

Lesson 11

Java Generics

Lesson Outline

1. Introduction to generics
2. Generic methods
3. Wildcards
4. Generic programming with generics

Introducing Generic Parameters

Prior to jdk 1.5, a collections of any type consisted of collections of Objects, and downcasting was required to retrieve elements of the correct type.

Example:

```
List words = new ArrayList();
words.add("Hello ");
words.add("world!");
String s = ((String)words.get(0))+((String)words.get(1))
assert s.equals("Hello world!");
```

In jdk 1.5, generic parameters were added to the declaration of collection classes, so that the above code could be rewritten as follows:

```
List<String> words = new ArrayList<String>();
words.add("Hello ");
words.add("world!");
String s = words.get(0)+words.get(1);
assert s.equals("Hello world!");
```

Benefits of Generics

1. *Stronger type checks at compile time.* A Java compiler applies strong type checking to generic code and issues errors if the code violates type safety. Fixing compile-time errors is easier than fixing runtime errors, which can be difficult to find.

Example of poor type-checking

```
List myList = new myList();
myList.add("Tom");
myList.add("Bob");
...
Employee tom = (Employee)myList.get(0); //no compiler check to prevent this
```

2. *Elimination of casts.* Downcasting is considered an “anti-pattern” in OO programming. Typically, downcasting should not be necessary; finding the right subtype should be accomplished with late binding.

Example of bad downcasting.

```
ClosedCurve[] closedCurves = //...populate with Triangles and Rectangles
if(closedCurves[0] instanceof Triangle)
    print( (Triangle)closedCurve[0].area());
else
    print( (Rectangle)closedCurve[0].area())
```

3. *Supports creation of generic algorithms.*

Example Task: Swap elements in a list (*generic methods* discussed in upcoming slide)

```
public void swapFirstLastNonGeneric(List list) {
    Object temp = list.get(0);
    list.set(0, list.get(list.size()-1));
    list.set(list.size()-1, temp);
}

public <T> void swapFirstLast(List<T> list) {
    T temp = list.get(0);
    list.set(0, list.get(list.size()-1));
    list.set(list.size()-1, temp);
}
```

Generics Terminology and Naming Conventions

1. In the List<String> example:

```
List<String> words = new ArrayList<String>();  
words.add("Hello ");  
words.add("world!");  
String s = words.get(0)+words.get(1);  
assert s.equals("Hello world!");
```

2. The class (found in the Java libraries) defined by

```
class ArrayList<T> { . . . }
```

is called a *generic class*, and T is called a *type variable* or *type parameter*.

3. The declaration

```
List<String> words;
```

is called a *generic type invocation*, String is (in this context) a *type argument*, and List<String> is called a *parametrized type*. Also, the class List, with the type argument removed, is called a *raw type*.

Note: When raw types are used where a parametrized type is expected, the compiler issues a warning because the compile-time checks that can usually be done with parametrized types cannot be done with a raw type.

4. The most commonly used type variables are:

- E - Element (used extensively by the Java Collections Framework)
- K - Key
- N - Number
- T - Type
- V - Value
- S, U, V etc. - 2nd, 3rd, 4th types

Creating Your Own Generic Class

```
public class SimplePair<K,V> {  
    private K key;  
    private V value;  
  
    public SimplePair(K key, V value) {  
        this.key = key;  
        this.value = value;  
    }  
    public K getKey() { return key; }  
    public V getValue() { return value; }  
}
```

Implementing a Generic Interface, Extending a Generic Class

```
public interface Pair<K, V> {  
    public K getKey();  
    public V getValue();  
}
```

One way: Create a parametrized type implementation

```
public class MyPair implements Pair<String, Integer>{  
    private String key;  
    private Integer value;  
  
    public MyPair(String key, Integer value) {  
        this.key = key;  
        this.value = value;  
    }  
  
    @Override  
    public String getKey() {  
        return key;  
    }  
  
    @Override  
    public Integer getValue() {  
        return value;  
    }  
}
```

Another way: Create a generic class implementation

```
public class OrderedPair<K, V> implements Pair<K, V> {  
    private K key;  
    private V value;  
  
    public OrderedPair(K key, V value) {  
        this.key = key;  
        this.value = value;  
    }  
  
    public K getKey() { return key; }  
    public V getValue() { return value; }
```

