Lecture 10: Java Generics:

Weaving the Universal into the Fabric of the Particular

Wholeness Statement

Java generics facilitate stronger type-checking, making it possible to catch potential casting errors at compile time (rather than at runtime), and in many cases eliminate the need for downcasting. Generics also make it possible to support the most general possible API for methods that can be generalized. We see this in simple methods like max and sort, and also in the new Stream methods like filter and map. Generics involve type variables that can stand for any possible type; in this sense they embody a universal quality. Yet, it is by virtue of this universal quality that we are able to specify particular types (instead of using a raw List, we can use List<T>, which allows us to specify a list of Strings – List<String> -- rather than a list of Objects, as we have to do with the raw List). This shows how the lively presence of the universal sharpens and enhances the particulars of individual expressions. Likewise, contact with the universal level of intelligence sharpens and enhances individual traits.

Lesson Outline

1. Introduction to generics
2. Generic methods
3. Wildcards
4. Understanding Common Generic Signatures
5. Generic programming with generics

Introducing Generic Parameters

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

```
List words = new ArrayList();
words.add("Hello");
words.add(" world!");
String s = ((String)words.get(0)) + ((String)words.get(1));
System.out.print(s);  //output: 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);
System.out.print(s);  //output: 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. Detecting errors at compile time is always preferable to discovering them at runtime (especially since, otherwise, the problem might not show up until the software has been released).

Example of poor type-checking

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

2. *Reduced Downcasting*. Downcasting is considered an "anti-pattern" in OO programming. Typically, downcasting should not be necessary (though there are plenty of exceptions to this rule); finding the right subtype should be accomplished with late binding.

Example of bad downcasting

```
//Populate with Triangles and Rectangles
ClosedCurve[] closedCurves = . . .
if(closedCurves[0] instanceOf Triangle) {
    print((Triangle)closedCurve[0].area());
}
else {
    print((Rectangle)closedCurve[0].area());
}
```

3. *Supports the most general possible API for methods that can be generalized .*

Example *Task*: get the `max` element in a list (difficult to do without generics)

```
public static Integer max0(List<Integer> list) {
            Integer max = list.get(0);
            for(Integer i : list) {
                  if(i.compareTo(max) > 0) {
                        max = i;
                  }
            }
            return max;
      }
```

```
public static <T extends Comparable<T>> T max1(List<T> list) {
   T max = list.get(0);
   for(T i : list) {
      if(i.compareTo(max) > 0) {
         max = i;
      }
   }
   return max;
}
```

Generics Terminology and Naming Conventions

1. In the following code:

```
List<String> words = new ArrayList<String>();
words.add("Hello");
words.add(" world!");
String s = words.get(0) + words.get(1);
System.out.print(s);  //output: Hello world!
```

the class (found in the Java libraries) with declaration

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

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

2. The delcaration

```
List<String> words; //read:"List of String"
```

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 the difference between *generic class* and *parameterized type*, and between *type variable* and *type argument*

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.

3. Commonly used type variables:

- 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 or Interface

```
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; }
}
```

Notes:

1. The class declaration introduces type variables K, V. These can then be used in the body of the class as types of variables and method arguments and return types. The same principle applies when defining a generic interface.
2. The type variables may be realized as any Java object type (even user-defined), but not as a primitive type.

Usage Example:

```
SimplePair<Integer,String> pair
     = new SimplePair<>(10123, "Jim");
String employeeId = pair.getKey(); //returns Jim's ID
```

Implementing a Generic Interface

```
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; }
```

See Demo: lesson10.lecture.generics.pairexamples

Extending a Generic Class

The same points apply for extending a generic class.

Either: Create a generic subclass

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

Or: Create a parametrized type subclass

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

How Java Implements Generics: *Type Erasure*

- ***Definition :*** *The information on generics is used by the compiler but is not available at runtime. This is called* type erasure.
- Generics are implemented using an approach called *type erasure*: The compiler uses the generic type information to compile the code, but erases it afterward.
- Thus, the generic information is not available at runtime. This approach enables the generic code to be backward compatible with the legacy code that uses raw types.
- The generics are present at compile time. Once the compiler confirms that a generic type is used safely, it converts the generic type to a raw type. For example, the compiler checks whether the following code in (a) uses generics correctly and then translates it into the equivalent code in (b) for runtime use. The code in (b) uses the raw type.

