

Lesson - 10

Generics

Weaving the Universal into the Fabric of the Particular

Wholeness of the Lesson

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

- Generics let you write true polymorphic code, which is code that works with any type.
- 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.

Example: // Pre JDK1.5

```
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:

Example :

```
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!
```

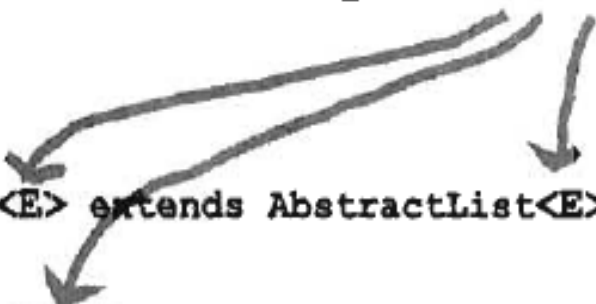
Using type parameters with ArrayList

THIS code:

```
ArrayList<String> thisList = new ArrayList<String>
```

Means ArrayList:

```
public class ArrayList<E> extends AbstractList<E> ... {  
  
    public boolean add(E o)  
    // more code  
}
```



Is treated by the compiler as:

```
public class ArrayList<String> extends AbstractList<String>... {  
  
    public boolean add(String o)  
    // more code  
}
```

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 ArrayList();
```

```
myList.add("Tom");
```

```
myList.add("Bob");
```

```
...
```

```
Employee tom = (Employee)myList.get(0); //no compiler check to prevent  
this
```

Cont...

2. *Elimination of casts.* 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.

```
ClosedCurve[] closedCurves = //...populate with Triangles  
and Rectangles
```

```
if(closedCurves[0] instanceof Triangle)  
    print( (Triangle)closedCurve[0].area());
```

- else
- print((Rectangle)closedCurve[0].area())

Cont...

3. Supports creation of cleaner 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);  
}
```

```
public static void main(String[] args) {  
    Swap s = new Swap();  
    List list1 = new ArrayList();  
    list1.add("hello");  
    list1.add("goodbye");  
    s.swapFirstLast(list1);  
    String newPositionZero = (String)list1.get(0);  
    System.out.println(newPositionZero);  
}
```

```
public static void main(String[] args) {  
    Swap s = new Swap();  
    List<String> list2 = new ArrayList<>();  
    list2.add("hello");  
    list2.add("goodbye");  
    s.swapFirstLast(list2);  
    String newPositionZero2 = list2.get(0);  
    System.out.println(newPositionZero2);  
}
```


Generics Terminology and Naming Conventions

- ```
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!
```

- In the `List<String>` example mentioned earlier:

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*.

The declaration - `List<String> words;` // Parameterized type

- is called a *generic type invocation*, `String` is (in this context) a *type argument*, and `List<String>` is called a *parameterized type*. Also, the class `List`, with the type argument removed, is called a *raw type*. `List words;` // raw type
- Note: When raw types are used where a parameterized type is expected, the compiler issues a warning because the compile-time checks that can usually be done with parameterized types cannot be done with a raw type.

- 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

```
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:

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 type variables may be realized as any Java object type (even user-defined), but not as a primitive type.

## Usage Example:

```
SimplePair<String,String> pair = new SimplePair<>("Hello", "World");
```

```
String hello = pair.getKey(); //hello contains the String "Hello"
```

# Implementing a Generic Interface, Extending a Generic Class

## 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;
 }
}

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

The same points apply for extending a generic class

See Demo: [lesson11.lecture.generics.pairexamples](#)