The same points apply for extending a generic class

Either:

```
class MyList<T> extends ArrayList<T>{  
}
```

Or:

```
public class MyList extends ArrayList<String >{  
}
```

How Java Implements Generics: *Type Erasure*

The compiler transforms the following generic code

```
List<String> words = new ArrayList<String>();
words.add("Hello ");
words.add("world!");
String s = words.get(0)+words.get(1);
assert s.equals("Hello world!");
```

into the following non-generic code:

```
List words = new ArrayList();
words.add("Hello ");
words.add("world!");
String s = ((String)words.get(0))+((String)words.get(1))
assert s.equals("Hello world!");
```

1. Java implements generics *by erasure* because the parametrized types like List<String>, List<Integer> and List<List<Integer>> are all represented at runtime by the single type List.
2. Also *erasure* is the process of converting the first piece of code to the second
3. The compiled code for generics will carry out the same downcasting as was required in pre-generics Java.

Benefits of this implementation approach:

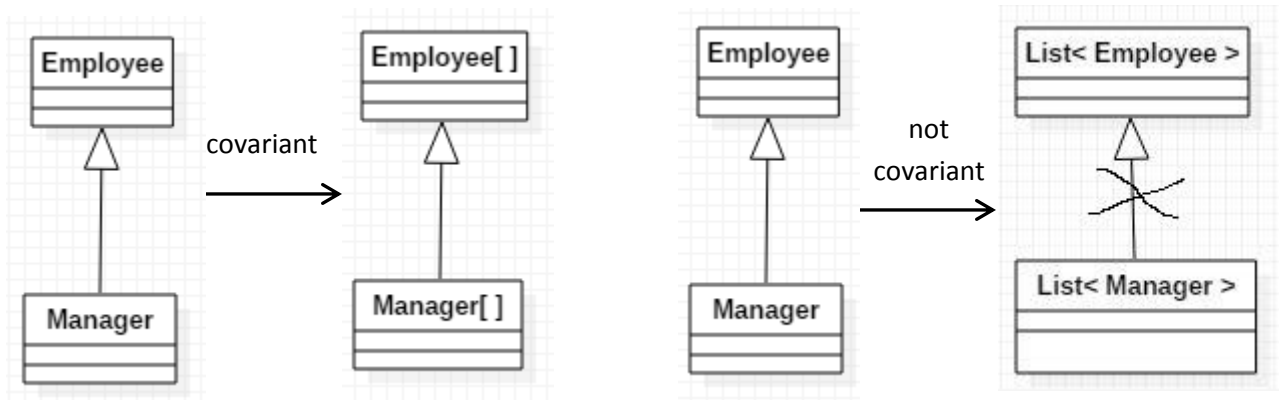
- A. No increase in the number of types in the language (in C++, each parametrized type is a genuinely different type)
- B. Backwards compatibility with non-generic code – for instance, in both generic and non-generic code, there is, at runtime, only one type List, so legacy code and generic code can intermingle without much difficulty.

The Downside of Java's Implementation of Generics

1. *Generic Subtyping Is Not Covariant.* For example: `ArrayList<Manager>` is not a subclass of `ArrayList<Employee>` (this is different from arrays: `ArrayList<Employee>` is a subclass of `AbstractList<Employee>`)

Example: What if this type of covariance were allowed?

```
List<Integer> ints = new ArrayList<Integer>();
ints.add(1);
ints.add(2);
List<Number> nums = ints; // compile-time error
nums.add(3.14);
assert ints.toString().equals("[1, 2, 3.14]");
```



Optional: (Requires strong mathematical background) The real meaning of “covariant”. A mathematical *category* is a collection of objects of the same type together with structure-preserving maps that map one object in the category to another. The *category of sets* has as its objects sets together with functions from one set to another. The collection *Class* of all classes (say Java classes) also forms a category; in this case, the “maps” between objects of this category are the arrows given by the subclass relation. Another category *ClassArray* is the collection of arrays having component type a Java class, like `Employee[]`, `Manager[]`, etc. Again the “maps” between these objects can be taken to be the subclass relation. The statement “array subtyping is covariant” means, technically speaking, that the transformation $F: \text{Class} \rightarrow \text{ClassArr}$ defined by $F(C) = C[]$ is *functorial*: If C is a subclass of D , then $F(C)$ is a subclass of $F(D)$. The transformation $G: \text{Class} \rightarrow \text{ParamList}$, given by $G(C) = \text{List}<C>$ is *not* functorial according to the rules of Java generics.

