. They are a real improvement in the language, and these are the scenarios where we will concentrate. There are other, more complex uses of generics that I find hard to read, and I'm not convinced they are worth using at all.

Three Uses Of Generics

- Here are the three common use patterns of generics that I think are worth knowing (examples below)…
- 1. Using a generic class, like using ArrayList<String>
- 2. Writing generic code with a simple <T> or <?> type parameter
- 3. Writing generic code with a <T extends Foo> type parameter

Generic 1 -- Use Generic Class

- Many Java library classes have been made generic, so instead of just raw Object, they can be used in a way that indicates the type of object they hold.
- For example, the code below creates an ArrayList that holds Strings. Both the variable holding the pointer and the call to the constructor have <String> added.

```
ArrayList<String> strings = new ArrayList<String>();
strings.add("hi");
strings.add("there");
String s = strings.get(0); // no cast required!
```

- The advantage is that the compiler knows that the add() and get() methods take and return Strings, and so it can do the right type checking at compile time. We do not have to put in the (String) cast. The code that uses the list is now more readable. As a benefit of compile time typing, Eclipse code-assist now knows that strings.get(0) is a String and so can do code completion.
- The type of the iterator -- e.g. Iterator<String> -- must match the type of the collection.
- The plain type "List" without any generic type is known as the "raw" version. Raw versions still work in Java, and they essentially just contain pointers of type Object. You can assign back and forth between generic and raw versions, and it works, although it may give a warning.
  - e.g. You can store a List<String> pointer in a variable just of type List. This works, but gives a warning that it's better to keep it in List<String> the whole time.
- At runtime, all the casts are checked in any case, so if the wrong sort of object gets into a List<String> it will be noticed at runtime. This is why Java lets us mix between List<String> and raw List with just a warning-- it's all being checked at runtime anyway as a last resort.
  - e.g. String s = strings.get(0); // checks at runtime that the pointer coming out really is a String
- Eclipse tip: with the cursor where the <String> would go, hit ctrl-space. If Eclipse can deduce from context that a <String> or whatever is required, it puts it in for you, yay!

Boxing/Unboxing

- Normally, you cannot store an int or a boolean in an ArrayList, since it can only store pointers to objects. You cannot create an ArrayList<int>, but you can create an ArrayList<Integer>.
- With Java 5 "auto boxing", when the code needs an Integer but has an int, it automatically creates the Integer on the fly, without requiring anything in the source code. Going the other direction, auto

unboxing, if the code has an Integer but needs an int, it automatically calls intValue() on the Integer to get the int value.

- This works for all the primitives -- int, char double, boolean, …
- This works especially well with generic collections, taking advantage of the fact that the collection type shows that it needs Integer or Boolean or whatever.

Warning: Unboxing Does Not Work With == !=

- Auto unboxing does not work correctly for == and !=.
- e.g. with two List<Integer> a and b, a.get(0) == b.get(0) -- does not unbox. Instead, it does an == pointer comparison between the two Integer objects, which is almost certainly not what you want.
- Use a.get(0).intValue() == b.get(0).intValue() to force the conversion to int. Or you could write a.get(0).equals(b.get(0)) which works for String, Integer, etc.
- It works this way to remain compatible with the original definition of ==, but I hope this particular aspect of backward compatibility is dropped someday.

Foreach Loop

- Java's for-loop that iterates over any collection or array:

```
List<String> strings = ...

for (String s: strings) {
  System.out.println(s);
}
```

- This is a shorthand for looping over the elements with an iterator that calls hasNext()/next() in the usual way. The fancy iterator features -- such as remove() -- are not available. Nonetheless, this syntax is very handy for the very common case of iterating over any sort of collection.
- The foreach does not provide a way to change the collection. In the loop, a statement like "s = something" just changes the local variable s, not that slot in the collection. However "s.doSomething()" is effective, since the local s points to the same object pointed by the collection.
- Design Lesson: if the clients of a system perform a particular operation very commonly (in this case, iterating over all the elements in a collection) -- it's a good design idea to have a simple facility that makes that common case very easy. It's ok if the facility does not handle more advanced uses. It can be simple and focus just on the common case. Putting in such a special-purpose facility is, in a way, inelegant -- it creates more than one way to do things. However, experience shows that making common cases very easy is worthwhile.