```
ArrayList<String> list = new ArrayList<String> ();
list.add("Oklahoma");
String state = list.get(0);
```

(a)

```
ArrayList list = new ArrayList();
list.add("Oklahoma");
String state = (String)(list.get(0));
```

(b)

How Java Implements Generics: *Type Erasure (cont.)*

1. Java is said to implement 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(a) to the second(b).

3. The compiled code for generics will carry out the same downcasting as was required in pre-generics Java.

The Downside(Restriction) or Generics Is Unintuitive

Ways That Java's Implementation of Generics Is Unintuitive

1. *Generic Subtyping Is Not Covariant.*

`Manager`	is a subclass of	`Employee`
	BUT	
`ArrayList<Manager>`	is NOT a subclass of	`ArrayList<Employee>`

2. *Array Subtyping Is Covariant*

`Manager`	is a subclass of	`Employee`
	AND	
`Manager[]`	is a subclass of	`Employee[]`

Exercise 11.1

This exercise illustrates one reason why generic subtyping is not allowed to be covariant. Examine the following code and answer the following:

1. In what step is the coder attempting to use covariance?
2. Assuming the Java designers had decided to permit covariant generic subtyping, what happens in the code that is undesirable – and should not be allowed? (Hint: What is contained in the `ints` list at the end?)

```
List<Integer> ints = new ArrayList<Integer>();
ints.add(1);
ints.add(2);
List<Number> nums = ints;
nums.add(3.14);
System.out.print(ints);
```

Solution to Exercise 11.1

```
List<Integer> ints = new ArrayList<Integer>();
ints.add(1);
ints.add(2);
//assuming covariance, this step would be allowed
List<Number> nums = ints;
nums.add(3.14);
System.out.print(ints);   //output:  [1, 2, 3.14] – not desirable
```

You will get this below error for this code

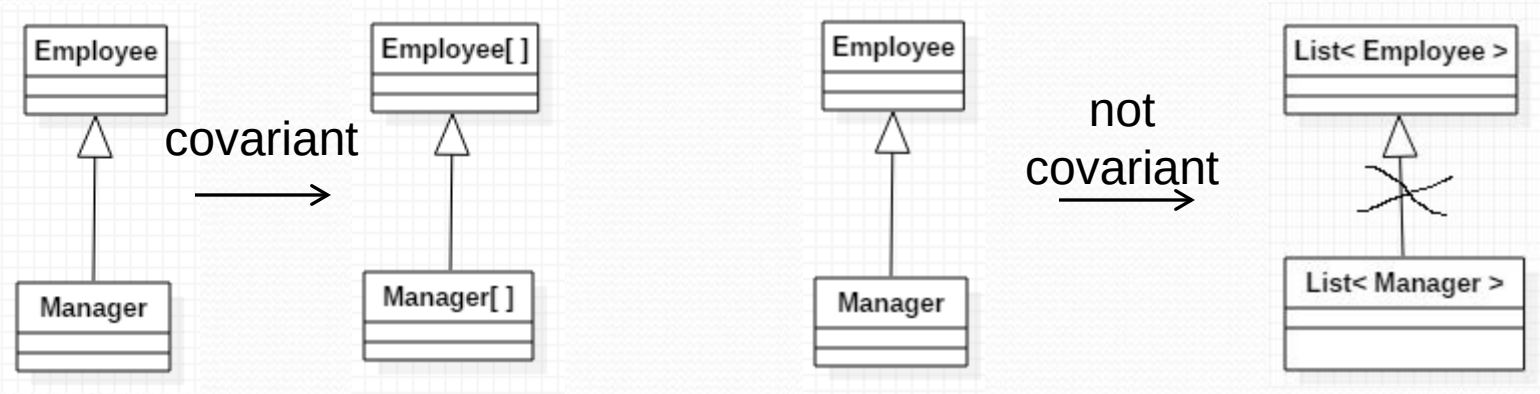
```
import java.util.ArrayList;
public class SlideExampleCode {
public static void main(String args[]){
    List<Integer> ints = new ArrayList<Integer>();
       ints.add(1);
       ints.add(2);
       List<Number> nums = ints;
       nums.add(3.14);
       System.out.print(int
    }
}
```

Type mismatch: cannot convert from List<Integer> to List<Number>

2 quick fixes available:

- Change type of 'nums' to 'List<Integer>'
- Change type of 'ints' to 'List<Number>'

Ways That Java's Implementation of Generics Is Unintuitive



Optional: The Origin of the Term "Covariant"

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 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 *class_array* 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.

Another category is *param_list* – the collection of all parametrized lists. The statement "array subtyping is covariant" means, technically speaking, that the transformation F: *class*-> *class_array* 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: *class*-> *param_list* , given by G(C) = List<C> is *not* functorial according to the rules of Java generics.

Ways That Java's Implementation of Generics Is Unintuitive (cont)

2. *Component type of an array is not allowed to be a type variable*. For example, we cannot create an array like this (the compiler has no information about what type of object to create)

```
T[] arr = null; //this is ok so far
arr = new T[5]; //this produces a compiler error
```

T[] arr = (T[]) new Object[5]; // No Error – Right way

Example: [This issue arises in Java's Collection classes]

