

Java Generics

Concepts

- Generalizing Collection Classes
- Using Generics with other Java 1.5 Features
- Integration of Generics with Previous Releases
- User built generic classes

What Are Generics?

- Generics abstract over Types
- Classes, Interfaces and Methods can be Parameterized by Types
- Generics provide increased readability and type safety

Java Generic Programming

- Java has class Object
 - Supertype of all object types
 - This allows "subtype polymorphism"
 - Can apply operation on class T to any subclass $S \leq T$
- Java 1.0 - 1.4 do not have templates
 - No parametric polymorphism
 - Many consider this the biggest deficiency of Java
- Java type system does not let you cheat
 - Can cast from supertype to subtype
 - Cast is checked at run time

Example generic construct: Stack

- Stacks possible for any type of object
 - For any type `t`, can have type `stack_of_t`
 - Operations `push`, `pop` work for any type
- In `C++`, would write generic stack class

```
template <type t> class Stack {  
    private: t data; Stack<t> * next;  
    public: void  push (t* x) { ... }  
            t* pop  (    ) { ... }  
};
```

- What can we do in Java?

Simple Example Using Interfaces

```
interface List<E> {  
    void add(E x);  
    Iterator<E> iterator();  
}  
  
interface Iterator<E> {  
    E next();  
    boolean hasNext();  
}
```

Java 1.0

```
class Stack {  
    void push(Object o) { ... }  
    Object pop() { ... }  
    ...}
```

```
String s = "Hello";  
Stack st = new Stack();  
...  
st.push(s);  
...  
s = (String) st.pop();
```

vs Generics

```
class Stack<A> {  
    void push(A a) { ... }  
    A pop() { ... }  
    ...}
```

```
String s = "Hello";  
Stack<String> st =  
    new Stack<String>();  
st.push(s);  
...  
s = st.pop();
```

How to Use Generics

```
List<Integer> xs = new LinkedList<Integer>();  
xs.add(new Integer(0));  
Integer x = xs.iterator.next();
```

Compare with

```
List xs = new LinkedList();  
xs.add(new Integer(0));  
Integer x = (Integer)xs.iterator.next();
```

Collection Class Example

```
HashMap<Integer, Double> hm =  
    new HashMap<Integer, Double> ();
```

```
// Note Auto-boxing from 1.5
```

```
hm.put (1,2.0);
```

```
double coeff = hm.get(1);
```

List Usage: Without Generics

```
List ys = new LinkedList();  
ys.add("zero");  
List yss;  
yss = new LinkedList();  
yss.add(ys);  
String y = (String)  
((List)yss.iterator().next()).iterator().next();
```

// Evil run-time error

```
Integer z = (Integer)ys.iterator().next();
```

List Usage: With Generics

```
List<String> ys = new LinkedList<String>();  
ys.add("zero");  
List<List<String>> yss;  
yss = new LinkedList<List<String>>();  
yss.add(ys);  
String y = yss.iterator().next().iterator().next();
```

// Compile-time error - much better!

```
Integer z = ys.iterator().next();
```

List Implementation w/o Generics

```
class LinkedList implements List {  
    protected class Node {  
        Object elt;  
        Node next;  
        Node(Object elt){elt = e; next = null;}  
    }  
    protected Node h, t;  
    public LinkedList() {h = new Node(null); t = h;}  
    public void add(Object elt){  
        t.next = new Node(elt);  
        t = t.next;  
    }  
}
```

List Implementation With Generics

```
class LinkedList<E> implements List<E>
{
    protected class Node {
        E elt;
        Node next;
        Node(E elt){elt = e; next = null;} }
    protected Node h, t;
    public LinkedList() {h = new Node(null); t = h;}
    public void add(E elt){
        t.next = new Node(elt);
        t = t.next;
    }
    // ...
}
```

Recall the Iterator Interface

```
class LinkedList<E> implements List<E>
```

```
- // ...
```

```
public Iterator<E> iterator(){
```

```
    return new Iterator<E>(){
```

```
        protected Node p = h.next;
```

```
        public boolean hasNext(){return p != null;}
```

```
        public E next(){
```

```
            E e = p.el;
```

```
            p = p.next;
```

```
            return e;
```

```
        }
```

```
    }
```

```
}
```

```
}
```

Anonymous Inner Class
(see Swing scribble
example)

Example: Generics and Inheritance

1st consider this code w/o Generics

```
class ListUtilities {  
    public static Comparable max(List xs) {  
        Iterator xi = xs.iterator();  
        Comparable w = (Comparable) xi.next();  
        while (xi.hasNext()) {  
            Comparable x = (Comparable) xi.next();  
            if (w.compareTo(x) < 0) w = x;  
        }  
        return w;  
    }  
}
```

What dangers lurk here?

