Wild Cards

The Wild Card

- Given G is a generic type, G<Object> is not the super type for all G<E>. G<Object> is just another type... it is certainly not a super type.
- You can't expect to be able to assign a reference for a G<E> type within a variable of G<Object> type.

Example:

```
MyStack<Integer> stackInteger = new MyStack<Integer>();
MyStack<Number> stack = stackInteger;
```

This code (although Integer extends Number) is illegal!

The Wild Card

- Given G is a generic type, G<?> is (kind of) the super type for any G<E>.
- Based on that, the last sample can be fixed:

```
MyStack<Integer> stackInteger = new MyStack<Integer>();
MyStack<?> stackNumber = stackInteger;
```

Sample

```
import java.util.*;

public class GenericSubtypeWildCard
{
        public static void main(String[] args)
        {
                MyStack<Date> stackDate = new MyStack<Date>();
                stackDate.push(new Date());
                MyStack<?> stackOb = stackDate;
                System.out.println(stackOb);
        }
}
```

Sample

```
class MyStack<E>
{
        ArrayList<E> array = new ArrayList<E>();

        void push(E element)
        {
                array.add(element);
        }
```

Sample

```
        E pop()
        {
                E reply = array.get(array.size()-1);
                array.remove(array.size()-1);
                return reply;
        }

        public String toString()
        {
                return array.toString();
        }
}
```

Object

- Any of the elements that belong to a generic type is an object that as any other object extends the class Object. Assigning the reference of an element that belongs to generic type collection (or another generic type...) into a variable of type Object shouldn't be a problem. It would always be legal.
- The opposite won't always work. It is impossible to assume that an object can be added as a new element to a given generic collection (unless the added object's type is exactly the same type the generic collection's elements have).

Object

- When dealing with a generic type that we don't know its type, trying to send it a new element would work if (and only if) the new element type is the same type of the generic unknown type.

  Example:

  ```
  Collection<?> c = new Vector<Date>();
  c.add(new Integer());
  ```

  The second line can't work. It won't compile. Integer is not a subtype of Date.

Object

- When dealing with getting a value returned from a generic type collection (or another..) we can assign the reference we get into any variable of type Object. All types extend Object so there shouldn't be any problem doing so.

  Example:

  ```
  Vector<Integer> vec = new Vector<Integer>();
  vec.addElement(new Integer(32));
  Object ob = vec.get(1);
  ```

  This will work. Object is the super class of all types.

Bounded Wild Card

- Given G is a generic type, G<?> is the super type for all G<E> and G<? extends M> is the super type for all G<E> as long as E is of type M or of a type that extends M.
- G<? extends M> is called a bounded wild card.
- The following sample presents the necessity of the bounded wild card. Try to run it and see the problem we have with calculateTotal method. Note the error messages we get.

Bounded Wild Card

```
import java.util.*;

public class BoundedWildCardSample
{
        public static double calculateTotal(Vector<Shape> vec)
        {
                double sum = 0;
                Iterator<Shape> iterator = vec.iterator();
                while(iterator.hasNext())
                {
                        sum += iterator.next().area();
                }
                return sum;
        }
```

Bounded Wild Card

```
    public static void main(String args[])
    {
        double total = 0;
        Vector<Circle> vector = new Vector<Circle>();
        vector.add(new Circle(4));
        vector.add(new Circle(8));
        vector.add(new Circle(2));
        total = calculateTotal(vector);
        System.out.println("total="+total);
    }
}
```

Bounded Wild Card

```
abstract class Shape
{
        abstract double area();
}
class Rectangle extends Shape
{
        double width, height;
        Rectangle(double wVal, double hVal)
        {
              width = wVal;
              height = hVal;
        }
        public double area()
        {
              return width*height;
        }
}
```

Bounded Wild Card

```
class Circle extends Shape
{
        double radius;
        Circle(double rad)
        {
            radius = rad;
        }
        public double area()
        {
            return 3.14*radius*radius;
        }
}
```

Bounded Wild Card

- In order to fix the last sample we need to use a bounded wild card and fix the calcualteTotal method replacing <Shape> with <? extends Shape>.
- The <? extends Shape> is a bounded wild card, which means that it fits for any generic that its elements' type is either Shape or another type that extends Shape.

Bounded Wild Card

- The new fixed version of the last sample includes the following small fix for the generics type the calculateTotal method uses. Instead of <Shape> we can write <? extends Shape>:

```
public static double calculateTotal(Vector<? extends Shape> vec)
{
        double sum = 0;
        Iterator<? extends Shape> iterator = vec.iterator();
        while(iterator.hasNext())
        {
                sum += iterator.next().area();
        }
        return sum;
}
```