Generic Programming

Simple Generic Methods

In addition to generic classes, Java also has generic methods. An example is the method `Collections.sort()`, which can sort collections of objects of any type. To see how to write generic methods, let's start with a non-generic method for counting the number of times that a given string occurs in an array of strings:

```
/**
 * Returns the number of times that itemToCount occurs in list.
Items in the
 * list are tested for equality using itemToCount.equals(),
except in the
 * special case where itemToCount is null.
 */
public static int countOccurrences(String[] list, String
itemToCount) {
   int count = 0;
   if (itemToCount == null) {
      for ( String listItem : list )
         if (listItem == null)
            count++;
   }
   else {
      for ( String listItem : list )
         if (itemToCount.equals(listItem))
            count++;
   }
   return count;
}
```

Once again, we have some code that works for type *String*, and we can imagine writing almost identical code to work with other types of objects. By writing a generic method, we get to write a single method definition that will work for objects of any type. We need to replace the specific type *String* in the definition of the method with the name of a type parameter, such as *T*. However, if that's the only change we make, the compiler will think that "T" is the name of an actual type, and it will mark it as an undeclared identifier. We need some way of telling the compiler that "T" is a type parameter. That's what the "`<T>`" does in the definition of the generic class "`class Queue<T> { ...`". For a generic method, the "`<T>`" goes just before the name of the return type of the method:

```
public static <T> int countOccurrences(T[] list, T itemToCount)
{
   int count = 0;
   if (itemToCount == null) {
      for ( T listItem : list )
         if (listItem == null)
            count++;
   }
   else {
      for ( T listItem : list )
         if (itemToCount.equals(listItem))
            count++;
   }
   return count;
}
```

The "`<T>`" marks the method as being generic and specifies the name of the type parameter that will be used in the definition. Of course, the name of the type parameter doesn't have to be "T"; it can be anything. (The "`<T>`" looks a little strange in that position, I know, but it had to go somewhere and that's just where the designers of Java decided to put it.)

Given the generic method definition, we can apply it to objects of any type. If `wordList` is a variable of type `String[]` and `word` is a variable of type *String*, then

```
   int ct = countOccurrences( wordList, word );
```

will count the number of times that `word` occurs in `wordList`. If `palette` is a variable of type `Color[]` and `color` is a variable of type *Color*, then

```
   int ct = countOccurrences( palette, color );
```

will count the number of times that `color` occurs in `palette`. If `numbers` is a variable of type `Integer[]`, then

```
   int ct = countOccurrences( numbers, 17 );
```

will count the number of times that `17` occurs in `numbers`. This last example uses autoboxing; the 17 is automatically converted to a value of type *Integer*, as if we had said "`countOccurrences( numbers, new Integer(17) )`". Note that, since generic programming in Java applies only to objects, we **cannot** use `countOccurrences` to count the number of occurrences of 17 in an array of type `int[]`.

A generic method can have one or more type parameters, such as the "T" in `countOccurrences`. Note that when a generic method is used, as in the function call "`countOccurrences(wordlist, word)`", there is no explicit mention of the type that is substituted for the type parameter. The compiler deduces the type from the types of the actual parameters in the method call. Since `wordlist` is of type `String[]`, the compiler can tell that in "`countOccurrences(wordlist, word)`", the type that replaces *T* is *String*. This contrasts with the use of generic classes, as in "`new Queue<String>()`", where the type parameter is specified explicitly.

The `countOccurrences` method operates on an array. We could also write a similar method to count occurrences of an object in any collection:

```
public static <T> int countOccurrences(Collection<T>
collection, T itemToCount) {
   int count = 0;
   if (itemToCount == null) {
      for ( T item : collection )
         if (item == null)
            count++;
   }
   else {
      for ( T item : collection )
         if (itemToCount.equals(item))
            count++;
   }
   return count;
}
```

Since *Collection<T>* is itself a generic type, this method is very general. It can operate on an *ArrayList* of *Integers*, a *TreeSet* of *Strings*, a *LinkedList* of *JButtons*, ...