Generic List Example Code

```
    // Shows basic use of the ArrayList and iterators
    // with the generics.
    public static void demoList() {
        // ** Create a List<String>
        List<String> a = new ArrayList<String>();
        a.add("Don't");
        a.add("blame");
        a.add("me");

        // ** foreach -- iterate over collection easily
        for (String str: a) {
            System.out.println(str);
        }

        // ** Instead of Iterator, make an Iterator<String>
        Iterator<String> it = a.iterator();
        while (it.hasNext()) {
            // NOTE: no cast required here -- it.next() is a String
            String string = it.next();
            System.out.println(string);
        }

        // ** Likewise, can make a List<Integer>
        List<Integer> ints = new ArrayList<Integer>();
        for (int i = 0; i < 10; i++) {
            ints.add(new Integer(i * i));
        }

        // No casts needed here -- it knows they are Integer
        int sum = ints.get(0).intValue() + ints.get(1).intValue();

        // With auto Unboxing, can just write it like this...
        sum = ints.get(0) + ints.get(1);

        // Can go back and forth between typed Collections and untyped "raw"
        // forms -- may get a warning.
        List<String> genList = new ArrayList(); // warning
        List rawList = new ArrayList<String>(); // no warning
        rawList.add("hello"); // warning
        genList = rawList; // warning
        rawList = genList; // no warning
```

Generic Map Example Code

```
    public static void demoMap() {
        // ** Make a map, specifying both key and value types
        HashMap<Integer, String> map = new HashMap<Integer, String>();

        // Map Integers to their words
        map.put(new Integer(1), "one");
        map.put(new Integer(2), "two");
        map.put(3, "three"); // Let the autoboxing make the Integer
        map.put(4, "four");

        String s = map.get(new Integer(3)); // returns type String
        s = map.get(3); // Same as above, with autoboxing
        // map.put("hi", "there");  // NO does not compile

        // ** Auto unboxing -- converts between Integer and int
        Integer intObj = new Integer(7);
        int sum = intObj + 3; // intObj unboxes automatically to an int, sum is 10

        // ** More complex example -- map strings to lists of Integer
        HashMap<String, List<Integer>> counts = new HashMap<String, List<Integer>>();

        List<Integer> evens = new ArrayList<Integer>();
        evens.add(2);
        evens.add(4);
        evens.add(6);
        counts.put("evens", evens);

        // Get the List<Integer> back out...
        List<Integer> evens2 = counts.get("evens");
        sum = evens2.get(0) + evens2.get(1); // unboxing here, sum is 6
    }
```

Generic 2 -- Define a Generic <T> Class/Method

- You can define your own class as a generic class. The class definition code is parameterized by a type, typically called <T>. This is more or less what ArrayList does. At the very start of the class, the parameter is added like this:
  `public class Foo<T> { ...`
- In the simplest case, the class code doe not have any specific constraint on what type T is. Inside the class, T may be used in limited ways:
  - declare variables, parameters, and return types of the type T
  - use = on T pointers
  - call methods that work on all Objects, like .equals()

