Prepared by Linan Qiu <lq2137@columbia.edu>

Bounded Type

Now let's find more ways for us to be lazy in Java. This involves a little more generics, namely **Bounded Type** and **Wildcards**. You will likely have to use both in future assignments (and life!), so do make sure that you're familiar with these.

Let's start with bounded type.

Let's say you have a class called `Square`

```java
//Square.java

public class Square implements Comparable<Square> {

  private int side;

  public Square(int side) {
    this.side = side;
  }

  @Override
  public int compareTo(Square o) {
    return side - o.side;
  }

  public String toString() {
    return "I'm a square of side " + side;
  }

  public static void main(String[] args) {
    Square square1 = new Square(1);
    System.out.println(square1); // I'm a square of side 1
  }
}
```

If you want to compare squares, you may want make a class called `Robot` that contains a static method to compare squares. We make the method static because there's really no need for us to instantiate a `Robot`, then use a separate robot object to compare squares. It should be like a helper function, and that's exactly what static methods shoudl be used for.

```java
// Robot.java

public class Robot {
  public static void sayBigger(Square a, Square b) {
    if (a.compareTo(b) > 0) {
      System.out.println(a);
    } else if (a.compareTo(b) < 0) {
      System.out.println(b);
    } else {
      System.out.println("Neither");
    }
  }
}
```

Cool! So you may use the static method as such:

```java
// in some main method far far away...

Square square1 = new Square(1);
Square square5 = new Square(5);
Robot.sayBigger(square1, square5); // I'm a square of side 5
```

This is, however, terribly silly. Let's say we wanted to compare the `Rectangle` class from your homework 1. We'd have to make a separate static method. Or say we simply wanted to a `Shape` from Java's GUI framework. We'd still have to create yet another static method. This is terribly cumbersome.

Generic Methods

We can make the static method generic. That is, we can write

```java
// Robot.java

public class Robot {
  public static <T> void sayBigger(T a, T b) {
    if (a.compareTo(b) > 0) {
      System.out.println(a);
    } else if (a.compareTo(b) < 0) {
      System.out.println(b);
    } else {
      System.out.println("Neither");
    }
  }
}
```

Note that this code won't compile, because it is wrong. I'm ignoring that error for now. We'll get to that in the next section

Notice that I did not declare `<T>` after the class. Instead, I defined it at the method level. This means that the `T` variable only exists for this specific method. This means that the class itself (`Robot`) is not generic and only the method is. I can define several methods like this:

```
// Robot.java

public class Robot {
  public static <T> void sayBigger(T a, T b) {
    if (a.compareTo(b) > 0) {
      System.out.println(a);
    } else if (a.compareTo(b) < 0) {
      System.out.println(b);
    } else {
      System.out.println("Neither");
    }
  }

  public static <AnyType> AnyType getBigger(AnyType a, AnyType b) {
    if (a.compareTo(b) > 0) {
      return a;
    } else if (a.compareTo(b) < 0) {
      return b;
    } else {
      return null;
    }
  }
}
```

Here, `AnyType` essentially works the same way as `T`, but is a completely separate variable. To illustrate this, in a separate main method, I can do this:

```
// in a main method far far away...
Rectangle rectangle1 = new Rectangle(1, 5);
Rectangle rectangle2 = new Rectangle(2, 5);
Square square1 = new Square(1);
Square square5 = new Square(5);

Robot.sayBigger(rectangle1, rectangle2);
Square biggerSquare = Robot.getBigger(square1, square5);
```

In fact, you can change both to `T`:

```java
// Robot.java

public class Robot {
  public static <T> void sayBigger(T a, T b) {
    if (a.compareTo(b) > 0) {
      System.out.println(a);
    } else if (a.compareTo(b) < 0) {
      System.out.println(b);
    } else {
      System.out.println("Neither");
    }
  }

  public static <T> T getBigger(T a, T b) {
    if (a.compareTo(b) > 0) {
      return a;
    } else if (a.compareTo(b) < 0) {
      return b;
    } else {
      return null;
    }
  }
}
```

And there'd be no change in functionality.