```
class AbstractCollectionFirstTry {
   public static <T> T[] toArray(Collection<T> coll) {
          T[] arr = new T[coll.size()]; //compiler error
          int k = 0;
          for(T element : coll)
             arr[k++] = element;
             return arr;
   }
}
```

See demo: lesson11.lecture.generics.toarray

Ways That Java's Implementation of Generics Is Unintuitive (cont)

3. *Component type of an array is not allowed to be a parametrized type*. For example: you cannot create an array like this: Causes compilation error.

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

Pair<String>[] a = new Pair<String>[10];

Example:

```
class Another {
   public static List<Integer>[] twoLists() {
        List<Integer> list1 = Arrays.asList(1, 2, 3);
        List<Integer> list2 = Arrays.asList(4, 5, 6);

        //compiler error
        return new List<Integer>[] {list1, list2};
   }
}
```

Reifiable Types & Non-Reifiable Types

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

A *reifiable* type is a type whose type information is fully available at runtime. This includes primitives, non-generic types, raw types, and invocations of unbound wildcards.

In case of arrays
String[] obj=new String[10];
will remain the same at runtime as well.

Non-reifiable types are types where information has been removed at compile-time by type erasure.
So List<String> list=new ArrayList<String>

at runtime will be
List list=new ArrayList();

In the case of
new List<Integer>[] //not allowed, because type has been erased
because the List type does not store component type information, the resulting array is unable to store component type information (which violates rules for arrays). We say *parameterized types* (as well as type variables) *are not reifiable.*

Lesson Outline

1. Introduction to generics
2. Generic methods
3. Wildcards
4. Understanding Common Generic Signatures
5. Generic programming with generics

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.

```
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());
}
```

Calling a Generic Method

Earlier versions of Java required that you specify the generic type arguments when calling a generic method (see below), but current versions are always able to infer types, so the type arguments can be left out.

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);
```

The generic type(s) can always 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);
```

Example

```
public class GenericMethod {
        public static void main(String[] args) {
                args = new String[]{"CA", "US", "MX", "HN", "GT"};
                print(args);
                Integer[] x = new Integer[]{10,20,30,40,50};
                print(x);
         }
       static <E> void print(E[] a) {
                    for (E ae : a) {
                    System.out.printf("%s ", ae);
                    }
                    System.out.println();
                    }
```

Using Generic Methods to Generalize Behavior

The following code counts occurrences of a target String inside a given input array.

```
public static int countOccurrences(String[] arr, String target) {
    int count = 0;
    if (target == null) {
        for (String item : arr)
            if (item == null)
                count++;
    } else {
        for (String item : arr)
            if (target.equals(item))
                count++;
    }
    return count;
}
```

But the same procedure could be used to find a target of any given type in an array of the same type. Generic methods allow us to generalize from type String to an arbitrary type T.

Exercise 11.2

The code for `countOccurrences` is in the package lesson11.exercise_2 in the InClassExercises project.

Do the following:

1. Turn the method into a generic method so that it can be used to count occurrences of an object of any type in an array whose components are of the same type.
2. Then try writing the code for your generic method using a `Stream` pipeline instead of imperative code

Main Point 1

Generic methods make it possible to create general-purpose methods in Java by declaring and using one or more type variables in the method. This allows a user to make use of the method using any data type that is convenient, with full compiler support for type-checking. Likewise, when individual awareness has integrated into its daily functioning the universal value of transcendental consciousness, the awareness is maximally flexible, able to flow in whatever direction is required at the moment, free of rigidity and dominance of boundaries.

Another Generalization Example: Finding the max

Problem: Find the max value in a List.

1. **Easy Case**: First try finding the max of a list of Integers:

```
public static Integer max(List<Integer> list) {
    Integer max = list.get(0);
    for(Integer i : list) {
        if(i.compareTo(max) > 0) {
            max = i;
        }
    }
    return max;
}
```

2. **Try to generalize** to an arbitrary type T (this first try doesn't quite work…)

```
public static <T> T max1(List<T> list) {
    T max = list.get(0);
    for(T i : list) {
        if(i.compareTo(max) > 0) {
            max = i;
        }
    }
    return max;
}
```

Problem: T may not be a type that has a `compareTo` operation – we get a compiler error

Solution: It is possible to use the "extends" keyword to force the type T to be an implementer of the Comparable interface. This produces a *bounded type variable* `T extends Comparable`

