Generics

Examples

What is Generics

- Collections can store Objects of any Type
- Generics restricts the Objects to be put in a collection
- Generics ease identification of runtime errors at compile time

How is Generics useful?

Consider this code snippet

```
List v = new ArrayList();
v.add(new String("test"));
Integer i = (Integer)v.get(0);  // Runtime error . Cannot cast from String to Integer
```

This error comes up only when we are executing the program and not during compile time.

How does Generics help

The previous snippet with Generics is

```
List<String> v = new ArrayList<String>();
v.add(new String("test"));
Integer i = v.get(0);  // Compile time error. Converting String to Integer
```

- The compile time error occurs as we are trying to put a String and convert it to Integer on retrieval.
- Observe we don’t have to do an explicit cast when we invoke the get method.
- We can also use interfaces in Generics

Wildcards

- Wildcards help in allowing more than one type of class in the Collections
- We come across setting an upperbound and lowerbound for the Types which can be allowed in the collection
- The bounds are identified using a ? Operator which means ‘an unknown type’

Upperbound

- List<? extends Number> means that the given list contains objects of some unknown type which extends the Number class

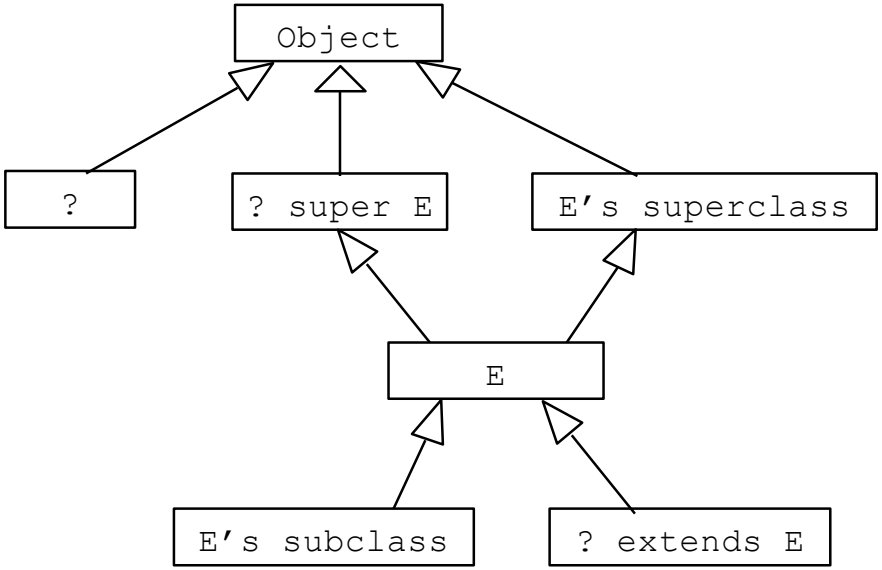
Consider the snippet

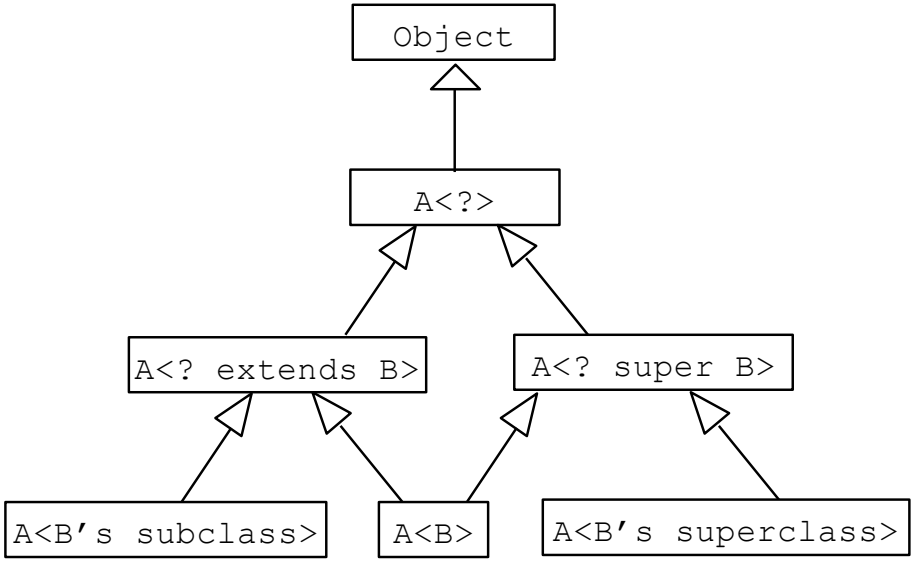
```
List<Integer> ints = new ArrayList<Integer>();
ints.add(2);
List<? extends Number> nums = ints;  // Allowed because of wildcards
nums.add(3.14);  // This is not allowed now after setting an upperbound
Integer x=ints.get(1);
```