However, this code won't compile. Why? Because we are making a huge assumption: We are hoping that whatever `T` or `AnyType` is, it'd come with the `compareTo` method. That's a bad assumption. Other than the methods that all `Objects` have (since all objects extend `Object`, the mother of all classes), we can't assume any other method on this `T` or `AnyType`.

How can we make the code work then?

Bounded Type

The nerds among you (which I hope is everyone) would have realized a pattern: both `Rectangle` and `Square` (and `Shape` and anything that you'd want to compare) implements the `Comparable` interface. In fact, they all share the class signature of `Something implements Comparable<Something>` where `Something` can be `Square`, `Rectangle`, or something else. Let's exploit this. We can tell Java: *for this method, can you please only accept classes that are comparable to itself?* Omit the "please" if you want.

This is how we say that in code:

```java
// Robot.java

public class Robot {
  public static <T extends Comparable<T>> void sayBigger(T a, T b) {
    if (a.compareTo(b) > 0) {
      System.out.println(a);
    } else if (a.compareTo(b) < 0) {
      System.out.println(b);
    } else {
      System.out.println("Neither");
    }
  }

  public static <T extends Comparable<T>> T getBigger(T a, T b) {
    if (a.compareTo(b) > 0) {
      return a;
    } else if (a.compareTo(b) < 0) {
      return b;
    } else {
      return null;
    }
  }
}
```

By writing `<T extends Comparable<T>>` to specify `<T>` instead of just `<T>` itself, we are saying that we only accept classes that implement Comparable of itself. We are **adding restrictions to `<T>`**. Hence the name bounded generics.

Now why do we use the word `extend` instead of `implements` since `Square implements Comparable<Square>` (since Comparable is an interface)? Well in Java's official Generics Tutorial, the folks at Oracle says:

> To declare a bounded type parameter, list the type parameter's name, followed by the extends keyword, followed by its upper bound, which in this example is Number. Note that, in this context, extends is used in a general sense to mean either "extends" (as in classes) or "implements" (as in interfaces)

Essentially, they wanted one word, and they settled for `extends`, confusing the hell out of all of us.

Then now, we'd be able to do this without any trouble at all:

```java
// in a main method far far away...
Rectangle rectangle1 = new Rectangle(1, 5);
Rectangle rectangle2 = new Rectangle(2, 5);
```

```
Square square1 = new Square(1);
Square square5 = new Square(5);

Robot.sayBigger(rectangle1, rectangle2);
Square biggerSquare = Robot.getBigger(square1, square5);
```

Again You Already Know This

You probably have used this already if you have ever used `Collections.sort()`. If you don't know what that is, that's totally cool.

Let's say you have an `ArrayList<Integer>` and you want to sort it:

```
// in a main method far far away...

ArrayList<Integer> list = new ArrayList<>(); // remember the shorthand syntax
// for generics from last week?

// generate a list of 100 random integers
Random random = new Random();
for (int i = 0; i < 100; i++) {
  list.add(random.nextInt());
}

Collections.sort(list); // sort the list

System.out.println(list); // sorted list
```

How does `Collections.sort` work? Well it sorts the list. However, how does it guarantee that whatever's passed in can be sorted? We can look at the method signature of `Collections.sort`.

```
// Collections.sort method signature

public static <T extends Comparable<? super T>> void sort(List<T> list)
```

Ignore the `? super T` part. We'll get to that in the next chapter. However, you can see that it is asking for a `T` that implements Comparable of itself. However, instead of taking in a `T` as an argument, it specifies a `List<T>`, a list containing `T`s. This makes sense, since it will sort a list of comparable items. Now since `ArrayList` itself implements List (`ArrayList<T> implements List<T>` Java API for ArrayList), `ArrayList<Integer>` is a valid argument.

Now you know bounded type! You can now be even lazier. *smirk*