```
public static <T extends Comparable> T max(List<T> list) {
    T max = list.get(0);
    for(T i : list) {
        if(i.compareTo(max) > 0) {
            max = i;
        }
    }
    return max;
}
```

Demo: lesson11.lecture.generics.genericprogrammingmax

Generalizing Even Further

- The Comparable interface is also generic. For a given class C, implementing the Comparable interface implies that comparisons will be done between a current instance of C and another instance; the other instance type is the type argument to use with Comparable. For example, `String implements Comparable<String>.` This leads to:

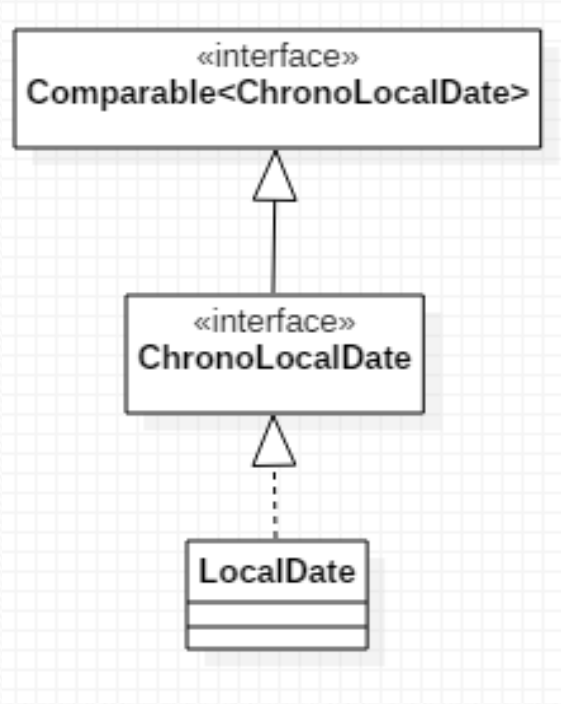
```
public static <T extends Comparable<T>> T max(List<T> list) {
    T max = list.get(0);
    for(T i : list) {
        if(i.compareTo(max) > 0) {
            max = i;
        }
    }
    return max;
}
```

- This version of max can be used for most kinds of Lists, but there are exceptions. Example:

```
public static void main(String[] args) {
    List<LocalDate> dates = new ArrayList<>();
    dates.add(LocalDate.of(2011, 1, 1));
    dates.add(LocalDate.of(2014, 2, 5));
    LocalDate mostRecent = max(dates); //compiler error
}
```

Finding the max (cont.)

- **The Problem**: `LocalDate` does not implement `Comparable<LocalDate>`. Instead, the relationship to `Comparable` is the following:



`LocalDate` implements `Comparable<ChronoLocalDate>`

What is needed is a max function that accepts types `T` that implement not just `Comparable<T>`, but even `Comparable<S>` for any supertype `S` of `T`.

Here, `T` is `LocalDate`. We want max to accept a list of `LocalDates` using a `Comparable<S>` for any supertype `S` of `LocalDate`.

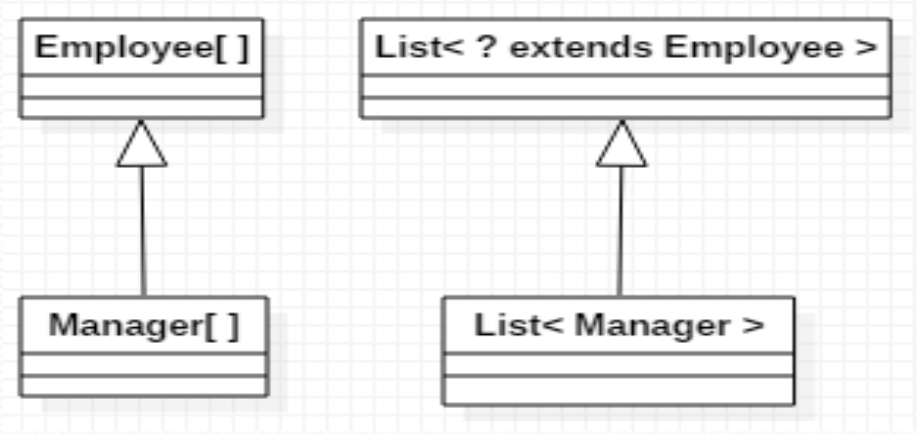
The solution lies in the use of *bounded wildcards.*

Lesson Outline

1. Introduction to generics
2. Generic methods
3. Wildcards
4. Understanding Common Generic Signatures
5. Generic programming with generics

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 to a large extent with the extends *bounded wildcard*.



The ? extends Bounded Wildcard (cont.)

- The ? is a *wildcard* and the "bound" in `List<? extends Employee>` is the class Employee. `List<? extends Employee>` is called 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
```