Lowerbound

- List<? super Number> means that the given list contains objects of some unknown type which is superclass of the Number class

Generic Types and Wildcard Types





Example 1

- Give code to iterate (using an iterator) across a List **x** containing Strings
- All the Strings in **x** should be appended to a String named **answer**

Example 1

```
String answer = "";
for (Iterator<String> i = x.iterator(); i.hasNext();)
    answer += i.next();
```

Raw Type is Unsafe: Use Generics to make the following code safe

```
// Max.java: Find a maximum object
public class Max {
  /** Return the maximum between two objects */
  public static Comparable max(Comparable o1, Comparable o2) {
    if (o1.compareTo(o2) > 0)
      return o1;
    else
      return o2;
  }
}
```

Runtime Error:

Max.max("Welcome", 23);

Make it Safe

```
// Max1.java: Find a maximum object
public class Max1 {
  /** Return the maximum between two objects */
  public static <E extends Comparable<E>> E max(E o1, E o2) {
    if (o1.compareTo(o2) > 0)
      return o1;
    else
      return o2;
  }
}
```

```
Max.max("Welcome", 23);
```

Generics with subclass

- Consider a method that takes some collection of **Shape**s as a parameter, and returns the sum of all the areas.
- the actual collection might be **Collection<Shape>** or **Collection<Circle>** (**Circle** is a subclass of shape), Verify if the given method works if we call it with a Collection<Circle> object.
- **public double areaOfCollection (Collection<Shape> c)**

```
{ double sum = 0.0;
  for (Shape s : c)
     sum += s.getArea();
}
```

Generics with subclass

- Consider a method that takes some collection of **Shape**s as a parameter, and returns the sum of all the areas.
- the actual collection might be **Collection<Shape>** or **Collection<Circle>** (**Circle** is a subclass of shape), Verify if the given method works if we call it with a Collection<Circle> object. Use wild card to make it work.
- **public double areaOfCollection (Collection<Shape> c)**

```
{ double sum = 0.0;
  for (Shape s : c)
      sum += s.getArea();
}
```

Generics with subclass

```
public double areaOfCollection (Collection<? extends Shape> c)
{
  double sum = 0.0;
  for (Shape s : c)
    sum += s.getArea();
}
```

Generics with Comparator

Comparator interface is also generic

```
public interface Comparator<T> {
  int compare(T o1, T o2);
  boolean equals(Object o);
}
```

Create a comparator CompareByLength to sort Strings by length in x

```
List<String> x = new ArrayList<String>();
 Collections.sort(x, new CompareByLength())
```

Generics with Comparator

```
public class CompareByLength implements
Comparator<String> {
  int compare(String o1, String o2)
  {return o1.length() - o2.length();

}
```

Generics with Comparator

– Method that takes an array of objects and a collection and puts all objects in the array into the collection

```
static <T> void fromArrayToCollection(T[] a, Collection<T> c) {
    for (T o : a) {
        c.add(o);
    }
}
```