## Another way: Create a generic class implementation

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

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; }
}

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

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*

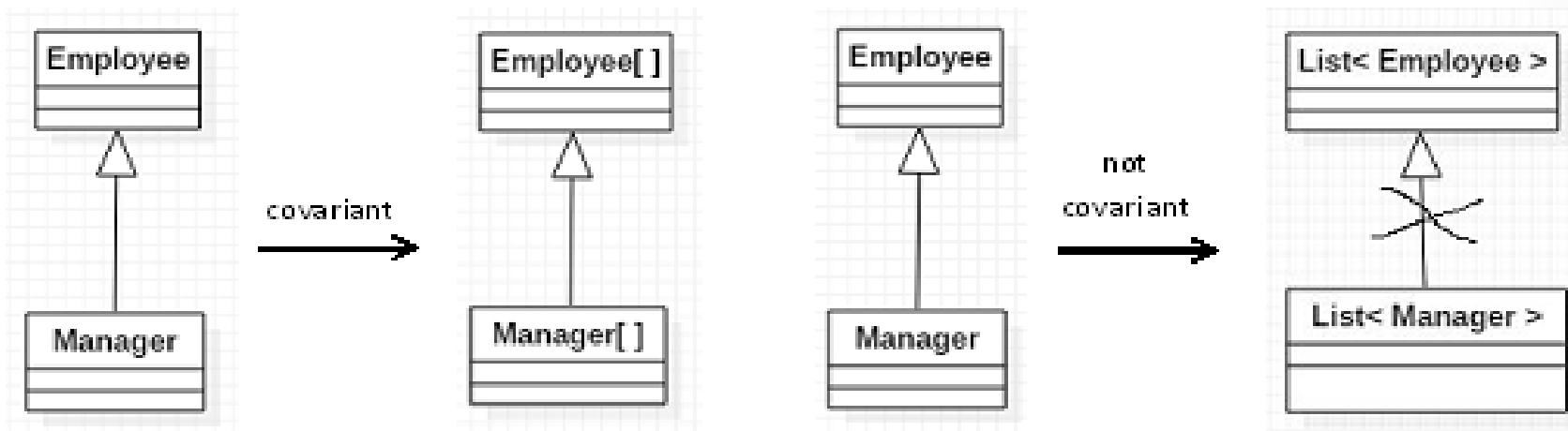
- Java is said to implement generics *by erasure* because the parameterized types like `List<String>`, `List<Integer>` and `List<List<Integer>>` are all represented at runtime by the single type `List`.
- Also *erasure* is the process of converting the first piece(a) of code to the second(b)
- The compiled code for generics will carry out the same down casting as was required in pre-generics Java.
- Support backwards compatibility with older versions

# The Downside(Restriction) of Java's Implementation of Generics

- **Rule 1 : Generic Subtyping Is Not Covariant.**
- For example: `ArrayList<Manager>` is not a subclass of `ArrayList<Employee>` (this is different from arrays: `Manager[]` is a subclass of `Employee[]`: *Array subtyping is covariant.*)

- Example

```
List<Integer> ints = new ArrayList<Integer>();
ints.add(1);
ints.add(2);
List<Number> nums = ints; //compiler error
```



- **Rule 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). Not allowed to create a new array of objects of the unknown type T.


```
T[] arr = new T[5]; // Compilation Error
```

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

Example:

```
class NoGenericType {
 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;
 }
}
```

- **Rule 3 : You cannot plug in a primitive type for a type parameter, instead it must be a reference type.**
- For example, if you want Pair<int>, you should use Pair<Integer>.

- 
- **Rule 4 :** *Component type of an array is not allowed to be a parameterized type.*
  - For example: you cannot create an array like this:
  - `List<String>[] = new List<String>[5];`
  - `Pair<String>[] a = new Pair<String>[10]; // compiling error`

Example :

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



## Reifiable Types & Non-Reifiable Types

The reason for rule (4) 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*.

# 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);
```

- Util is a class have static method compare, just we invoked to compare.
- The generic type 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();
 }
}
```

# Exercise

- Write a generic method `countOccurrences` that counts the number of occurrences of a target object of type `T` in an array of type `T[]`. (You may assume that “equals” comparisons provide an accurate count of occurrences. You may also assume that if the target object is null, we will count the number of nulls that occur in the array.)
- We start with the simple case of an array of `Strings` (shown below). Now how can this method be generalized to arbitrary types?
- See demo `lesson11.lecture.generics.countoccurrences`

```
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;
}
```

## Example: Finding the max

**Problem:** Find the max value in a List.

**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;
}
```

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

```
public static <T> T max(List<T> list) {
 Comparable<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:** Use the extends keyword, creating a *bounded type variable*

```
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;
}
```

# Main Point

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.

# Finding the max (continued)

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:

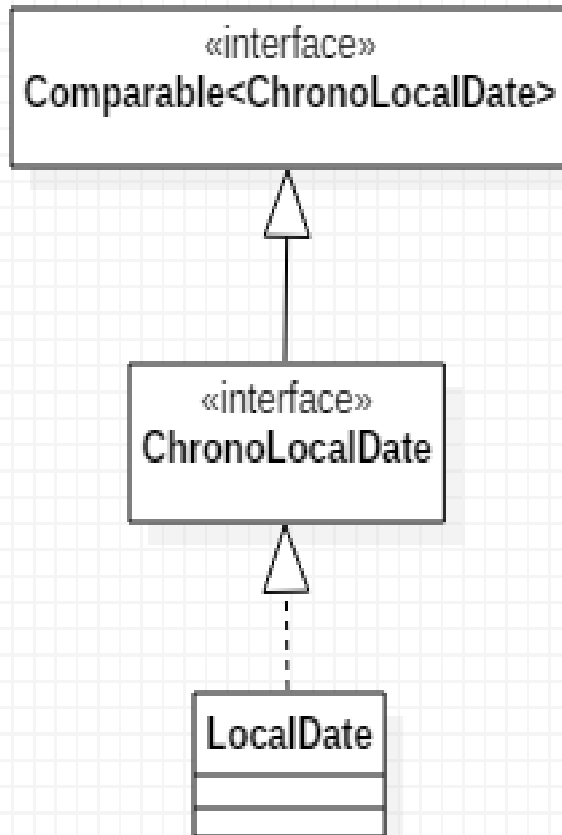
```
public static <T extends Comparable<T>> T max(List<T> list) {
 T max = list.get(0);
 for(T i : list) {
 if(i.compareTo(max) > 0) {
 max = i;
 }
 }
 return max;
}
```