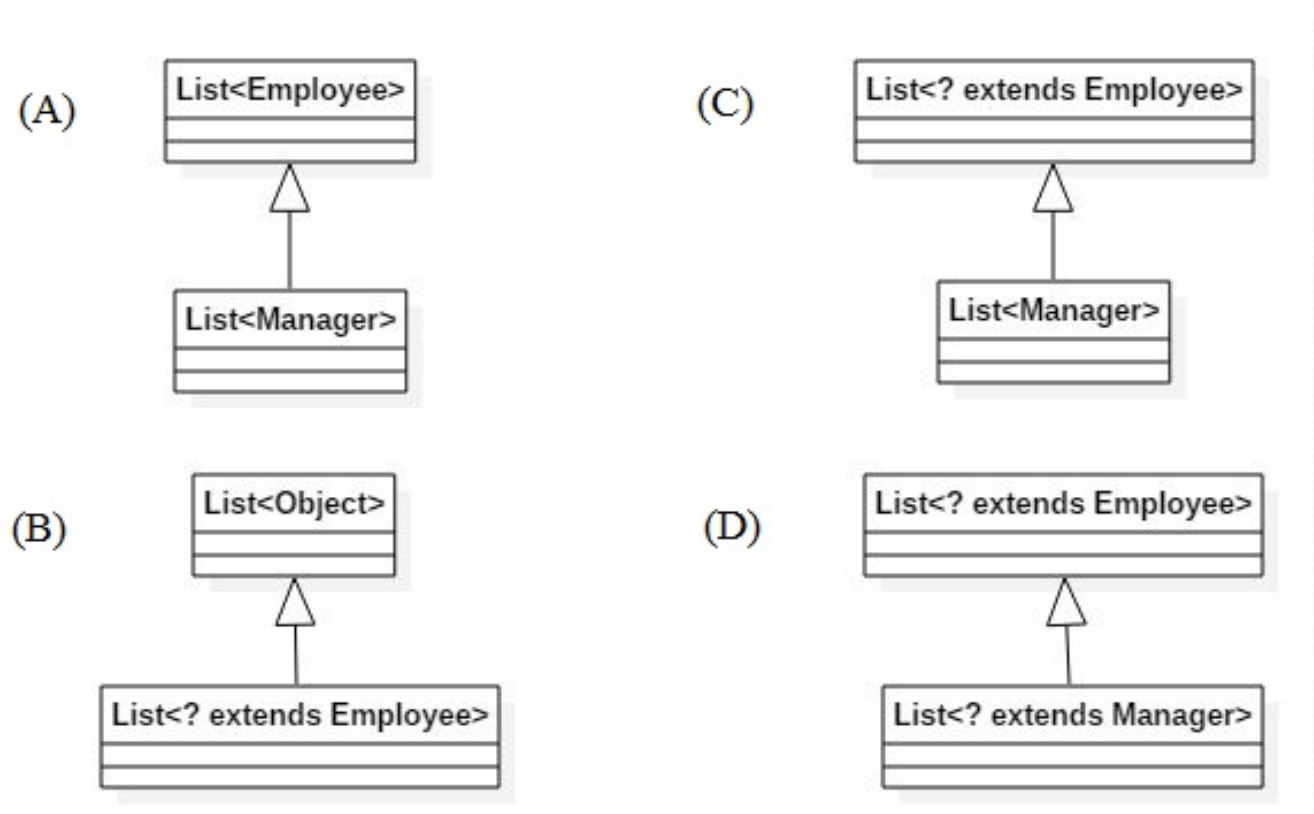
the following does work:

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

(See demo lesson10.lecture.generics.extend)

Exercise 11.3

Determine which diagrams are correct and which are not.



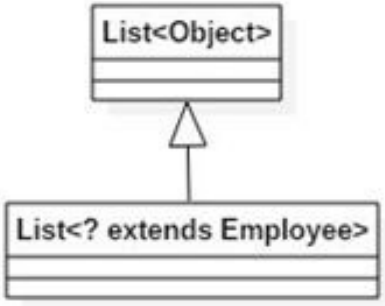
Exercise 11.3

Determine which diagrams are correct and which are not.

(A)

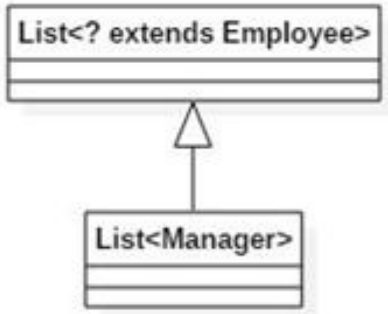
X

(B)



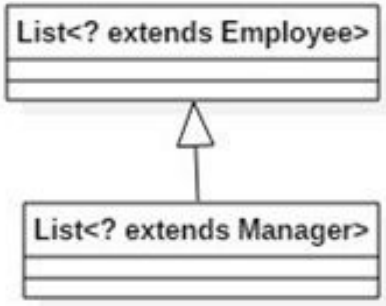
X

(C)



√

(D)



√

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

Applications of the ? extends Wildcard

If the interface method had been declared like this:

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

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

```
List<Employee> list1 = //….populate
List<Employee> list2 = //…populate
list1.addAllBad(list2);   //OK
```

BUT

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

See the demo: lesson11.lecture.generics.addall

Another Example Using addAll

```
List<Number> nums = new ArrayList<Number>();
List<Integer> ints = Arrays.asList(1, 2);
List<Double> doubles = Arrays.asList(2.78, 3.14);
nums.addAll(ints);
nums.addAll(doubles);
System.out.println(nums);   //output: [1, 2, 2.78, 3.14]
```

Here, since `Integer` and `Double` are both subtypes of `Number`, it follows that `List<Integer>` and `List<Double>` are subtypes of `List<? extends Number>`, and `addAll` maybe used on `nums` to add elements from both `ints` and `doubles`.