Generics with wildcards

- Does the following code compile? Make suitable modifications in the add method

```
public class WildCardDemo3 {
  public static void main(String[] args) {
    GenericStack<String> stack1 = new GenericStack<String>();
    GenericStack<Object> stack2 = new GenericStack<Object>();
    stack2.push("Java");
    stack2.push(2);
    stack1.push("Sun");
    add(stack1, stack2);
  }

  public static<T> void add(GenericStack<T> s1, GenericStack<T> s2) {
    while (!s1.isEmpty()) {
      s2.push(s1.pop());
    }
  }
}
```

Generics with wildcards

- Modified add method

```
public class WildCardDemo3 {
    public static void main(String[] args) {
      GenericStack<String> stack1 = new GenericStack<String>();
      GenericStack<Object> stack2 = new GenericStack<Object>();
      stack2.push("Java");
      stack2.push(2);
      stack1.push("Sun");
      add(stack1, stack2);
    }

    public static<T> void add(GenericStack<T> s1, GenericStack<? super T> s2) {
      while (!s1.isEmpty()) {
        s2.push(s1.pop());
      }
    }
  }
```

Generics with multiple bounds

- The syntax for specification of type parameter bounds is:
- <TypeParameter extends
  Class **&** Interface $_1$ **&** ... **&** Interface $_N$ >
- A list of bounds consists of one class and/or several interfaces.
- Example
- class Pair<A extends **Comparable<A> & Cloneable** ,
  B extends **Comparable<B> & Cloneable** >
  implements Comparable<Pair<A,B>>, Cloneable

{ ... }

This is a generic class with two type arguments A and B , both of which have two bounds.

Generics with Classes

```
public class Box<T> {

    private T item;

    public Box(T item) {
        this.item=item;
    }

    public T get() {
        return item;
    }

    public void set(T item) {
        this.item=item;
    }

    public String toString() {
        if(item!=null)
            return ""+item;
        else return "not set";
    }
}
```

Our box class is very basic

The type of Object is recorded as T (filled in when you declare a variable of type Box)

T is used to specify the type for item when declared as an instance datum, or passed as a parameter to a method or returned from a method

Note that T needs to have a toString implemented or this returns the address of item

What if we want to do

```
c.set(b.get( )+1);
```

This yields an error because b.get( ) returns an Integer and c.set expects a Double, so instead use

```
c.set(new
Double(b.get( )+1));
```

```
public class BoxUsers {
    public static void main(
        String[] args)    {
        Box<String> a;
        Box<Integer> b;
        Box<Double> c;
        Box<Object> d=null;
        a=new Box<>("hi there");
        b=new Box<>(100);
        c=new Box<>(100.1);
        System.out.println(a.get());
        a.set("bye bye");
        c.set(c.get()+1);
        System.out.println(a);
        System.out.println(c);
        System.out.println(d);
    }
}
```

Output:

hi there
bye bye
101.1
null

Multiple Generics

```
import java.util.ArrayList;
public class GenericStack<E, F> {
  private ArrayList<E> list =
    new ArrayList<E>();

  public int getSize() {
    return list.size();
  }
  public E peek() {
    return list.get(getSize()-1);
  }
  public void push(E o) {
    list.add(o);
  }
  public E pop() {
    E o = list.get(getSize()-1);
    list.remove(getSize()-1);
    return o;
  }
  public boolean isEmpty() {
    return list.isEmpty();
  }
  public void print(F f) {
    System.out.println(f);
  }
  public static void main(String [] args)
{
    GenericStack<String, Double> stack1 =
      new GenericStack<String, Double>();
    stack1.push("CSCI103");
    stack1.push("CSCI104");
    stack1.push("CSCI201");
    stack1.print(3.5);
    GenericStack<Integer, String> stack2
=
      new GenericStack<Integer,
String>();
    stack2.push(103);
    stack2.push(104);
    stack2.push(201);
    stack2.print("Hello CSCI201");
  }
}
```