Prepared by Linan Qiu <lq2137@columbia.edu>

Wildcards

Wildcards are essentially an even lazier species in generics. It comes in useful when you really don't give a hoot. Here's an example.

Refresher on Vanilla Generics

Let's say you're trying to print out an entire list of items:

```
// Test.java

public class Test {
  public static void main(String[] args) {
    List<Integer> list = new ArrayList<>();
    for(int i = 0 ; i < 100; i++) {
      list.add(i);
    }
    printList(list); // this line results in an error
  }

  public static void printList(List<Object> list) {
    for (Object item : list) {
      System.out.println(item);
    }
  }
}
```

Now this won't run. Java will scream at you, specifically at the line where you go `printList(list)`. The error is

> The method `printList(List<Object>)` in the type `Test` is not applicable for the arguments (`List<Integer>`)

In other words, it is saying that `Integer` isn't the same as `Object`, hence the printing won't work. That's cool. We know generics. We can always change the `printList` method to:

```
// in Test.java

public static <T> void printList(List<T> list) {
```

```
  for (T item : list) {
    System.out.println(item);
  }
}
```

Indeed, it works. However, something feels a little off. We just wrote 3 `T`s, but we didn't really do anything with it. The folks at Oracle realized this before we just did, and they came up with wildcards.

Wildcards

The method above can be rewritten using wildcards:

```
// in Test.java

public static void printList(List<?> list) {
  for (Object item : list) {
    System.out.println(item);
  }
}
```

In this case, we are saying: *I don't really care what goes inside the list, so long as the list is well a list.* Notice that within the for loop, we have changed `T` to `Object`. This is for two reasons:

1. `T` no longer exists as a variable
2. More importantly, since we used `?`, we are accepting any class. Since all classes extend `Object`, we have to use Object as the catch-all. Anything more specific will give an error.

Difference from Type Parameters

Restrictions on Type Relations

The `<T>` and `<AnyType>` we have been using are called *Type Parameters*. They parameterize the types for methods / classes. Now that we have introduced wildcards, the natural question is: when do we use which?

Remember I mentioned that wildcards are a more *I-don't-really-care* version of generics? Well the difference is exactly that.

Let's say we have a method like this:

```java
// in Test.java

public static <T extends Number> void copy(List<T> list1, List<T> list2) {
  for(T i : list1) {
    list2.add(i);
  }
}
```

It copies every element from `list1` to `list2`. It can be used in this way:

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

List<Integer> listIntA = new LinkedList<>();
List<Integer> listIntB = new LinkedList<>();
for (int i = 0; i < 100; i++) {
  listIntA.add(i);
  listIntB.add(100 - i);
}
copy(listIntA, listIntB);
System.out.println(listIntB);
// 100 99 98 ... 1 0 0 1 ... 98 99 100

List<Double> listDoubleA = new LinkedList<>();
List<Double> listDoubleB = new LinkedList<>();
for (int i = 0; i < 100; i++) {
  listDoubleA.add((double) i);
  listDoubleB.add((double) (100 - i));
}
copy(listDoubleA, listDoubleB);
System.out.println(listDoubleB);
// 100.0 99.0 98.0 ... 1.0 0.0 0.0 1.0 ... 98.0 99.0 100.0

copy(listDoubleA, listIntA); // compiler error
```

Recall that `Integer` and `Double` both extend `Number` in the Java API. The last line will give us a compiler error since the method requires that `T` be present in both `list1` and `list2`, and that `T` be the same `T` for both. Hence we can't have one `List<Integer>` and one `List<Double>`. Essentially, the method that you have defined ensures that both `list1` and `list2` will have the same type.

However, if we write the method using wildcards, we allow that:

```java
// Test.java

public static void copy(List<? extends Number> list1, List<? extends Number> list2) {
```

```
  for(Object i : list1) {
    list2.add(i); // this line will fail compilation as well
  }
}
```

In this case, we can pass a `List<Integer>` as `list1` and a `List<Double>` as `list2` and it will still be allowed. This is because each `?` means a different (or possibly same) class. There is no resriction that they refer to the same class. Then, this method wouldn't make sense at all. Moving elements from `list1` to `list2` would not be safe (since moving Integers to a list of Doubles would give you a runtime error).

Method Parameterization

The second difference is that wildcards can't be used directly to parameterize methods. Here's what I mean