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
}
```



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



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

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

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

# GENERIC WILDCARDS

- The symbol ? can be used as a wildcard, in place of a generic variable. It stands for “unknown type,” and is called the *wildcard type*.
- Example :

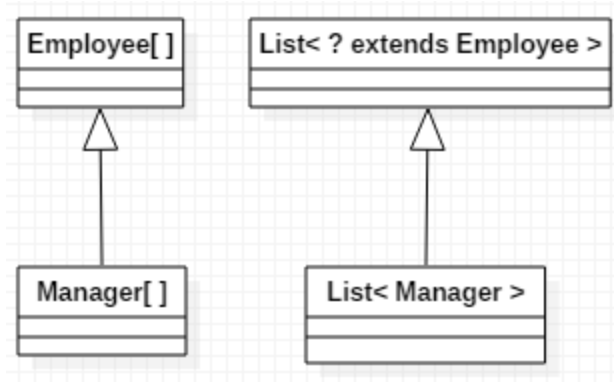
```
static void display(Collection<?> c) {
 for (Object o : c) {
 System.out.printf("%s ", o);
 }
 System.out.println();
}
```

# The ? With Bounded Wildcard

- an *upper bounded* wildcard restricts the unknown type to be a specific type or a subtype of that type and is represented using the extends keyword.
- To declare an upper-bounded wildcard, use the wildcard character ('?'), followed by the extends keyword, followed by its *upper bound* : `<? extends superclass>`
- In a similar way, a *lower bounded* wildcard restricts the unknown type to be a specific type or a *super type* of that type.
- A lower bounded wildcard is expressed using the wildcard character ('?'), following by the super keyword, followed by its *lower bound*: `<? super subclass>`.

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

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 method had been declared like this:

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

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

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

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

```
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 dbls.

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:

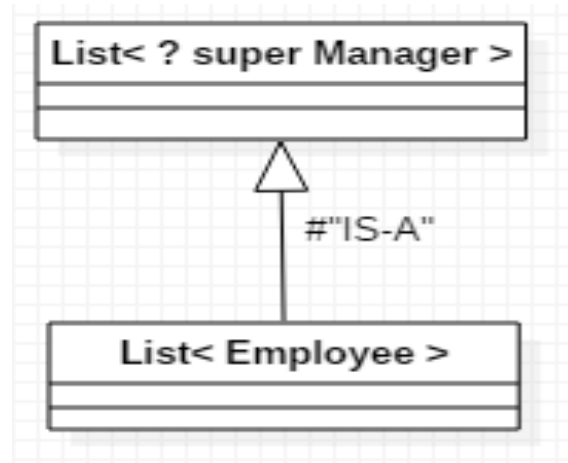
- Recall the addAll method from Collection:

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

- The following produces a compiler error:
- ```
List<Integer> ints = new ArrayList();
ints.add(1);
ints.add(2);
List<? extends Number> nums = ints;
nums.add(3.14); //compiler error – it does not allow any input except null, ? – unknown, add
method allow for specific type, not for unknown type
System.out.println(ints.toString()); //output: [1, 2]
nums.add(null); //OK – see below
```
- 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 super type 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 super type of `Manager`. Therefore, a `List<Employee>` IS-A `List<? super Manager>`.



# Find Max Solution

//fix by expanding the range of type arguments for Comparable, like this:

```
public static <T extends Comparable<? super T>> T
max2(List<T> list) {
 T max = list.get(0);
 for(T i : list) {
 if(i.compareTo(max) > 0) {
 max = i;
 }
 }
 return max;
}
```

# 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:

// You can get the value and print. Not assign to other variable

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

//output: 5

- 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 super type 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 - see below
```

# 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**.
- Example1. 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 List<Integer>, List<Double> are subtypes of Collection<? extends Number>, the following are legal:

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

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

-

- Example2 (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 the destination.
- *Sample usage:*

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

Example 3 : (using ? super) Whenever you use the add method for a Collection, you are inserting values, and so ? super should be used. This code inserts the elements into the list from 0 to n .

