Chapter 14
Generics

Hello!

The topic for this chapter is generics. Collections rely heavily upon generics, and so they are also discussed at length now. Both generics and collections let us write code that works at many different types.

Collections

Java provides the Java Collections Framework (JCF), which gives us many powerful ways to create data structures (such as lists and maps and sets) and perform operations over them (such as add, remove, etc).

We will just briefly discuss the layout and then a couple of specific classes that we want some familiarity with (ArrayList, mainly).

The JCF uses some advanced features of the Java language:

- There are many interfaces that extend each other (yes, interfaces using the extends keyword!).
- There are many classes that implement specific interfaces. For instance, the ArrayList and LinkedList classes both implement the List interface.
- Basically all these interfaces and classes use generics: instead of the ArrayList class storing just Objects (not very useful), we have the ArrayList<E> class that stores objects of type E.

Here are some of the interfaces, shown in their own hierarchy:

- Collection<E>
  - Set<E>
    - SortedSet<E>
  - List<E>
  - Queue<E>
  - Deque<E>
- Map<K,V>
  - SortedMap<K,V>

Each one provides some group of abstract methods. Child interfaces provide all the abstract methods of the parent interface, plus any more that they define. We're particularly interested in the List<E> interface. Please take a moment to browse all the methods available, as shown in its API:

http://docs.oracle.com/javase/6/docs/api/java/util/List.html

We're thus also interested in classes that implement these interfaces, and the List<E> interface in particular. On the List<E> API, you can see "All Known Implementing Classes". Open up the API documentation for both ArrayList and LinkedList.

ArrayList
ArrayList<E> provides all the Listr<E> definitions, as well as one or two extras such as trimToSize(). An ArrayList is an implementation of a List that has a hidden array of values inside. It keeps track of how much space is currently not used. If we add too many items, the ArrayList will automatically create another array twice as long, copy everything over, and then continue (now that it has plenty of space to add values again). The trimToSize() method lets us say, "I removed a whole bunch of stuff from my ArrayList, so let's regain all that space that's no longer needed. Create an array juuuust big enough to hold the contents." While looking up items in the ArrayList is fast, inserting items is really slow.

LinkedList
The LinkedList<E> class also implements the List<E> interface, only it doesn't have an array hiding around. It relies upon lots of individual objects that each store a single value and then a reference to the next and previous items in the list. While inserting items into the LinkedList is pretty fast, looking up items in the list is slower.

Quick Comparison
We see that they provide basically the same methods, with actual implementations. So: a List<E> is some interface type that represents a list of E values. We can actually create lists of E values using an ArrayList<E> or a LinkedList<E>. There are a very small number of extra methods, for instance the ArrayList

Not Quite Your Turn… ☹
- In order to create things of type ArrayList, we need to use generics, or else the compiler gets really mad at us all over the place. These non-generic uses are called "raw types", and we don't want to use them, because Java can't perform the type checking we're used to relying upon.
- So let's get started on generics, and then we can eventually understand what it takes to make values like an `ArrayList<Integer>` or `ArrayList<String>`.
- Using an ArrayList will feel quite like a Python list (if you have experience with them). We just have to call methods for all operations, instead of the [ ] syntax here and there.

Generics

Motivation: too much re-writing
In Java, we occasionally find ourselves writing the same code, except that type types change in each version. Suppose we wanted to make our own pairs in Java (length-2 tuples). We would normally have to decide ahead of time what the type of the contained things are:

```
public class IntPair {
  public int fst, snd;
  public IntPair(int fst, int snd){
    this.fst = fst;
    this.snd = snd;
  }
  public String toString(){
    return "("+fst+","+snd+")";
  }
}

// another version, for doubles:
public class DoublePair {
  public double fst, snd;
  public DoublePair(double fst, double snd){
    this.fst = fst;
    this.snd = snd;
  }
  public String toString(){
    return "("+fst+","+snd+")";
  }
}
```