Type Wildcards

There is a limitation on the sort of generic classes and methods that we have looked at so far: The type parameter in our examples, usually named *T*, can be any type at all. This is OK in many cases, but it means that the only things that you can do with *T* are things that can be done with **every** type, and the only things that you can do with objects of type *T* are things that you can do with **every** object. With the techniques that we have covered so far, you can't, for example, write a generic method that compares objects with the `compareTo()` method, since that method is not defined for all objects. The `compareTo()` method is defined in the *Comparable* interface. What we need is a way of specifying that a generic class or method only applies to objects of type *Comparable* and not to arbitrary objects. With that restriction, we should be free to use `compareTo()` in the definition of the generic class or method.

There are two different but related syntaxes for putting restrictions on the types that are used in generic programming. One of these is bounded type parameters, which are used as formal type parameters in generic class and method definitions; a bounded type parameter would be used in place of the simple type parameter *T* in "`class GenericClass<T> ...`" or in "`public static <T> void genericMethod(...`". The second syntax is wildcard types, which are used as type parameters in the declarations of variables and of formal method parameters; a wildcard type could be used in place of the type parameter *String* in the declaration statement "`List<String> list;`" or in the formal parameter list "`void max(Collection<String> c)`". We will look at wildcard types first, and we will return to the topic of bounded types later in this section.

Let's start with a simple example in which a wildcard type is useful. Suppose that *Shape* is a class that defines a method `public void draw()`, and suppose that *Shape* has subclasses such as *Rect* and *Oval*. Suppose that we want a method that can draw all the shapes in a collection of *Shapes*. We might try:

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

This method works fine if we apply it to a variable of type *Collection<Shape>*, or *ArrayList<Shape>*, or any other collection class with type parameter *Shape*. Suppose, however, that you have a list of *Rects* stored in a variable named `rectangles` of type *Collection<Rect>*. Since *Rects* are *Shapes*, you might expect to be able to call `drawAll(rectangles)`. Unfortunately, this will not work; a collection of *Rects* is **not** considered to be a collection of *Shapes*! The variable `rectangles` cannot be assigned to the formal parameter `shapes`. The solution is to replace the type parameter "`Shape`" in the declaration of `shapes` with the wildcard type "`? extends Shape`":

```
public static void drawAll(Collection<? extends Shape> shapes)
{
   for ( Shape s : shapes )
      s.draw();
}
```

The wildcard type "`? extends Shape`" means roughly "any type that is either equal to *Shape* or that is a subclass of *Shape*". When the parameter `shapes` is declared to be of type *Collection<? extends Shape>*, it becomes possible to call the `drawAll` method with an actual parameter of type *Collection<Rect>* since *Rect* is a subclass of *Shape* and therefore matches the wildcard "`? extends Shape`". We could also pass actual parameters to `drawAll` of type *ArrayList<Rect>* or *Set<Oval>* or *List<Oval>*. And we can still pass variables of type *Collection<Shape>* or *ArrayList<Shape>*, since the class *Shape* itself matches "`? extends Shape`". We have greatly increased the usefulness of the method by using the wildcard type.

(Although it is not essential, you might be interested in knowing *why* Java does not allow a collection of *Rects* to be used as a collection of *Shapes*, even though every *Rect* is considered to be a *Shape*. Consider the rather silly but legal method that adds an oval to a list of shapes:

```
static void addOval(List<Shape> shapes, Oval oval) {
   shapes.add(oval);
}
```

Suppose that `rectangles` is of type *List<Rect>*. It's illegal to call `addOval(rectangles,oval)`, because of the rule that a list of *Rects* is not a list of *Shapes*. If we dropped that rule, then `addOval(rectangles,oval)` would be legal, and it would add an *Oval* to a list of *Rects*. This would be bad: Since *Oval* is not a subclass of *Rect*, an *Oval* is **not** a *Rect*, and a list of *Rects* should never be able to contain an *Oval*. The method call `addOval(rectangles,oval)` does not make sense and **should** be illegal, so the rule that a collection of *Rects* is not a collection of *Shapes* is a good rule.)