2. *Component type of an array may not be a type variable.* For example, cannot create an array like this:

```
T[] arr = new T[5];
```

Example:

```
class Annoying {
    public static <T> T[] toArray(Collection<T> c) {
        T[] a = new T[c.size()]; // compile-time error
        int i=0; for (T x : c) a[i++] = x;
        return a;
    }
}
```

3. *Component type of an array may not be a parametrized type.* For example: you cannot create an array like this:

```
List<String>[] = new List<String>[5];
```

Example:

```
class AlsoAnnoying {
    public static List<Integer>[] twoLists() {
        List<Integer> a = Arrays.asList(1,2,3);
        List<Integer> b = Arrays.asList(4,5,6);
        return new List<Integer>[] {a, b}; // compile-time error
    }
}
```

The reason for rules (2) and (3) is that *the component type of an array must be a reifiable type*.

Consider the analogous situation with arrays: The following statement

```
new String[size]
```

allocates an array, and stores in that array an indication that its components are of type `String`. However, executing

```
new ArrayList<String>()
```

allocates a list, but does not store in the list any indication of the type of its elements.

We say that Java *reifies* array component types but does not reify list element types (or other generic types). In the case of

```
new T[5]
```

there is no type information – we say a type variable is *not reifiable*.

Precise definition: A type is *reifiable* if the type is completely represented at run time — that is, if erasure does not remove any useful information.

Generic Methods

Generic methods are methods that introduce their own type parameters. This is similar to declaring a generic type, but the type parameter's scope is limited to the method where it is declared. Static and non-static generic methods are allowed, as well as generic class constructors.

The syntax for a generic method includes a type parameter, inside angle brackets, and appears before the method's return type. For static generic methods, the type parameter section must appear before the method's return type.

```
public static <K, V> boolean compare(SimplePair<K, V> p1, SimplePair<K, V> p2) {  
    return p1.getKey().equals(p2.getKey()) &&  
        p1.getValue().equals(p2.getValue());  
}
```

The complete syntax for invoking this method would be:

```
SimplePair<Integer, String> p1 = new SimplePair<>(1, "apple");  
SimplePair<Integer, String> p2 = new SimplePair<>(2, "pear");  
boolean areTheySame = Util.<Integer, String>compare(p1, p2);
```

In this case, as in most cases, the generic type can be inferred by the compiler, and can be left out.

```
SimplePair<Integer, String> q1 = new SimplePair<>(1, "apple");  
SimplePair<Integer, String> q2 = new SimplePair<>(2, "pear");  
boolean areTheySame2 = Util.compare(q1, q2);
```

Two cases where generic type must be displayed:

```
List<Integer> ints = Lists.<Integer>toList();  
List<Object> objs = Lists.<Object>toList(1, "two");
```

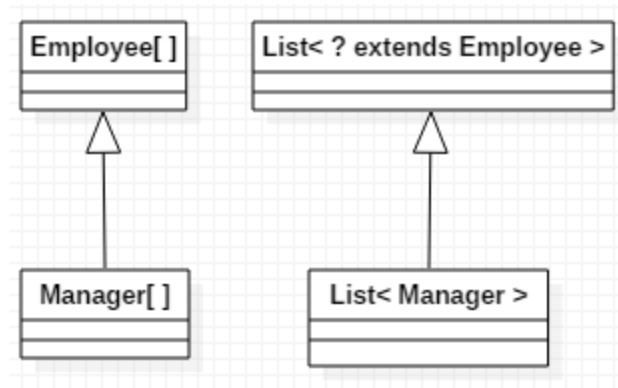
In the first case, without the type parameter, there is too little information. In the second, the possible types would be ambiguous – could be Object, Serializable, or Comparable.

RULE: In a call to a generic method:

- (1) if there are one or more arguments that correspond to a type parameter and they all have the same type then the type parameter may be inferred
- (2) if either
 - (a) there are no arguments that correspond to the type parameter, or
 - (b) the arguments belong to different subtypes of the intended typethen the type parameter must be given explicitly

The ? extends Bounded Wildcard

The fact that generic subtyping is not covariant – as in the example that `List<Manager>` is not a subtype of `List<Employee>` is inconvenient and unintuitive. This is remedied with the *extends bounded wildcard*.



The `?` is a *wildcard* and the “bound” in `List< ? extends Employee>` is the class `Employee`. `List< ? extends Employee>` is a *parametrized type with a bound*.

For any subclass `C` of `Employee`, `List<C>` is a subclass of `List< ? extends Employee>`.