This can get out of hand. We need a separate copy of the class for every pair of types we'd like to put in a tuple!

```
// another version, for a String and an int:
public class StringIntPair {
      public String fst;
      public int snd;
      public StringIntPair(String fst, int snd){
            this.fst = fst; this.snd = snd;
      }
      public String toString(){return "("+fst+","+snd+")";}
}


// another version for each different usage!
public class IntStringPair    { public int    fst; public String snd; ... }
public class DoubleStringPair { public double fst; public String snd; ... }
public class StringDoublePair { public String fst; public double snd; ... }
public class IntListPair      { public Int[]  fst, snd; ... }
//  please, make it stop…
```

Motivation: too much casting

Well, we could try to get around it by just putting in Objects, once and for all:

```
public class ObjectsPair {
      public Object fst, snd;
      public ObjectsPair(Object fst, Object snd){
            this.fst = fst; this.snd = snd;
      }
      public String toString(){return "("+fst+","+snd+")";}
}
```

Our usage would then require us to perform casts on **everything** that came out of it:

```
      ObjectsPair op = new ObjectsPair(4, "hello");
      String s = (String) op.snd;
      //must convince Java to convert from Object to int via Integer...
      int i = (int) (Integer) op.fst;
      char loc = s.charAt(i);
      System.out.printf("found '%s' at index %d in %s.\n",loc, i, s);
```

Relying on the programmer for correctly performing all these casts is a bad idea. It would be much nicer if we didn't have to convince Java what values were there, and if it could just remember for us that this Pair is supposed to always hold specific types at each location (fst or snd, in our Pair example).

There was/is the ArrayList type, which provides all the convenience of lists (a la Python), even though the underlying representation uses arrays. Before generics were added to Java, we had the same problem: whatever we put into it with the add method, we forgot anything about it other than being an Object, and had to cast everything as it came out:

```
ArrayList xs = new ArrayList();
xs.add("hello");
xs.add(new Integer(3));
char c = ((String)xs.get(0)).charAt((int)(Integer)xs.get(1));
System.out.println("Char is '"+c+"'");
```

Raw Types
If you compile that code in modern versions of Java, you'll actually get warnings. Not errors (the code is still compiled for you), but it complains that you've used a "raw" type, which is to say you didn't use generics when you should have.

Hopefully by now, we're seeing the problem that generics solve: we don't want to have to re-write code at multiple specific types, but we also don't want to completely forget what types are being used.

Generics: Parameterizing Code with Types!

Generics allow us to write code that is parameterized over *types*. We're already quite comfortable parameterizing a method over values with our usual formal parameters list; we will now be able to write entire classes that are parameterized over types, so that we can write the once-and-for-all definition of Pair. We will be able to parameterize a method over types as well, as a sort of infinite overloading. (These methods still have their usual formal parameters list for values).

What does a type parameters list look like?
Whether used at the class or method level, we place new identifiers in angular brackets, and separate them with commas, such as:

<T>   <R,S>   <A, B, More, AnyName, Goes, Here>

We can choose any unique name for our type parameters. By convention, they ought to start capitalized, as they represent types. Also, since they can be instantiated by any type we can think of, it can be hard to think of meaningful names. Single-letter names are common, especially T (for type), E (for element), and so on.

Let's look at the example for Pair, as it ought to be written:

```
public class Pair<R,S> {
    public R fst;
    public S snd;
    public Pair(R fst, S snd){
        this.fst = fst;
        this.snd = snd;
    }
    public String toString(){
        return "("+fst+","+snd+")";
    }
}
```

All of the occurrences of the type parameters are highlighted, but notice that we only have the type parameters list at the class declaration itself; all other uses just use the already-in-existence types named R or S.