Generic <T> Operations, Erasure

- In the source code implementing the generic class, T may be used to declare variables and return types, and may be used to declare local generic variables, such as a List<T>. Essentially, T behaves like a real type such as String or Integer.
- At a later stage in the compilation, T is replaced by Object. So at run-time, the notion of T is gone -- it's just Object by then. This is known as the "erasure" system, where T plays its role to check the sources early in compilation, but is then erased down to just plain Object.
- Remember: where you see "T", it is just replaced by "Object" to produce the code for runtime. So the ArrayList<String> code and the ArrayList<Integer> code … those two are actually just the ArrayList<Object> code at runtime.
- Therefore things that need T at runtime do not work -- e.g. creating a "T" array with code like: "new T[100]". This cannot work -- at runtime, there is no T, just Object.
- The erasure system provides basic generic type checking at compile time. The erasure approach is a compromise that adds generics to Java while remaining compatible with pre Java 5 systems.
- In Java 7 (roughly 2008), it is possible that some of the limitations of the erasure system will be removed.

Generic <T> Pair Example (add List<T>)

```
/*
 Generic <T> "Pair" class contains two objects, A and B, and supports
 a few operations on the pair. The type T of A and B is a generic
 parameter.

 This class demonstrates parameterization by a <T> type.
 The class does not depend on any feature of the T type --
 just uses "=" to store and return its pointers. The T type can be used
 for variables, parameters and return types. This approach is quite easy to do,
 and solves all the cases where we don't really care about the T type.
 This is basicaly how ArrayList and HashMap do it.
*/

public class Pair<T> {
    private T a;             // Can declare T variables
    private T b;
    private List<T> unused;  // Can use T like this too

    public Pair(T a, T b) {
        this.a = a;
        this.b = b;
    }

    public T getA() {
        return a;
    }

    public T getB() {
        return b;
    }

    public void swap() {
        T temp = a;  // NOTE T temporary variable
        a = b;
        b = temp;
    }

    // True if a and b are the same
    public boolean isSame() {
        return a.equals(b);
        // NOTE Can only do things on T vars that work on any Object
    }

    // True if a or b is the given object
    // NOTE: use of T as a parameter
    public boolean contains(T elem) {
        return (a.equals(elem) || b.equals(elem));
    }

    public static void main(String[] args) {
        // ** Make a Pair of Integer
        Pair<Integer> ipair = new Pair<Integer>(1, 2);
        Integer a = ipair.getA();
        int b = ipair.getB();  // unboxing

        // ** Also make a Pair of String
        Pair<String> spair = new Pair<String>("hi", "there");
        String s = spair.getA();
    }

    // Show things that do not work with T
    private void doesNotWork(Object x) {
        // T var = new T();  // NO, T not real at runtime -- erasure

        // T[] array = new T[10];  // NO, same reason

        // T temp = (T) x;  // NO, same reason (like (Object) cast)
    }
}
```

Generic <T> Method

- Rather than making a whole class generic, the generic syntax can work on a single method. The syntax is that the <T> goes before the return type: **`public <T> void foo(List<T> list) {`**
- Here is an example where T allows the method to work for any sort of List:

```
// <T> Method -- use a <T> type on the method to
// identify what type of element is in the collection.
// The <T> goes just before the return type.
// T can be used to declare variables, return types, etc.
// This is ok, but slightly heavyweight, since in this case
// we actually don't care what type of thing is in there.
// This removes elements that are == to an adjacent element.
public static <T> void removeAdjacent(Collection<T> coll) {
    Iterator<T> it = coll.iterator();
    T last = null;
    while (it.hasNext()) {
        T curr = it.next();
        if (curr == last)
            it.remove();
        last = curr;
    }
}
```

Generic <?> Method

- The <?> generic parameter is like a <T> parameter but a little simpler. The "?" is a wildcard which I think of as meaning "don't care" -- a type that can be anything, since the code does not care what it is.
- Here is a bar() that uses "?" to take a list of anything. The syntax is a little more simple, since there is not an extra <T> required before the return type. The <?> just goes in the argument list and that's it:
  `void bar(List<?> list) {`
- We can use the list inside bar(), but we cannot assume anything about what type it contains -- we don't even have a name for the type it contains.
- Inside the List<?> method, we can make local generic variables that mention the <?>, but that's the only use of <?> that works, such as:

```
    List<?> temp = list;
    Iterator<?> it = list.iterator();
```