```
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]
```

- 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.

# 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(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<>();
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

The Get and Put Rule describes conditions under which a parametrized type should be used only for reading elements (when using a list is of type  $? \text{ extends } T$ ), other conditions under which the parametrized type should be used only for inserting elements (when using a list of type  $? \text{ super } T$ ), 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

1. The wildcard `?`, without the `super` or `extends` qualifier, is called the *unbounded wildcard*.
2. Important application of the unbounded wildcard involves *wildcard capture*: In some cases, the compiler infers the type of a wildcard. For example, a list may be defined as `List<?>` but, when evaluating an expression, the compiler infers a particular type from the code. This scenario is known as *wildcard capture*.

Example: Try to copy the 0<sup>th</sup> element of a general list to the end of the list

```
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 uses wildcard. You can fix it by writing a *private helper method* which captures the 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);
}
```

Note:

- 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.

# Guidelines for Wildcard Use

- Unbounded Wild Card ( ? )
- The unbounded wildcard type is specified using the wildcard character (?), for example, `List<?>`. This is called a *list of unknown type*. There are two scenarios where an unbounded wildcard is a useful approach:
  1. If you are writing a method that can be implemented using functionality provided in the `Object` class.
  2. 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`. In fact, `Class<?>` is so often used because most of the methods in `Class<T>` do not depend on `T`.

- One of the more confusing aspects when learning to program with generics is determining when to use an upper bounded wildcard and when to use a lower bounded wildcard. This page provides some guidelines to follow when designing your code. For purposes of this discussion, it is helpful to think of variables as providing one of two functions:
  - **An "In" Variable:** An "in" variable serves up data to the code. Imagine a copy method with two arguments: `copy(src, dest)`. The `src` argument provides the data to be copied, so it is the "in" parameter.
  - **An "Out" Variable:** An "out" variable holds data for use elsewhere. In the copy example, `copy(src, dest)`, the `dest` argument accepts data, so it is the "out" parameter.
  - Of course, some variables are used both for "in" and "out" purposes — this scenario is also addressed in the guidelines.
- You can use the "in" and "out" principle when deciding whether to use a wildcard and what type of wildcard is appropriate.
- The following list provides the guidelines to follow:
  - An "in" variable is defined with an upper bounded wildcard, using the `extends` keyword.
  - An "out" variable is defined with a lower bounded wildcard, using the `super` keyword.
  - In the case where the "in" variable can be accessed using methods defined in the `Object` class, use an unbounded wildcard.
  - In the case where the code needs to access the variable as both an "in" and an "out" variable, do not use a wildcard.

# Common Generic Signature of API

```
Person.persons().stream()
```

```
.filter(p -> p.getIncome() > 5000.0 && p.getGender()==Gender.MALE)
```

```
.map(Person::getName).forEach(System.out::println);
```

*filter accepts Predicate and the signature from API is :*

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

*signature for the given example is :*

```
Stream<Person> java.util.stream.Stream.filter(Predicate<? super Person>
```

Here ? Should be a same person type or super type of person

*map accepts Function and the signature from API is :*

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

*signature for the given example is :*

```
<String> Stream<String> java.util.stream.Stream.map(Function<? super Person,
? extends String> mapper)
```

*forEach accepts Consumer and the signature from API is :*

```
void forEach(Consumer<? super T> action);
```

*signature for the given example is :*

```
void java.util.stream.Stream.forEach(Consumer<? super String> action)
```

# 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: 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 super type 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.
- Here is an example of both of these situations:

```
public static void main(String[] args) {
 List<Double> someDoubles = Arrays.asList(2.3, 3.5, 6.8);
 List<String> words = Arrays.asList("dog", "elephant", "peacock");
 List<Manager> mans = Arrays.asList(
 new Manager("John", 100000, 2000, 10, 15),
 new Manager("Steve", 120000, 1998, 2, 17));
 List<Number> numbers =
 //here, type R is Number and word.length() is of type Integer
 words.stream().map(word -> word.length())
 .collect(Collectors.toList());
 numbers.addAll(someDoubles);
 //here, type T is Manager, and Employee is supertype
 numbers.addAll(mans.stream().map((Employee e) -> e.getSalary())
 .collect(Collectors.toList()));
 System.out.println(numbers);
}
```

# Generic Programming Using Generics

- Generic programming is the technique of implementing a procedure so that it can accommodate the broadest possible range of inputs.
- See demo [lesson11.lecture.generics.genericprogrammingmax](#) for a development of examples leading to the most general possible version.



# CONNECTING THE PARTS OF KNOWLEDGE WITH THE WHOLENESS OF KNOWLEDGE

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.