Generics with Classes

In order to use the Pair class, we need to supply actual types for each type parameter. This occurs when:

- we call the constructor
- we create a variable to hold one of our pairs

First, let's look at the constructor calls:

```
Pair<Integer,String> pintstr = new Pair<Integer,String>(4,"yo");

System.out.println("pintstr: "+pintstr);
pintstr.fst = 12;          // Java knew that putting an int here was okay.
int left = pintstr.fst;     // Java already knows there should be an Integer here.
String right = pintstr.snd; // Java also knows that there's a String here.
System.out.println("as parts: fst="+left+", snd="+right);
```

This particular usage decided to use Integer as the first type, and String as the second type.

We can use this one Pair class with any types we like except for primitives:

```
// they can be the same
Pair<Integer,Integer> pints = new Pair<Integer,Integer>(2,4);

// they can be arrays!
Pair<int[], short[]> parrs = new Pair<int[],short[]>(   new int[]{0,1,2},

// they can even be other Pair<..> types!
Pair<String,Pair<Integer,Boolean>> triplet =
  new Pair<String,Pair<Integer,Boolean>>("a", new Pair<Integer,Boolean>(5,true));

// minimal usage...
System.out.println(pints+"\n"+parrs+"\n"+triplet);
```

So far, we at least are able to use type parameters to make a class that operates over some particular type, and yet we get to choose that particular type for each unique usage of the class. Furthermore, Java can perform extra type checking on our code, and can help us avoid many castings!

Your Turn!

- Make a class definition that is generic; it should be named Box. A Box can hold one item of any type, but that type is decided by the class's type parameter.
- In a separate class named `TestGenerics` (we'll use the name later on), create a couple of Box items and store them to variables (also using generics). What are some interesting types of Boxes you can make? What about boxes containing boxes? Boxes of arrays of boxes?
- access your box's value. Then, update its value (assign a new value to it).
- Perhaps more importantly, try to mis-use your Box: put the wrong thing in it, mismatch the constructor type parameters with the variable's type's type parameters. What do Java's compilation errors look like?

Using the Type Parameters within the Class

Let's continue with the Pair class, and add getters and setters (even though its fields were declared public for convenience's sake above). Put the following methods into your Pair class:

```
public R getFst(){ return fst; }
public S getSnd(){ return snd;}
```

```
    public void setFst(R r) { fst = r;}
    public void setSnd(S s) { snd = s;}
```

Because this code is inside the class, the types R and S are in scope: we can use them like any other type in our code while inside the Pair class.

Notice that our getters have R and S as return types. We don't know exactly what R will be when we write this code, but we do know that whatever eventually gets used is the type that `fst` will have; we're returning the value of `fst`, so we'll also definitely have that same type.

Similarly, note how our setters accept parameters of R and S respectively. They know those are the exact types needed for `fst` and `snd`.

Your Turn!

- make the item in your Box class private. (sorry if this breaks your earlier testing code!)
- add the method `replaceItem` to your Box class. (it's a setter).
- add the method `unpack` to your Box class. (it's a getter).
- Try using your Box with these nicely-named getters and setters.

Generics and Methods

Type parameter lists may also be added to individual methods, so that they are only in scope during that method. This means that we can call these methods at different types, and Java can keep track of those specific types for each usage.

Consider the following method. It accepts a Pair, and creates a new pair by duplicating the first item in both spots.

```
public static <R,S> Pair<R,R> duplicateFirst(Pair<R,S> p){
      return new Pair<R,R>(p.fst,p.fst);
}
```

It's important to understand that the yellow-highlighted type parameters list is the only parameters list in this method. In the text `Pair<R,R>` and `Pair<R,S>`, we are merely using R and S as plain old types. The fact that they came from a type parameters list doesn't matter.

