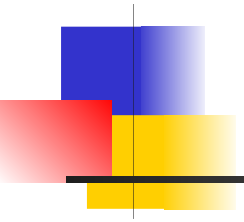


Generics and Type Parameters

Simple Generic Class

Example you have already written:

```
public class ArrayIterator<T>
             implements Iterator<T> {
    private T[] array;

    public ArrayIterator(T[] a) {
        this.array = a;
    }
    T get(int index) { return a[index]; }
```

```
new ArrayIterator<String>( string_ array );
new ArrayIterator<Double>( double_array );
```

Class Type Parameters is allowed only on instance members

- Java processes type parameters using "type erasure".
- As a result, a class type param cannot be used on static members.

```
public class MyClass<T> {
    private T attribute;        // OK
    private static T  x;        // ERROR

    public T getAttribute() { // OK
         return attribute;
    }
    public static T newInstance() // ERROR
```

Type Parameter with Bound

- You can limit type parameters using "`extends`" and "`super`".
- T is any type that implements Runnable:

```
public class TaskManager<T extends Runnable>
{
    private List<T> tasks;

    public void runAll() {
        for(Task t: tasks) t.run();
    }
    public void addTask(T task) {
        tasks.add(task);
    }
```

Wildcard Character ?

- Means "anything".
- Can use "?" on non-generic methods, too.
- Can be used with bounds, like "? extends Valuable".

```
public void printAll(List<?> list) {

    // forEach is same as "for-each" loop
    // requires Java 8
    list.forEach( (x) ->
        System.out.println(x) );
}
```

Demo: MoneyUtil.sortByCurrency

- The signature of the method is:

```
public static void sortByCurrency(
                List<? extends Valuable> list) {
    // sort items by currency
}
```

- But this code won't compile. Why? How to fix?

```
List<Coin> coins = Arrays.asList(
   new Coin(5,"Cents"),new Coin(1,"Baht"));

sortByCurrency( coins );    // ERROR
```

Generic Methods

- A static method can have its own type parameter.
- It can be method in an ordinary (non-generic class).
- java.util.Arrays and java.util.Collections are examples

```
public static <E> reverse(E[ ] array) {

    int size = array.length - 1;
    for(int k=0; k<size/2; k++) {
        E temp = a[k];
        a[k] = a[size-k];
        a[size-k] = temp;
    }
}
```

Calling Generic Methods

- The caller does not mention the type parameter.
- Java compiler *infers* the actual type from context.

```
Number[] array = new Number[]{1, 2, 3};

reverse( array ); // Java infers E = Number

String[] words = {"a", "b", "c"};

reverse( words ); // Java infers E = String
```

Example

- Write a generic "max" method.
- Find "max" of two objects that implement Comparable.

```
public static <E extends Comparable<E>>
      E max(E a, E b) {
    if (a.compareTo(b) > 0) return a;
    return b;
}
// This method has a problem:
String m = max("Cat", "Dog");    // OK
Coin m2 = max(new Coin(5), new Coin(10));
      // Compile error
      // Coin implements Comparable<Valuable>
```

"? super E"

Look at `Collections.fill` - replace all elements in list with copies of an object. It works even if the value (obj) is from a *subclass* of the List type.

```
public static <T> void
              fill(List<? super T>, T obj) ;

// This should work: fill a list of Number
// with an Integer (Integer extends Number)
List<Number> list = new ArrayList<>(10);
... // add some stuff to list
Collections.fill( list, new Integer(5) );
```

"? super E"

- `Collections.sort()` - can sort List using any Comparator that accepts T or a *superclass of T*.
- Can you see why this is necessary?

```
public static <T> sort(List<T> list,
      Comparator<? super T> comp)

List<Money> money = // create some money
Comparator<Valuable> compByCurrency =
    (a,b) -> a.getCurrency().compareTo(b.getCurrency);
// This should work:
Collections.sort( money, compByCurrency);
```