- We cannot declare a variable of type "?", however "Object" works fine:

```
        Object elem = list.get(0);
```

- You do not **return** a <?> type -- in that case you want a <T> in the return type and also somewhere else specified by the client, such as a parameter. The <?> works for cases so simple that the <?> only appears in one place. For multiple <>, use <T>.

```
// <?> Method -- Use a "wildcard" <?> on the parameter, which
// is basically a "don't care" marker. The <?> is not listed
// before the return type.
// We cannot declare variables of type ?, but we can use Object
// instead.
// We can declare local generic variables that mention <?>.
public static void removeAdjacent2(Collection<?> coll) {
    Iterator<?> it = coll.iterator(); // Can use <?> in local variables
    Object last = null; // Cannot use ? as a var type, use Object
    while (it.hasNext()) {
        Object curr = it.next();
        if (curr == last)
            it.remove();
        last = curr;
    }
}
```

Passing Any Sort of List -- List<Object> vs. List<T>

- Suppose we want a method that takes as a parameter any sort of list -- list of Strings, list of Integers etc.
- At first glance, it looks like a parameter of type List<Object> could work as a catch-all for any sort of list. However, that does not work, because of the container anomaly described below.
- That is why List<T> and List<?> exist -- they are the way to encode that you take a List (or whatever) and it can contain any type.

Generic 3 -- ?/T with "extends" Generics

- The plain <?>/<T> generics doesn't let us assume anything about T. The "extends" form allows us to add a constraint about T.
- For example, with a <T extends Number> -- for any T value, we can assume it is a Number subclass, so .intValue() etc. just work. This is known as a "bounded wildcard".
- This technique can be used on a whole class, such as a Pair that contains numbers:

```
public class PairNumber <T extends Number> {
    private T a;
    private T b;

    public PairNumber(T a, T b) {
        this.a = a;
        this.b = b;
    }

    public int sum() {
        // Note: here we can use Number messages, like intValue().
        return (a.intValue() + b.intValue());
    }
    ...
```

?/T Extends Method

- The "extends" constraint also works on ?/T methods

```
// ?-Extends -- rather than Collection<?>, the "extends Number"
// adds a constraint that the elements must be subclasses of Number.
// The Number subclasses include Integer, Double, Long, etc.
// Returns the int sum of all the numbers in the collection.
public static int sumAll(Collection<? extends Number> nums) {
    int sum = 0;
    for (Number num : nums) {
        sum += num.intValue(); // All the Number types respond to intValue().
    }
    return sum;
}
```

Container Anomaly -- Why We Need Extends

- Why must we use List<T> to for a list of any type, rather than just List<Object>?
- There is a basic problem between the type of a container and the type of thing it contains.
- A pointer to a String (the subclass) can be stored in a variable of type Object (the superclass). We do that all the time.
- However, a pointer to a List<String> cannot be stored in a variable type List<Object> -- this is unintuitive, but it is a real constraint. The reason for this constraint is demonstrated below.

Container Anomaly Example Code

```
    // say we have this method
    public static void doSomething(List<Object> lo) {

    }

    public static void problem() {
        Object o;
        String s = "hello";
        o = s;  // fine

        List<Object> lo = new ArrayList<Object>();
        List<String> ls = new ArrayList<String>();

        // lo = ls;  // NO does not compile

        // The following would be a big prob if above were allowed.
        // If we allow ls to be in lo, we lose track of the constraint
        // that ls should only contain Strings.
        lo.add(new Integer(3));

        // doSomething(ls);  // NO same problem

        // The point: you can assign a pointer of type sub to a pointer of type
        // super. However, you cannot assign a pointer type container(sub) to a
        // pointer of type container(super).
        // Therefore Collection<Object> will not work as a sort of catch-all
        // type for any sort of collection.
```