Using generic methods

We have to write somewhat funky syntax to use these generic methods. We still (usually) have to supply the actual types to be used. Given that our method is static, we can call it with ClassName.methodName, only now we have to wedge in the type parameters:

```
ClassName.<Type,Params,Here>methodName(regular,arguments,here)
```

Suppose we've been writing our code in the TestGenerics class as suggested above. We'd write:

```
Pair<Integer,String> pis = new Pair<Integer,String>(5,"hi");
Pair<Integer,Integer> pii = TestGenerics.<Integer,String>duplicateFirst(pis);
System.out.println(pii);
```

If we're lucky, Java can actually figure out what types we mean to use. Given that we have one parameter that exactly uses R and S, it seems reasonable that Java could use those as cues to instantiate the type parameters for us. Indeed, that is allowed:

```
    Pair<Integer,String> pis = new Pair<Integer,String>(5,"hi");
    // look ma, no explicitly instantiated type parameters!
    Pair<Integer,Integer> pii = TestGenerics.duplicateFirst(pis);
    System.out.println(pii);
```

We can't always rely on Java to know the best type, so we will often have to list them explicitly.

Why didn't we just use a class type parameter and use that type in our method? Because (1) we'd have to lock in that type once and for all, and couldn't call the same method at different types; and (2) the method is static, so we don't have an actual object from which to figure out what types were used during its instantiation.

If our method was not static, we could do the exact same with the object in front of the dot instead of the className:

```
    // create an object of the class...
    TestGenerics tg = new TestGenerics();
    // explicit type parameter instantiation
    Pair<Integer,Integer> pii = tg.<Integer,String>duplicateFirst(pis);
    // implicit type parameter instantiation
    Pair<Integer,Integer> pii2 = tg.duplicateFirst(pis);
```

Your Turn!

- In your testing class, write the generic method `repeatFst` (which will use your Pair class). It accepts an int `count` representing how many times to repeat the value of `fst`, also accepts a pair, and then returns an array that is `count` spots long, each filled with a copy of the pair's `fst` value.
- Write a static generic method, called `zip`, which accepts two arrays of two different types. It returns an array of Pairs, where the items from the two input arrays have been zipped together: the first Pair contains the first items of each array; the second Pair contains the second value from each array, and so on. Whichever array is shorter, that is how long the resulting array is. (if either array is null, just return null).

Quick Aside: Wildcards
We actually don't care what S is, to the point that we don't ever need to use it! We can get rid of S, and use a wildcard (represented with a question mark) to make this point clear:

```
    public static <R> Pair<R,R> duplicateFirst(Pair<R,?> p){
          return new Pair<R,R>(p.fst,p.fst);
    }
```

This code only has one type parameter in its type parameters list. When we get to the parameter `Pair<R,?>`, we're stating that whatever the second type of this Pair is, we don't care and will never be relying upon it anywhere else.

Case Study: Interfaces and Generics

Consider the following interface:

```
    public interface Changer<A,B>{
          public B change(A a);
    }
```

The interface has a type parameters list, but the method inside of it just uses that type parameter. If we wanted to implement this interface, we'd have to instantiate the type parameters in the "implements" portion of the class declaration:

```
public class IntBoxToIntPair implements Changer<Box<Integer>,Pair<Integer,Integer>> {
	public Pair<Integer,Integer> change(Box<Integer> b){
		return new Pair<Integer,Integer>(b.unpack(),b.unpack());
	}
}
```

We didn't have to use a type parameters list for our IntBoxToIntPair class, because we're immediately listing at what type we're implementing the Changer interface.

Here is how we might use our IntBoxToIntPair class:

```
	Box<Integer> myBox = new Box<Integer>(5);

	IntBoxToIntPair ibtip = new IntBoxToIntPair();
	Pair<Integer,Integer> plainChange = ibtip.change(myBox);
	System.out.println("ibtip got "+plainChange);
```

The construction of ibtip doesn't need any generics. Calling its particular implementation of change is like any interface, and doesn't need any type parameters (there's only one version of it for this class). The return type of ibtip's change is always Pair<Integer,Integer>, so that is the type of variable we can store the output to.