Consider what happens (and when)

```
List xs = new LinkedList();  
xs.add(new Byte(0));  
Byte x = (Byte) ListUtilities.max(xs);  
List ys = new LinkedList();  
ys.add(new Boolean(false));
```

// Evil run-time error

```
Boolean y = (Boolean) ListUtilities.max(ys);
```

With Generics We Get Compile Check

```
List<Byte> xs = new LinkedList<Byte>();
```

```
xs.add(new Byte(0));
```

```
Byte x = ListUtilities.max(xs);
```

```
List<Boolean> ys = new LinkedList<Boolean>();
```

```
ys.add(new Boolean(false));
```

```
// Compile-time error
```

```
Boolean y = ListUtilities.max(ys);
```

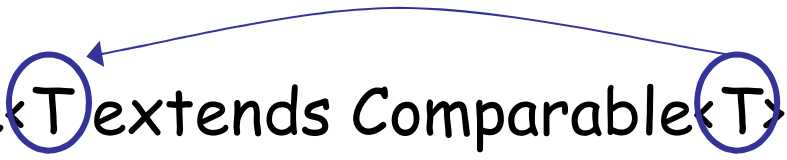
Generics and Inheritance

- Suppose you want to restrict the type parameter to express some restriction on the type parameter
- This can be done with a notion of subtypes
- Subtypes (weakly construed) can be expressed in Java using inheritance
- So it's a natural combination to combine inheritance with generics
- A few examples follow

Priority Queue Example

```
interface Comparable<I> { boolean lessThan(I); }
```

```
class PriorityQueue<T> extends Comparable<T> {  
    T queue[ ]; ...  
    void insert(T t) {  
        ... if ( t.lessThan(queue[i]) ) ...  
    }  
    T remove() { ... }  
    ...  
}
```



Said to be bounded

Bounded Parameterized Types

- The `<E extends Number>` syntax means that the type parameter of `MathBox` must be a subclass of the `Number` class
 - We say that the type parameter is **bounded**

`new MathBox<Integer>(5); //Legal`

`new MathBox<Double>(32.1); //Legal`

`new MathBox<String>("No good!"); //Illegal`

Bounded Parameterized Types

- Inside a parameterized class, the type parameter serves as a valid type. So the following is valid.

```
public class OuterClass<T> {  
  
    private class InnerClass<E extends T> {  
        ...  
    }  
    ...  
}
```

Syntax note: The <A extends B> syntax is valid even if B is an interface.

Bounded Parameterized Types

- Java allows multiple inheritance in the form of implementing multiple interfaces, so multiple bounds may be necessary to specify a type parameter. The following syntax is used then:

`<T extends A & B & C & ...>`

- Example

```
interface A {...}
```

```
interface B {...}
```

```
class MultiBounds<T extends A & B> {
```

```
...
```

```
}
```

Generics and Subtyping

- Is the following code snippet legal?

```
List<String> ls = new ArrayList<String>(); //1
```

```
List<Object> lo = ls; //2
```

- Line 1 is certainly legal. What about line 2? Is a List of Strings a List of Object. Intuitive answer for most is "sure!". But wait!
- The following code (if line 2 were allowed) would *attempt to assign an Object to a String!*

```
lo.add(new Object()); // 3
```

```
String s = ls.get(0); // 4: Problem !!!!!
```

Subclassing a generic class

```
import java.awt.Color;

public class Subclass extends MyClass<Color> {

    // You almost always need to supply a constructor
    public Subclass(Color color) {
        super(color);
    }

    public static void main(String[ ] args) {
        Subclass sc = new Subclass(Color.GREEN);
        sc.print(Color.WHITE);
    }
}
```

Wildcards

- Consider the problem of writing code that prints out all the elements in a collection before 1.5.

```
void printCollection(Collection c) {  
    Iterator i = c.iterator();  
    for (k = 0; k < c.size(); k++) {  
        System.out.println(i.next());  
    }  
}
```

Wildcards !!!

- So what *is* the supertype of all kinds of collections?
- Pronounced "collection of unknown" and denoted `Collection<?>`,
- A collection whose element type matches anything.
- It's called a *wildcard type* for obvious reasons.

```
void printCollection(Collection<?> c) {  
    for (Object e : c) {  
        System.out.println(e);  
    }  
}
```