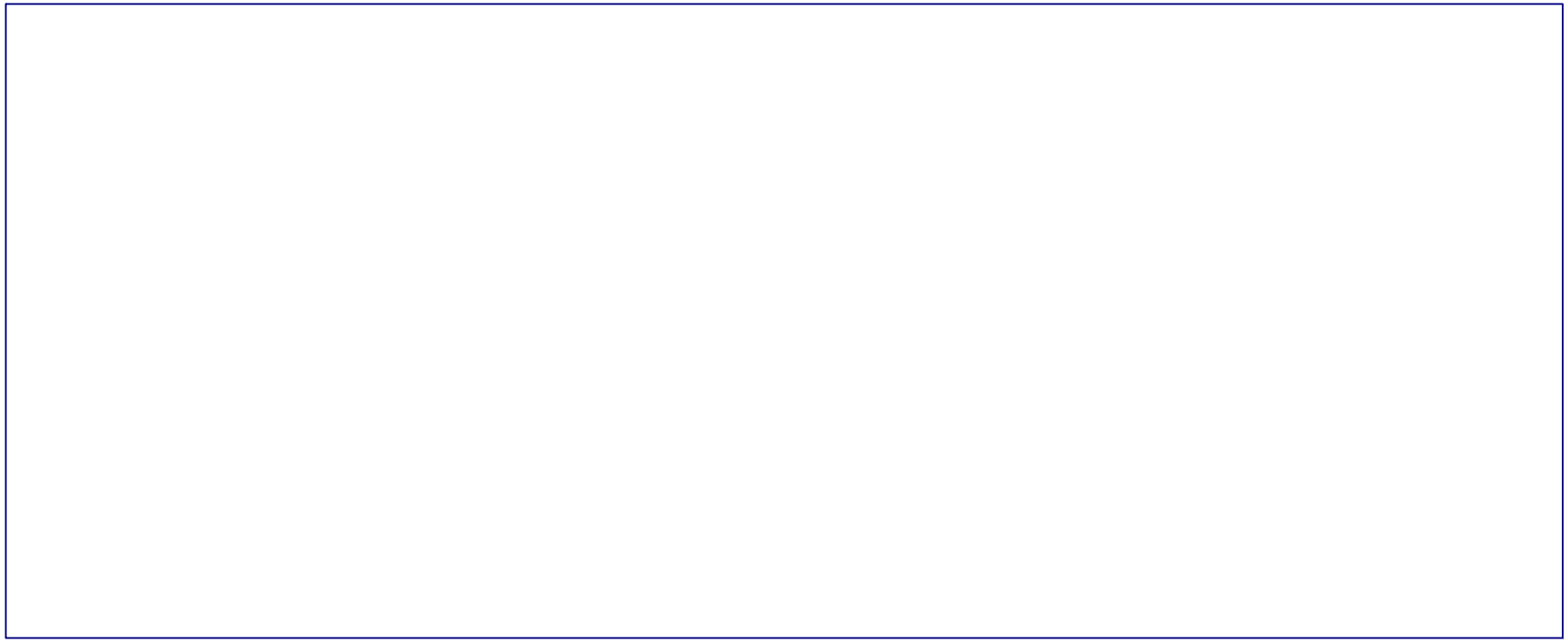
Exercise

- max accepts 2 objects that implement Comparable<E>
- Coin implements Comparable<Valuable>
- How can we make max( ) work with Coin?

```
static <E extends Comparable<E>>
      max(E a, E b) {
      if (a.compareTo(b)>0) return a;
      return b;
}
String s = max("Alpha", "Beta");  // OK!
Coin m = max(new Coin(5), new Coin(10));
          // ERROR!
```

Demo: CoinUtil.filterByCurrency

- Always returns `List<Valuable>`



Summary

1. Class type parameters apply to instance members only.
2. Static methods can have their own type parameter.
3. Bounds and wildcards:

   ? = match anything (can be used on *any* method)

   <? super E> = match superclass of E

   <E extends Foo> = bound on type param E

Type parameters are used a **lot** in API docs*, so you need to be able to read and understand them.*

References

Core Java for the Impatient - has a chapter on generics with many examples

Big Java, Chapter 18 (Generics)

Generics.doc - my write-up on generics and type param