We can also implement a generic interface without committing to the final types, if our class itself is generic. Consider this more general version of converting a box to a pair:

```
	public class BoxToPair <R> implements Changer<Box<R>,Pair<R,R>> {
		public Pair<R,R> change(Box<R> b){
			return new Pair<R,R>(b.unpack(),b.unpack());
		}
	}
```

And here is how we use it, with all type parameter instantiations highlighted:

```
// create the box.
Box<Integer> myBox = new Box<Integer>(5);

// create instances of the class at a specific type that can change stuff for us.
BoxToPair<Integer> box2pair = new BoxToPair<Integer>();

// use the change method to perform the conversion.
Pair<Integer,Integer> pii = box2pair.change(myBox);

// show the results!
System.out.println("got: "+pii);

// bizarre one-liner version...
Pair<String,String> p2 =(new BoxToPair<String>()).change(new Box<String>("present"));
System.out.println("horrible one-liner returned: "+p2);
```

This kind of class does nothing other than implement a generic interface at one particular type(s) instantiation, and then is used as a sort of facilitator.

Your Turn!

- Create another class, named PairToString. It should implement `Changer<Pair<R,S>,String>`. Its `change` method can just return a String looking something like "a Pair of 5 and hello". Does your class need a type parameters list?
    - Continue with the previous usage code: create one of your PairToString objects, and use its change method to convert the Pair<Integer,Integer> (that our BoxToPair gave us) into a String.

Case Study: java.util.Collections' sort method

The java.util.Collections class has the following method:

```
static <T> void sort(List<T> list, Comparator<? super T> c).
```

The entire method has a type parameter list (`<T>`). Its first parameter is just any List<T>. Note that List<T> is an interface, so we'll actually have some object from a class that implements it, such as ArrayList<E> or LinkedList<E>. That last parameter is something a bit beyond how far we want to go with generics, but here's enough of a description that we can at least use this method.

Comparator<T> is an interface providing one method:

```
interface Comparator <T> {
        int compare(T o1, T o2);
}
```

Its job as an object is just to know how two things of type T could be compared, and return a negative number if they are already in ascending order (o1 < o2), return a zero if they are equivalent (o1.equals(o2) in some sense), or a positive number if they are out of ascending order (o1>o2).

We've already seen the wildcard (?). Here, its usage means that we don't need to ever use the specific type of things we're comparing, as long as it's some super-type of T.

The `super` keyword is being used to indicate that whatever is used for this comparator, it has to be some type that is no more specific than T. Suppose we had a list of poodles: the type of it is List<Poodle>. We could sort them using an implementation of `Comparator<Poodle>`, but we could also sort them with an implementation of `Comparator<Dog>` or `Comparator<Animal>` or `Comparator<Object>`. In order to allow us to use any of these, the `Collections.sort` method doesn't just accept a `Comparator<T>`. It instead is telling us that the List<T> can be sorted by any comparator for things that T can behave like. What can T behave like? Any of its ancestor classes. Thus, we're putting a *lower bound* on our generic type! `Comparator<? super T>` means that any comparator implementation that could be used on this type of list is allowed to be used. **Whew**!

So suppose we create our own class that implements Comparator<T> for some T.

```
import java.util.*;
public class CompareBoxes implements Comparator<Box<Integer>> {
        public int compare(Box<Integer> b1, Box<Integer> b2) {
                int b1val = b1.unpack();
                int b2val = b2.unpack();

                if (b1val < b2val) { return -1; }
```

```
            if (b1val == b2val) { return 0; }
            return 1;
        }
}
```

Now we can use this with the Collections.sort method. An instance of our CompareBoxes class serves as a witness/facilitator to the sort, telling it whether two items are relatively in order yet or not.

```
CompareBoxes cb = new CompareBoxes();
List<Box<Integer>> boxes = new ArrayList<Box<Integer>>();
boxes.add(new Box<Integer>(5));
boxes.add(new Box<Integer>(3));
boxes.add(new Box<Integer>(4));
System.out.println("before: "+boxes);
Collections.sort(boxes,cb);
System.out.println("after: "+boxes);
```