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);        //compiler error
System.out.println(ints.toString());
nums.add(null);    //OK
```

Limitations of the extends Wildcard (cont.)

- 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`:

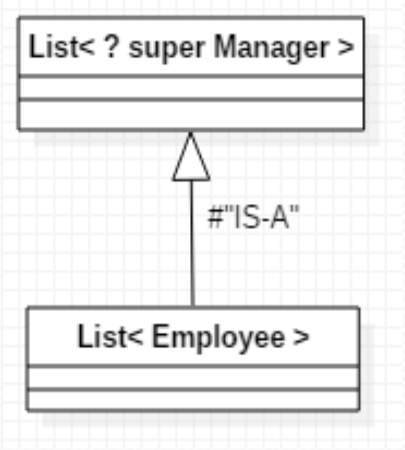
```
remove, removeAll, retainAll
```

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.



- This diagram can be read as follows: A `List<Employee>` is a `List` whose type argument `Employee` is a supertype of `Manager`. Therefore, a `List<Employee>` IS-A `List<? super Manager>`.

Limitations of the super Wildcard

When the super wildcard is used to define a Collection of parametrized type, it is inconvenient to *get* elements from the Collection; elements can be gotten, but not typed.

Example:

```
List<? super Integer> test = new ArrayList<>();
test.add(5);
System.out.println(test.get(0));
```

However, if we try to assign a type to the return of the get method, we get a compiler error – the compiler has no way of knowing which supertype of Integer is being gotten.

```
Integer val = test.get(0);      //compiler error
Number val = test.get(0);       //compiler error
Comparable val = test.get(0);   //compiler error
Object val = test.get(0);       //OK
```

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 at all when you *both get and put* values.

Example-1 This method takes a collection of numbers, converts each to a `double`, and sums them up. The loop *gets* from an ? extends type of Collection.

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

Since `List<Integer>`, `List<Double>` are subtypes of `Collection<? extends Number>`, the following are legal:

```
List<Integer> ints = Arrays.asList(1, 2, 3);
Integer val = sum(ints);   //output: 6.0

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

The Get and Put Principle for Bounded Wildcards (cont.)

Example-2 (from the Collections class)

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

Note that we *get* from `source`, which is typed using `extends`, and we *insert* into `destination`, which is typed using `super`. It follows that any subtype of `T` may be *gotten* from `source`, and any supertype of T may be *inserted* into `destination`.

Sample usage:

```
List<Object> objs = Arrays.asList(2, 3.14, "four");
List<Integer> ints = Arrays.asList(5, 6);
//copy the narrow type (Integer) into the wider type (Object)
Collections.copy(objs, ints);
System.out.println(objs.toString());      //output:  [5, 6, four]
```

The Get and Put Principle for Bounded Wildcards (cont.)

Example-3 (using `? super`) Whenever you use the `add` method for a `Collection`, you are inserting values, and so `? super` should be used.

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

The count method "counts" from 0 to n-1, adding these numbers to the input Collection ints.

The Get and Put Principle for Bounded Wildcards (cont.)

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]
```

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

- In the second call, `ints2` is of type `List<Number>` which "IS-A" `Collection<? super Integer>` (since `Number` is a superclass of `Integer`), so the `count` method can be called.
- In the third call, `ints3` is of type `List<Object>` which also "IS-A" `Collection<? super Integer>` (since `Object` is a superclass of `Integer`), so the `count` method can be called here too.
- Note that the `add` methods shown here have nothing to do with the `? super` declaration – you can add a `double` to a `List<Number>` and a `String` to a `List<Object>` for the usual reasons.

The Get and Put Principle for Bounded Wildcards (cont.)

Example-4 Improving implementation of the `max` function

We saw before that this implementation of `max` was not general enough

```
public static <T extends Comparable<T>> T max(List<T> list) {
    T max = list.get(0);
    for(T i : list) {
        if(i.compareTo(max) > 0) {
            max = i;
        }
    }
    return max;
}
```

We encountered a compiler error here:

```
public static void main(String[] args) {
    List<LocalDate> dates = new ArrayList<>();
    dates.add(LocalDate.of(2011, 1, 1));
    dates.add(LocalDate.of(2014, 2, 5));
    LocalDate mostRecent = max(dates); //compiler error
}
```

- To ensure that the type `T` extends `Comparable<S>` for any supertype of `T`, we can use `? super`

```
public static <T extends Comparable<? super T>> T max(List<T> list) {
    T max = list.get(0);
    for(T i : list) {
        if(i.compareTo(max) > 0) {
            max = i;
        }
    }
    return max;
}
```

Using this version eliminates the earlier compiler error.

When You Need to Do Both Put and Get

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(nms, n);
        return sum(nums);
}
```

The collection is passed to both `sum` and `count`, so its element type must both extend `Number` (because of `sum`) and be a superclass of `Integer` (because of the `count` method). The only two classes that satisfy both requirements are `Number` and `Integer`. In this code, `Number` was chosen.