So, even though the following gives a compiler error:

```
List<Manager> list1 = //... populate with managers
List<Employee> list2 = list1; //compiler error
```

the following does work:

```
List<Manager> list1 = //... populate with managers
List< ? extends Employee> list2 = list1; //compiles
```

(See demo `lesson11.lecture.generics.extend`)

Applications of the extends Wildcard

The Java Collection interface has an `addAll` method:

```
interface Collection<E> {  
    ...  
    public boolean addAll(Collection<? extends E> c);  
    ...  
}
```

The extends wildcard in the definition makes the following possible:

```
List<Employee> list1 = //...populate  
List<Manager> list2 = //... populate  
list1.addAll(list2);    //OK
```

If the interface had been defined like this:

```
interface Collection<E> {  
    . . .  
    public boolean addAll(Collection<E> c);  
    . . .  
}
```

it would mean for example that `addAll` could accept only a Collection of *Employees*:

```
List<Employee> list1 = //...populate  
List<Employee> list2 = //...populate BUT  
list1.addAll(list2); //OK
```

```
List<Employee> list1 = //...populate  
List<Manager> list2 = //...populate  
list1.addAll(list2); //compiler error
```

Another Example Using addAll

```
List<Number> nums = new ArrayList<Number>();  
List<Integer> ints = Arrays.asList(1, 2);  
List<Double> dbls = Arrays.asList(2.78, 3.14);  
nums.addAll(ints);  
nums.addAll(dbls);  
assert nums.toString().equals("[1, 2, 2.78, 3.14]");
```

Limitations of the extends Wildcard

When the extends wildcard is used to define a parametrized type, the type cannot be used for adding new elements.

Example:

Recall the addAll method from Collection:

```
interface Collection<E> {  
    ...  
    public boolean addAll(Collection<? extends E> c);  
    ...  
}
```

The following produces a compiler error:

```
List<Integer> ints = new ArrayList<Integer>();  
ints.add(1);  
ints.add(2);  
List<? extends Number> nums = ints;  
nums.add(3.14); // compile-time error  
assert ints.toString().equals("[1, 2, 3.14]");
```

The error arises because an attempt was made to insert a value in a parametrized type with extends wildcard parameter. With the extends wildcard, values can be *gotten* but not *inserted*.

The difficulty is that adding a value to nums makes a commitment to a certain type (Double in this case), whereas nums is defined to be a List that accepts subtypes of Number, but *which* subtype is not determined. The value 3.14 cannot be added because it might not be the right subtype of Number.

NOTE: Although it is not possible to *add* to a list whose type is specified with the extends wildcard, this does not mean that such a list is read-only. It is still possible to do the following operations available to any List:

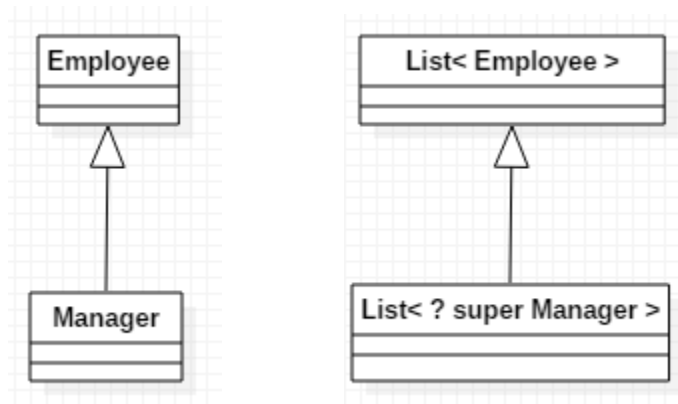
remove, removeAll, retainAll

and also execute the static methods from Collections:

sort, binarySearch, swap, shuffle

The ? super Bounded Wildcard

The type `List<? super Manager>` consists of objects of any supertype of the `Manager` class, so objects of type `Employee` and `Object` are allowed.



Example from the `Collections` class:

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

Typical use:

```
List<Object> objs = Arrays.<Object>asList(2, 3.14, "four");
List<Integer> ints = Arrays.asList(5, 6);
Collections.copy(objs, ints);
assert objs.toString().equals("[5, 6, four]");
```

Values can be read from (or “gotten” from) `src`, which is of type `List<? extends T>`, but then to insert values in the destination list `dst`, `dst` may not be typed using the `extends` wildcard. Instead, it is typed as `List<? super T>`. This means that any object whose type is a supertype of `T` may be placed in `dst`.

In the example, `src` is a `List<Integer>` and `dst` is a `List<Object>`. The type `T` is `Integer` and any supertype of `Integer` may be copied into `objs`.

