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.

## 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:

```java
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:
```java
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:
```java
public class StringIntPair {
    public String fst;
```
```java
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:

```java
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:

```java
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:

```java
ArrayList xs = new ArrayList();
xs.add("hello");
xs.add(new Integer(3));
char c = ((String)xs.get(0)).charAt(((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:

```java
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:

```java
Pair<Integer,String> pintstr = new Pair<Integer,String>(new Integer(4),"yo");
```
System.out.println("pintstr: "+pintstr);
pintstr.fst = new Integer(12); // Java knew that putting an int here was okay.
Integer 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>(new Integer(2), new Integer(4));

// they can be arrays!
Pair<int[], short[]> parrs = new Pair<int[], short[]> (new int[]{0, 1, 2},
                                                        new short[]{3, 4, 5});

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!
1. 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.
2. 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?
3. Access your box’s value. Then, update its value (assign a new value to it).
4. 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:

```java
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 \texttt{fst} will have; we're returning the value of \texttt{fst}, so we'll also definitely have that same type.

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

Your Turn!
5. make the item in your \texttt{Box} class private. (sorry if this breaks your earlier testing code!)
6. add the method \texttt{replaceItem} to your \texttt{Box} class. (it’s a setter).
7. add the method \texttt{unpack} to your \texttt{Box} class. (it’s a getter).
8. Try using your \texttt{Box} with these nicely-named getters and setters.