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

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<>();
ints.add(1);
ints.add(2);
List<? extends Number> nums = ints;
nums.add(null);  //OK
System.out.println(nums.toString()); //output:  [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<? super Integer> list = new ArrayList<>();
list.add(1);
list.add(2);
Object ob = list.get(0);
System.out.println(ob.toString());   //output: 1
```

Main Point 2

The Get and Put Rule describes conditions under which a parametrized type should be used only for reading elements (when using a list of type ? extends T), other conditions under which the parametrized type should be used only for inserting elements (when using a list of type ? super T), and still other conditions under which the parametrized type can do both (when no wildcard is used). The Get and Put principle brings to light the fundamental dynamics of existence: there is dynamism (corresponding to Put); there is silence (corresponding to Get) and there is wholeness, which unifies these two opposing natures (corresponding to Both).

Unbounded Wildcard, Wildcard Capture, Helper Methods

- The wildcard ?, without the super or extends qualifier, is called the *unbounded wildcard*. This is called *unknown type*.
- Collection<?> is an abbreviation for
  Collection<? extends Object>
- Collection<?> is the supertype of all parametrized type Collections.
- There are two scenarios where an unbounded wildcard is a useful approach:
  - If you are writing a method that can be implemented using functionality provided in the Object class.
  - When the code is using methods in the generic class that don't depend on the type parameter. For example, List.size or List.clear

- Important application of the unbounded wildcard involves *wildcard capture:*

Example Try to copy the 0^{th} 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.

Exercise 11.4

The following code attempts to reverse the elements of a generic list, but the code does not compile. Fix the code by creating a helper method that captures the wildcard. Startup code is in the lesson11.exercise_4 package of the InClassExercises project. Test your code with the main method that has been provided.

```
public static void reverse(List<?> list) {
   List<Object> tmp = new ArrayList<Object>(list);
   for (int i = 0; i < list.size(); i++) {
      list.set(i, tmp.get(list.size()-i-1));
   }
}
```

Lesson Outline

1. Introduction to generics
2. Generic methods
3. Wildcards
4. Understanding Common Generic Signatures
5. Generic programming with generics

Understanding Common Generic Signatures: forEach

- The new default forEach method in Iterable has the following declaration:

void forEach(Consumer<? super T> action)

- Here, the type T signifies the type of the collection elements under consideration. The bounded wildcard indicates that forEach can accept a Consumer type that is a supertype of the particular Collection type T.
- Here is an example:

```
public class ForEach {
    @SuppressWarnings("rawtypes")
    public static void main(String[] args) {
        List<Comparable> nonNullComparables = new ArrayList<>();
        List<Integer> ints = Arrays.asList(1, 2, 3);
        List<String> strings = Arrays.asList("A", "B", "C");
        //The Consumer type here must be Comparable, but T is Integer
        //or String
        ints.forEach(x -> {if(x != null) nonNullComparables.add(x);});
        strings.forEach(x -> {if(x != null) nonNullComparables.add(x);});
    }
}
```

- Demo: lesson11.lecture.generics.signatures

Understanding Common Generic Signatures: forEach (cont.)

- In more detail: When we add `ints` to my list of `Comparables`, using `forEach`, we use a `Consumer<Comparable>` (which IS-A `Consumer<? super Integer>`) as an argument to `forEach` as it traverses a `List<Integer>`.
- In the second use of `forEach`, we again use a `Consumer<Comparable>` (which IS-A `Consumer<? super String>`) as an argument to `forEach` as it traverses a `List<String>`.

Understanding Common Generic Signatures: filter

The filter method on a `Stream<T>` has this signature:

`Stream<T> filter(Predicate<? super T> predicate)`

This means that tests that are made on the elements of the `Stream` can be based on relationships in a supertype of T. Here is an example:

```
static Employee employeeOfTheYear = new Employee("Brian", 100000, 2004, 2, 17);
public static void main(String[] args) {
    List<Manager> managers = Arrays.asList(new Manager("Bob", 100000, 2001, 1, 10),
            new Manager("Rich", 110000, 2002, 3, 15),
            new Manager("Tom", 130000, 2011, 8, 20),
            new Manager("Dennis", 200000, 1991, 11, 8));

    //find the managers from the list who are similar to the employee of the year
    List<Manager> similarTo =   //the Predicate is of type Employee but stream is of type Manager
            managers.stream().filter((Employee e) -> e.isSimilarTo(employeeOfTheYear))
                             .collect(Collectors.toList());
    System.out.println(similarTo);
}
```

Exercise 11.5

Re-implement the `Predicate` that is used in the `filter` method for this code by creating a nested class `MyPredicate`. Startup code is in the lesson11.exercise_5 package of the InClassExercises project.

