Principles of Software Development

V. Adamchik

Lecture 6

CSCI 201

USC

Bounded wildcards

Type Erasure

Terminology

Term	Example
Parameterized type	List<String>
Generic type	List<T>
Formal type parameter	T
Unbounded type parameter	List<T>
Unbounded wildcard type	List<?>
Upper-Bounded type parameter	List<T extends Number>
Lower-Bounded type parameter	List<T super Integer>
Upper-Bounded wildcard type	List<? extends Number>
Lower-Bounded wildcard type	List<? super Integer>
Generic method	public <T> List<T> asList(T[] a)

Generics with Upper Bounds

Upper bound is a boundary that includes the specified class or any which it extends.

```
<T extends superClass>
```

Java Generics supports multiple bounds

```
<T extends superClassName & Interface>
```

In lecture 2 we introduced a *generic method*

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

Upper Bounded Wildcards

The question mark (?), represents the *wildcard*, stands for unknown type in generics. You can use an upper bounded wildcard to relax the restrictions on a variable.

For example, List<Number> is more restrictive than List<? extends Number>, the latter matches a list of type Number or any of its subclasses (Integer, Double).

```
public void demo(List<? extends  Number> list)
```

You cannot ADD to the list

Because you can call this method of type List<Double>, therefore adding Integer would be a mistake.

Code Example 1

```
public class Demo {
  public static void main(String[] args) {
      List<Integer> li = Arrays.asList(5, 6, 7);
      System.out.println("sum = " + sum (li));
       List<Double> ld = Arrays.asList(1.2, 3.8, 8.2);
       System.out.println("sum = " + sum (ld));
  }
  public static double sum (List<? extends Number> list) {
        double s = 0.0;
        for (Number n : list) s += n.doubleValue();
        return s;
  }
}
```

Exercise 1

Why doesn't this code compile? Fix the print() method.

```
public class Demo {
  public static void main(String[] args) {
      List<Integer> li = Arrays.asList(5, 6, 7);
      print(li);
      List<Double> ld = Arrays.asList(1.2, 3.8, 8.2);
      print(ld);
}
 public static void print(List<Number> list){
    for (Object elem : list)
        System.out.print(elem + " ");
    System.out.println();
  }}
```

Exercise 2

Which of these declarations will cause a compile-time error?

1. `List list1 = new ArrayList<?>();`
2. `List list2 = new ArrayList<List<?>>();`

A. 1

B. 2

C. both

D. none

Generics with Lower Bounds

Lower bound is a boundary that includes the specified class or any super class to it.

`<T super childClass>`

Note a difference

1. `<T extends Number>`
2. `<T super Number>`

The first line works for Number and any its subclasses (Integer and Double).

The second line works only for Number and Object (for all superclases)

Mostly we use lower bounds with a wildcard: `List<? super Integer>`

This method will work for List<Integer>, List<Number>, List<Object>

Code Example 2

```
public class Demo {
  public static void main(String[] args) {
      List<Integer> li = new ArrayList<Integer>();
      sum(li);
      List<Number> ld = new ArrayList<Number>();
      sum(ld);
      List<Object> lo = new ArrayList<Object>();
      sum(lo);
}
 public static void  sum2(List<? super Integer> list) {
       for(int i = 1; i < 5; i++) list.add(i);
       System.out.println(list);
  }}
```

When to Use Bounded Wildcards

Use <? extends > when you only GET objects.

Use <? super > when you PUT values into a structure.

This is known as Producer Extends and Consumer Super.

```
public static <T> void copy (
                                    List < ? extends T> src,
                                    List < ? super T> dest)
{
    for( int k = 0; k < src.size(); k++)
        dest.set(k, src.get(k));
}
```

Type erasure

The compiler applies a process called *type erasure* on generics at compile time.

Compiler javac removes all type parameters and replaces it with their bounds or with Object if the type parameter is unbounded. We will see several examples in the next slides.

Erasure is not specific to Java, nor to generics. This tool translates the high-level language code into a lower level.

For example, JVM has instructions for moving integers (iload, istore) and doing arithmetics (iadd, imul). But there are no instructions for chars, or booleans – they are erased to ints by the compiler.
See https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-6.html for the list of instructions.

Many other features of Java are erased during compilation.

Type erasure

Generics are type-checked at compile time, but then a generic type like List<String> is erased to List when generating bytecode, and type variables such as <T extends Object> are erased to the erasure of their bound.

Consider method type erasure for unbound type:

```
public <T> List<T> genericMethod(List<T> lst)
```

This will become

```
public List genericMethod(List lst)
```

Class type erasure

```
public class Uba<T> {
    private T[] data;
…
}
```

Upon compilation, the compiler replaces the unbound type parameter T with Object.

```
public class Uba {
    private Object[] data;
…
}
```

Bounded Type

```
public <T extends Comparable > void genericMethod(T lst)
```

will become

```
public void genericMethod(Comparable lst)
```

Class type erasure:

```
public class Uba <T extends Comparable<T>> {
    private T[] data;
    public Uba (int cap) {
        data = (T[]) Object[cap];
…
}
```

```
public class Uba {
private Comparable[] data;
public Uba (int cap) {
data = (Comparable[]) Object[cap];
…
}
```

Exercise 2

If you look at Java's Collections class, you may see

```
static <T extends Object & Comparable<? super T>>  T
                                        max(Collection<? extends T> coll)
```

Why T extends Object?

What if we remove Object?

Solution

Under erasure (note, we removed Object)

```
static <T extends Comparable<? super T>>  T
                                     max(Collection<? extends T> coll)
```

will become

Exercise 3

Will the following class compile? If not, why?

```
public class Example {
        public void print(Set<String> strSet) { }
        public void print(Set<Integer> intSet) { }
}
```

Exercise 4

Will the following code compile?

Will this code run?

```
1. List<String> strings = new ArrayList<String>();
2. List objects = strings;
3. objects.add(new Object());
4. Object tmp1 = strings.get(0);
5. String tmp2 = strings.get(0);
```

Example

```
class Demo<T> {
  private T data;
  public void set(T obj) {
      data = obj;
}}
```

```
class MyDemo extends Demo<Integer> {
    public void set (Integer data) {
        super.set(data);
    }
}
```

After type erasure:

In Demo: public void set(

In MyDemo: public void set (

Exercise 5

What will be the output of the following program?

```
public class Demo {
    public static void main(String args[]) {
        Generics<double> obj = new Generics<double>(12.0);
        obj.print();
}}
class Generics<T> {
    T obj;
    Generics (T obj { this.obj = obj;}
     void print() {
           System.out.println(obj);
      }
}
```