Your Turn! (Solutions below)

- Create a different class, `SortIntPairs`, that knows how to sort `Pair<Integer,Integer>` based on their first values. If the first values are equal, then look at the second value to determine the sortedness.
- Create an `ArrayList` filled with `Pair<Integer,Integer>` values. Create one of your `SortIntPairs` objects. Use both in a call to Collections.sort and print out your arraylist before and after the sort.
- (Super Hard!!!!) Create another class, SortPairs. It wants to sort pairs of anything, so it needs four type parameters: R and S, for the contents of the pairs; C1 and C2 (arbitrarily chosen names), which are types that are able to compare things of type R and S respectively. So we actually want this class signature:

  ```
  public class SortPairs
        <R,S,C1 extends Comparator<R>, C2 extends Comparator<S>>
       implements Comparator<Pair<R,S>> { … }
  ```

  what fields should a SortPairs object have? How will we implement the compare method?
- Create some ArrayList of Pairs of something else (perhaps pairs of boxes of Integers). Construct all the "witness" objects necessary in order to call Collections.sort on your List of pairs of things you know how to compare.

Solutions to the Last "Your Turn":

```
import java.util.*;
public class SortIntPairs implements Comparator<Pair<Integer,Integer>> {

  public int compare (Pair<Integer,Integer> p1, Pair<Integer,Integer> p2) {
    // doesn't return exactly -1, 0, 1; but comparisons don't have to! Just needs neg, zero, pos.
    int firsts = p1.fst - p2.fst;
    int seconds = p1.snd - p2.snd;
    if (firsts != 0){return firsts;}
    else            {return seconds;}
  }
}
```

```
List<Pair<Integer,Integer>> piis = new ArrayList<Pair<Integer,Integer>>();
piis.add(new Pair<Integer,Integer>(3,4));
piis.add(new Pair<Integer,Integer>(2,5));
piis.add(new Pair<Integer,Integer>(3,1));
System.out.println("piis before: "+piis);
Collections.sort(piis, new SortIntPairs());
System.out.println("piis after: "+piis);
```

```
import java.util.*;
public class SortPairs
       <R,S,C1 extends Comparator<R>, C2 extends Comparator<S>>
       implements Comparator<Pair<R,S>>
{

  public C1 c1;
  public C2 c2;
  public SortPairs(C1 c1, C2 c2){
    this.c1 = c1;
    this.c2 = c2;
  }

  public int compare (Pair<R,S> p1, Pair<R,S> p2) {
    int firsts = c1.compare(p1.fst, p2.fst);
    int seconds = c2.compare(p1.snd, p2.snd);
    if (firsts != 0){return firsts;}
    else            {return seconds;}
  }

}
```

```
CompareBoxes cb = new CompareBoxes();
SortPairs<Box<Integer>,Box<Integer>,CompareBoxes,CompareBoxes> sp
  = new SortPairs<Box<Integer>,Box<Integer>,CompareBoxes,CompareBoxes>(cb, cb);
List<Pair<Box<Integer>,Box<Integer>>> boxpairs = new ArrayList<Pair<Box<Integer>,Box<Integer>>>();
boxpairs.add(new Pair<Box<Integer>,Box<Integer>>(new Box<Integer>(3), new Box<Integer>(4)));
boxpairs.add(new Pair<Box<Integer>,Box<Integer>>(new Box<Integer>(2), new Box<Integer>(5)));
boxpairs.add(new Pair<Box<Integer>,Box<Integer>>(new Box<Integer>(3), new Box<Integer>(1)));
System.out.println("boxpairs before: "+boxpairs);
Collections.sort(boxpairs,sp);
System.out.println("boxpairs after: "+boxpairs);
```