```
static Employee employeeOfTheYear = new Employee("Brian", 100000, 2004, 2, 17);
public static void main(String[] args) {
    List<Manager> managers = Arrays.asList(new Manager("Bob", 100000, 2001, 1, 10),
            new Manager("Rich", 110000, 2002, 3, 15),
            new Manager("Tom", 130000, 2011, 8, 20),
            new Manager("Dennis", 200000, 1991, 11, 8));

    //find the managers from the list who are similar to the employee of the year
    List<Manager> similarTo =   //the Predicate is of type Employee but stream is of type Manager
            managers.stream().filter((Employee e) -> e.isSimilarTo(employeeOfTheYear))
                             .collect(Collectors.toList());
    System.out.println(similarTo);
}
```

Solution

```
public class Filter {
    static Employee employeeOfTheYear = new Employee("Brian", 100000, 2004, 2, 17);
    public static void main(String[] args) {
        List<Manager> managers = Arrays.asList(new Manager("Bob", 100000, 2001, 1, 10),
                new Manager("Rich", 110000, 2002, 3, 15),
                new Manager("Tom", 130000, 2011, 8, 20),
                new Manager("Dennis", 200000, 1991, 11, 8));

        //find the managers from the list who are similar to the employee of the year
        List<Manager> similarTo =   //the Predicate is of type Employee but stream is of type Manager
                managers.stream().filter(new MyPredicate())
                                 .collect(Collectors.toList());
        System.out.println(similarTo);
    }


    static class MyPredicate implements Predicate<Employee> {
        public boolean test(Employee e) {
            return e.isSimilarTo(employeeOfTheYear);
        }
    }
}
```

Understanding Common Generic Signatures: map

The map operation on Stream<T> has the following signature.

```
Stream<R> map(Function<? super T,? extends R> mapper)
```

This means that the type the map is transforming can be a supertype of the type of the list or collection that is being traversed, and that the type the map sends to can be a subtype of the expected return type.

Map: Example 1

```
Stream<R> map(Function<? super T,? extends R> mapper)
```

```
void example1() {
    List<String> words = Arrays.asList("dog", "elephant", "peacock");

    //T = String (type of Stream for words.stream())
    //R = Number  (since numberStream is of type Stream<Number>)
    //Note: w.length() is a subtype of Number
    Stream<Number> numberStream =
        words.stream().map(w -> w.length());
}
```

Map: Example 2

```
Stream<R> map(Function<? super T,? extends R> mapper)
```

```
void example2() {
    List<Manager> mans = Arrays.asList(
            new Manager("John", 100000, 2000, 10, 15),
            new Manager("Steve", 120000, 1998, 2, 17));

    //T = Manager (mouseover stream())
    //R = Number (mouseover map)
    //Note Employee is a supertype of T, getSalary is a subtype of R
    Stream<Number> stream = mans.stream().map((Employee e) -> e.getSalary());
}
```

Lesson Outline

1. Introduction to generics
2. Generic methods
3. Wildcards
4. Understanding Common Generic Signatures
5. Generic programming with generics

Generic Programming Using Generics

1. Generic programming is the technique of implementing a procedure so that it can accommodate the broadest possible range of inputs.
2. For instance, we have considered several implementations of a max function. The goal of generic programming in this case is to provide the most general possible max implementation.
3. See demos lecture.generics.max.BoundedTypeVariable and lecture.generics.max.BoundedTypeVariable2 for a development of examples leading to the most general possible version.

Connecting the Parts of Knowledge With the Wholeness of Knowledge

Generic Programming Using Java's Generic Methods

1. Using the raw Lists of pre-Java 1.5, one can accomplish the generic programming task of swapping two elements in an arbitrary list using the signature void swap(List, int pos1, int pos2). Using this swap method requires the programmer to recall the component types of the List, and there are no type checks by the compiler.
2. Using generic Lists of Java 1.5 and the technique of wildcard capture, it is possible to swap elements of an arbitrary List with compiler support for type-checking, using the following signature:

```
<T> void swap(List<?> list, int pos1, int pos2)
```

---

3. *Transcendental Consciousness* is the universal value of the field of consciousness present at every point in creation.
4. *Impulses Within the Transcendental Field*. The presence of the transcendental level of consciousness within every point of existence makes individual expressions in the manifest field as rich, unique, and diversified as possible.
5. *Wholeness Moving Within Itself.* In Unity Consciousness, life is appreciated in the fullest possible way because the source of both unity and diversity have become a living reality.