As another example, consider the method `addAll()` from the interface *Collection<T>*. In my description of this method in Subsection 10.1.4, I say that for a collection, `coll`, of type *Collection<T>*, `coll.addAll(coll2)` "adds all the objects in `coll2` to `coll`. The parameter, `coll2`, can be any collection of type *Collection<T>*. However, it can also be more general. For example, if *T* is a class and *S* is a sub-class of *T*, then `coll2` can be of type *Collection<S>*. This makes sense because any object of type *S* is automatically of type *T* and so can legally be added to `coll`." If you think for a moment, you'll see that what I'm describing here, a little awkwardly, is a use of wildcard types: We don't want to require `coll2` to be a collection of of objects of type *T*; we want to allow collections of any subclass of *T*. To be more specific, let's look at how a similar `addAll()` method could be added to the generic *Queue* class that was defined earlier in this section:

```
class Queue<T> {
   private LinkedList<T> items = new LinkedList<T>();
   public void enqueue(T item) {
      items.addLast(item);
   }
   public T dequeue() {
      return items.removeFirst();
   }
   public boolean isEmpty() {
      return (items.size() == 0);
   }
   public void addAll(Collection<? extends T> collection) {
         // Add all the items from the collection to the end of
the queue
      for ( T item : collection )
         enqueue(item);
   }
}
```

Here, *T* is a type parameter in the generic class definition. We are combining wildcard types with generic classes. Inside the generic class definition, "`T`" is used as if it is a specific, though unknown, type. The wildcard type "`? extends T`" means some type that extends that specific type. When we create a queue of type *Queue<Shape>*, "`T`" refers to "Shape", and the wildcard type "`? extends T`" in the class definition means "`? extends Shape`", meaning that the `addAll` method of the queue can be applied to collections of *Rects* and *Ovals* as well as to collections of *Shapes*.

The for-each loop in the definition of `addAll` iterates through the `collection` using a variable, `item`, of type *T*. Now, `collection` can be of type *Collection<S>*, where *S* is a subclass of *T*. Since `item` is of type *T*, not *S*, do we have a problem here? No, no problem. As long as *S* is a subclass of *T*, a value of type *S* can be assigned to a variable of type *T*. The restriction on the wildcard type makes everything work nicely.

The `addAll` method adds all the items from a collection to the queue. Suppose that we wanted to do the opposite: Add all the items that are currently on the queue to a given collection. An instance method defined as

```
public void addAllTo(Collection<T> collection)
```

would only work for collections whose base type is exactly the same as *T*. This is too restrictive. We need some sort of wildcard. However, "`? extends T`" won't work. Suppose we try it:

```
public void addAllTo(Collection<? extends T> collection) {
      // Remove all items currently on the queue and add them
to collection
   while ( ! isEmpty() ) {
      T item = dequeue();  // Remove an item from the queue.
      collection.add( item );  // Add it to the collection.
ILLEGAL!!
   }
}
```

The problem is that we can't add an `item` of type *T* to a collection that might only be able to hold items belonging to some subclass, *S*, of *T*. The containment is going in the wrong direction: An `item` of type *T* is not necessarily of type *S*. For example, if we have a queue of type *Queue<Shape>*, it doesn't make sense to add items from the queue to a collection of type *Collection<Rect>*, since not every *Shape* is a *Rect*. On the other hand, if we have a *Queue<Rect>*, it would make sense to add items from that queue to a *Collection<Shape>* or indeed to any collection *Collection<S>* where *S* is a **super**class of *Rect*.

To express this type of relationship, we need a new kind of type wildcard: "`? super T`". This wildcard means, roughly, "either *T* itself or any class that is a superclass of *T*." For example, *Collection<? super Rect>* would match the types *Collection<Shape>*, *ArrayList<Object>*, and *Set<Rect>*. This is what we need for our `addAllTo` method. With this change, our complete generic queue class becomes:

```
class Queue<T> {
   private LinkedList<T> items = new LinkedList<T>();
   public void enqueue(T item) {
      items.addLast(item);
   }
   public T dequeue() {
      return items.removeFirst();
   }
   public boolean isEmpty() {
      return (items.size() == 0);
   }
   public void addAll(Collection<? extends T> collection) {
         // Add all the items from the collection to the end of
the queue
      for ( T item : collection )
         enqueue(item);
   }
   public void addAllTo(Collection<? super T> collection) {
         // Remove all items currently on the queue and add
them to collection
      while ( ! isEmpty() ) {
         T item = dequeue();  // Remove an item from the queue.
         collection.add( item );  // Add it to the collection.
      }
   }
}
```