The Get and Put Principle for Bounded Wildcards

The Get and Put Principle: use an *extends* wildcard when you only *get* values out of a structure, use a *super* wildcard when you only *put* values into a structure, and don't use a wildcard when you *both* get and put.

Example. This method takes a collection of numbers, converts each to a double, and sums them up:

```
public static double sum(Collection<? extends Number> nums) {  
    double s = 0.0;  
    for (Number num : nums) s += num.doubleValue();  
    return s;  
}
```

Since this uses *extends*, all of the following calls are legal:

```
List<Integer> ints = Arrays.asList(1,2,3);  
assert sum(ints) == 6.0;
```

```
List<Double> doubles = Arrays.asList(2.78,3.14);  
assert sum(doubles) == 5.92;
```

```
List<Number> nums = Arrays.<Number>asList(1,2,2.78,3.14);  
assert sum(nums) == 8.92;
```

The first two calls would not be legal if *extends* was not used.

Whenever you use the add method for a Collection, you are inserting values, and so ? super must be used.

Example:

```
public static void count(Collection<? super Integer> ints, int n) {  
    for(int i = 0; i < n; ++i) {  
        ints.add(i);  
    }  
}
```

Since super was used, the following are legal:

```
List<Integer> ints1 = new ArrayList<>();  
count(ints1, 5);  
System.out.println(ints1); //output: [0,1,2,3,4]  
  
List<Number> ints2 = new ArrayList<>();  
count(ints2, 5);  
ints2.add(5.0);  
System.out.println(ints2); //output: [0,1,2,3,4, 5.0]  
  
List<Object> ints3 = new ArrayList<>();  
count(ints3, 5);  
ints3.add("five");  
System.out.println(ints3); //output: [0,1,2,3,4, five]
```

The last two calls would not be legal without the use of the ? super wildcard.

Whenever you both put values into and get values out of the same structure, you should not use a wildcard.

```
public static double sumCount(Collection<Number> nums, int n) {  
    count(nums, n);  
    return sum(nums);  
}
```

The collection is passed to both `sum` and `count`, so its element type must both extend `Number` (as `sum` requires) and be super to `Integer` (as `count` requires). The only two classes that satisfy both of these constraints are `Number` and `Integer`, and we have picked the first of these. Here is a sample call:

```
List<Number> nums = new ArrayList<Number>();  
double sum = sumCount(nums,5);  
assert sum == 10;
```

Since there is no wildcard, the argument must be a collection of `Number`.

.Two Exceptions to the Get and Put Rule

1. In a Collection that uses the extends wildcard, null can always be added legally (null is the “ultimate” subtype)

```
List<Integer> ints = new ArrayList<Integer>();
ints.add(1);
ints.add(2);
List<? extends Number> nums = ints;
nums.add(null); // ok
assert nums.toString().equals("[1, 2, null]");
```

2. In a Collection that uses the super wildcard, any object of type Object can be read legally (Object is the “ultimate” supertype).

```
List<Object> objs = Arrays.<Object>asList(1,"two");
List<? super Integer> ints = objs;
String str = "";
for (Object obj : ints) str += obj.toString();
assert str.equals("1two");
```

Unbounded Wildcard, Wildcard Capture, Helper Methods

1. The wildcard `?`, without the `super` or `extends` qualifier, is called the *unbounded wildcard*.
2. `Collection<?>` is an abbreviation for `Collection<? extends Object>`
3. Important application of the unbounded wildcard involves *wildcard capture*:

Example: Try to copy the 0th element of a general list to the end of the list

First Try:

```
public void copyFirstToEnd(List<?> items) {  
    items.add(items.get(0)); //compiler error  
}
```

Compiler error arises because we are trying to add to a List whose type involves the extends wildcard.

Solution: Write a helper method that *captures the wildcard*.

```
public void copyFirstToEnd2(List<?> items) {  
    copyFirstToEndHelper(items);  
}  
  
private <T> void copyFirstToEndHelper(List<T> items) {  
    T item = items.get(0);  
    items.add(item);  
}
```

Notes:

- A. Passing `items` into the helper method causes the unknown type `?` to be “captured” as the type `T`.
- B. In the helper method, getting and setting values is legal because we are not dealing with wildcards in that method.

Generic Programming Using Generics

Generic programming is the technique of implementing a procedure so that it can accommodate the broadest possible range of inputs.

For instance, consider implementation of a max function. It is not desirable to have a different max function for Integer, String, Double, etc. Generic programming , in this case, would create a max function that could handle every possible type for which the concept “max” makes sense.

See demo `lecture.generics.BoundedTypeVariable` for examples.