**What are java Generics and wildcards?**

Java Generic's wildcards is a mechanism in Java Generics aimed at making it possible to cast a collection of a certain class, e.g A, to a collection of a subclass or superclass of A. This text explains how.

Here is a list of the topics covered:

**The Basic Generic Collection Assignment Problem**

Imagine you have the following class hierarchy:

public class A { }

public class B extends A { }

public class C extends A { }

The classes B and C both inherit from A.

Then look at these two List variables:

List<A> listA = new ArrayList<A>();

List<B> listB = new ArrayList<B>();

Can you set listA to point to listB ? or set listB to point to listA? In other words, are these assignments valid:

listA = listB;

listB = listA;

The answer is no in both cases. Here is why:

In listA you can insert objects that are either instances of A, or subclasses of A (B and C). If you could do this:

List<B> listB = listA;

then you could risk that listA contains non-B objects. When you then try to take objects out of listB you could risk to get non-B objects out (e.g. an A or a C). That breaks the contract of the listB variable declaration.

Assigning listB to listA also poses a problem. This assignment, more specifically:

listA = listB;

If you could make this assignment, it would be possible to insert A and C instances into the List<B> pointed to by listB. You could do that via the listA reference, which is declared to be of List<A>. Thus you could insert non-B objects into a list declared to hold B (or B subclass) instances.

**When are Such Assignments Needed?**

The need for making assignments of the type shown earlier in this text arises when creating reusable methods that operate on collections of a specific type.

Imagine you have a method that processes the elements of a List, e.g. print out all elements in the List. Here is how such a method could look:

public void processElements(List<A> elements){

for(A o : elements){

System.out.println(o.getValue());

}

}

This method iterates a list of A instances, and calls the getValue() method (imagine that the class A has a method called getValue()).

As we have already seen earlier in this text, you can not call this method with a List<B> or a List<C> typed variable as parameter.

**Generic Wildcards**

The generic wildcard operator is a solution to the problem explained above. The generic wildcards target two primary needs:

* Reading from a generic collection
* Inserting into a generic collection

There are three ways to define a collection (variable) using generic wildcards. These are:

List<?> listUknown = new ArrayList<A>();

List<? extends A> listUknown = new ArrayList<A>();

List<? super A> listUknown = new ArrayList<A>();

The following sections explain what these wildcards mean.

**The Unknown Wildcard**

List<?> means a list typed to an unknown type. This could be a List<A>, a List<B>, a List<String> etc.

Since the you do not know what type the List is typed to, you can only read from the collection, and you can only treat the objects read as being Object instances. Here is an example:

public void processElements(**List<?> elements**){

for(Object o : elements){

System.out.println(o);

}

}

The processElements() method can now be called with any generic List as parameter. For instance a List<A>, a List<B>, List<C>, a List<String> etc. Here is a valid example:

List<A> listA = new ArrayList<A>();

processElements(listA);

**The extends Wildcard Boundary**

List<? extends A> means a List of objects that are instances of the class A, or subclasses of A (e.g. B and C).

When you know that the instances in the collection are of instances of A or subclasses of A, it is safe to read the instances of the collection and cast them to A instances. Here is an example:

public void processElements(**List<? extends A> elements**){

for(A a : elements){

System.out.println(a.getValue());

}

}

You can now call the processElements() method with either a List<A>, List<B> or List<C>. Hence, all of these examples are valid:

List<A> listA = new ArrayList<A>();

processElements(listA);

List<B> listB = new ArrayList<B>();

processElements(listB);

List<C> listC = new ArrayList<C>();

processElements(listC);

The processElements() method still cannot insert elements into the list, because you don't know if the list passed as parameter is typed to the class A, B or C.

**The super Wildcard Boundary**

List<? super A> means that the list is typed to either the A class, or a superclass of A.

When you know that the list is typed to either A, or a superclass of A, it is safe to insert instances of A or subclasses of A (e.g. B or C) into the list. Here is an example:

public static void insertElements(List<? super A> list){

list.add(new A());

list.add(new B());

list.add(new C());

}

All of the elements inserted here are either A instances, or instances of A's superclass. Since both B and C extend A, if A had a superclass, B and C would also be instances of that superclass.

You can now call insertElements() with either a List<A>, or a List typed to a superclass of A. Thus, this example is now valid:

List<A> listA = new ArrayList<A>();

insertElements(listA);

List<Object> listObject = new ArrayList<Object>();

insertElements(listObject);

The insertElements() method cannot read from the list though, except if it casts the read objects to Object. The elements already present in the list when insertElements() is called could be of any type that is either an A or superclass of A, but it is not possible to know exactly which class it is. However, since any class eventually subclass Object you can read objects from the list if you cast them to Object. Thus, this is valid:

Object object = list.get(0);

But this is not valid:

A object = list.get(0);