```
// Test.java

public static void say(? something) {
  System.out.println(something)
}
```

This would not be possible. Instead, `?` can only be used to parameterize classes, such as `ArrayList<?>`, `List<?>`, `Collections<?>` etc. It cannot be used directly as a type parameter.

Upper, Unbounded, and Lower Bounds

Now you're ready to understand that `? super T` mentioned in the previous set of notes.

Upper Bounded Wildcards

Let's say we have a method that prints a list of numbers:

```
// Test.java

public static void sayNumbers(List<? extends Number> list) {
  for (Number item : list) {
    System.out.println(item);
  }
}
```

It makes sense for us to want it to work on `Number` ,`Integer` and `Float`. In other words, we want it to work on `Number` and all its subclasses. Hence we used an upper bound wildcard `? extends Number` telling Java to accept any list containing `Number`s or its subclasses.

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

List<Integer> list1 = new LinkedList<>();
// populate the list ...

List<Double> list2 = new LinkedList<>();
// populate the list too ...

sayNumbers(list1);
sayNumbers(list2);
// both works
```

That's essentially upper bounded wildcards.

Unbounded Wildcards

Unbounded wildcards is simply when anything goes.

```
// in Test.java

public static void printList(List<?> list) {
  for (Object item : list) {
    System.out.println(item);
  }
}
```

The first example we ever used in this chapter uses it. Nothing much to be said about this other than the fact that if you ever have to use elements in the list, you have to reference them using `Object`. Since you chose to be lazy and accepted everything, you are stuck with `Object`, the mother of all classes.

Lower Bounded Wildcards

Upper bounded wildcards restricts the type to a class and all its subclasses. Similarly, lower bounded wildcards restricts the type and all its super classes. Why would you ever want that?

(The following example is taken from the Official Java Tutorial). Say you want to write a method that puts `Integer` objects into a list. To maximize flexibility,

you would like the method to work on `List<Integer>`, `List<Number>`, and `List<Object>` — anything that can hold Integer values.

To write the method that works on lists of Integer and the supertypes of Integer, such as Integer, Number, and Object, you would specify `List<? super Integer>`. The term `List<Integer>` is more restrictive than `List<? super Integer>` because the former matches a list of type `Integer` only, whereas the latter matches a list of any type that is a supertype of `Integer`.

```
// in Test.java

public static void addNumbers(List<? super Integer> list) {
  for (int i = 1; i <= 10; i++) {
    list.add(i);
  }
}
```

Going back to our `Collections.sort` method signature,

```
// Collections.sort method signature

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

we are essentially saying that `T` has to implement comparable of itself or its superclass. Still confused? So was I when I first read it. So here's an example to show you what this all means. Let's say I have a `Pokemon` class. I want to be able to compare them against each other:

```
// Pokemon.java

public class Pokemon implements Comparable<Pokemon> {

  public int level;

  public Pokemon(int level) {
    this.level = level;
  }

  public int compareTo(Pokemon o) {
    return level - o.level;
  }
}
```

That's cool. However, being the weirdo that I am, I am also interested in `ShinyPokemon`s as well. They're a type of `Pokemon`, so I'll go ahead and extend

`Pokemon`. (If you don't know what they are, google it. They're like normal Pokemon but way more awesome).

```
// ShinyPokemon.java

public class ShinyPokemon extends Pokemon {

  public int level;

  public ShinyPokemon(int level) {
    super(level);
  }

  public void shine() {
    System.out.println("Bling bling");
  }

  @Override
  public int compareTo(Pokemon o) {
    return level - o.level;
  }
}
```

Now if I have a list of Pokemon AND ShinyPokemon, `Collections.sort` should be able to compare them against each other right? However, note that `ShinyPokemon` doesn't actually directly implement the `Comparable` interface. It only extends `Pokemon` which implements the `Comparable<Pokemon>` interface. There's no `ShinyPokemon implements Comparable<ShinyPokemon>`. In fact, there won't be a `Comparable<ShinyPokemon>`. Then, we'd want to tell Java to allow comparing of `ShinyPokemon` and `Pokemon`. Or more generally, allow any class whose super class implements `Comparable<Pokemon>`. This is exactly what our lower bound wildcard says.

Hence, the `<T extends Comparable<? super T>>` says to allow any type `T` that implements `Comparable<Something>` where `Something` is a superclass of `T`. In our instance, `T` will be `Pokemon` and since `ShinyPokemon` is a superclass of `Pokemon`, this signature matches.

Now you see how upper, unbounded, and lower bound wildcards pretty much catch them all. (Very bad pun intended)