In a wildcard type such as "`? extends T`", *T* can be an `interface` instead of a class. Note that the term "`extends`" (not "`implements`") is used in the wildcard type, even if *T* is an interface. For example, recall that *Runnable* is an `interface` that defines the method `public void run()`. Runnable objects are usually associated with threads (see Section 8.5). Here is a method that runs all the objects in a collection of *Runnables* in parallel, by creating a separate thread for each object:

```
public static runAllInParallel( Collection<? extends Runnable>
runnables ) {
   for ( Runnable runnable : runnables ) {
      Thread runner;  // A thread to run the method
runnable.run()
      runner = new Thread( runnable );   // Create the thread.
      runner.start();  // Start the thread running.
   }
}
```

Wildcard types are used **only** as type parameters in parameterized types, such as *Collection<? extends Runnable>*. The place where a wildcard type is most likely to occur, by far, is in a formal parameter list, where the wildcard type is used in the declaration of the type of a formal parameter. However, they can also be used in a few other places. For example, they can be used in the type specification in a variable declaration statement.

One final remark: The wildcard type "`<?>`" is equivalent to "`<? extends Object>`". That is, it matches any possible type. For example, the `removeAll()` method in the generic interface *Collections<T>* is declared as

```
public boolean removeAll( Collection<?> c ) { ...
```

This just means that the `removeAll` method can be applied to any collection of any type of object.

Bounded Types

Wildcard types don't solve all of our problems. They allow us to generalize method definitions so that they can work with collections of objects of various types, rather than just a single type.

However, they do not allow us to restrict the types that are allowed as type parameters in a generic class or method definition. Bounded types exist for this purpose.

Let's look at a generic method in which a bounded type parameter is essential. The code below presented a code segment for inserting a string into a sorted list of strings, in such a way that the modified list is still in sorted order:

```
static void sortedInsert(List<String> sortedList, String
newItem) {
   ListIterator<String> iter = sortedList.listIterator();
   while (iter.hasNext()) {
      String item = iter.next();
      if (newItem.compareTo(item) <= 0) {
         iter.previous();
         break;
      }
   }
   iter.add(newItem);
}
```

This method works fine for lists of strings, but it would be nice to have a generic method that can be applied to lists of other types of objects. The problem, of course, is that the code assumes that the `compareTo()` method is defined for objects in the list, so the method can only work for lists of objects that implement the *Comparable* interface. We can't simply use a wildcard type to enforce this restriction. Suppose we try to do it, by replacing *List<String>* with *List<? extends Comparable>*:

```
static void sortedInsert(List<? extends Comparable> sortedList,
???? newItem) {
   ListIterator<????> iter = stringList.listIterator();
   ...
```

We immediately run into a problem, because we have no name for the unknown type represented by the wildcard. We **need** a name for that type because the type of `newItem` and of `iter` should be the same as the type of the items in the list. The problem is solved if we write a generic method with a bounded type parameter, since then we have a name for the unknown type, and we can write a valid generic method:

```
static <T extends Comparable> void sortedInsert(List<T>
sortedList, T newItem) {
   ListIterator<T> iter = sortedList.listIterator();
   while (iter.hasNext()) {
      T item = iter.next();
      if (newItem.compareTo(item) <= 0) {
         iter.previous();
         break;
      }
   }
   iter.add(newItem);
}
```

There is still one technicality to cover in this example. *Comparable* is itself a parameterized type, but I have used it here without a type parameter. This is legal but the compiler might give you a warning about using a "raw type." In fact, the objects in the list should implement the parameterized interface *Comparable<T>*, since they are being compared to items of type *T*. This just means that instead of using *Comparable* as the type bound, we should use *Comparable<T>*:

```
static <T extends Comparable<T>> void sortedInsert(List<T>
sortedList, ...
```