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

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.

Example:

```
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);
```

   With Generics:

```
//compiler error to cast String to Employee
List<String> list = new ArrayList<>();
list.add("mike");
list.add("andy");
Employee e = (Employee)list.get(0);
```

2. *Elimination of casts*. Downcasting is considered an "anti-pattern" in OO programming. Typically, downcasting should not be necessary (though there are some 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())
```

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

Example *Task*: get the max element in a list (*generic methods* discussed in upcoming slide)

```
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 List<String> example mentioned earlier:

```
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;
```

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.

```
List list = new ArrayList<>();
list.add("mike");
list.add("andy");
```

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

```
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.
2. 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

```
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 class implementation

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

Or: Create a parametrized type implementation

```
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);
System.out.print(s);  //output: 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));
System.out.print(s);  //output: Hello world!
```

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

The Downside of Java's Implementation of Generics

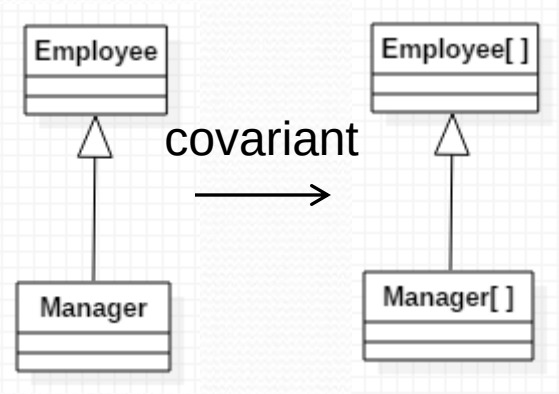
1. *Generic Subtyping Is Not Covariant.* For example: `ArrayList<Manager>` is not a subclass of `ArrayList<Employee>` (this is different from arrays: `Manager[]` is a subclass of `Employee[]`: *Array subtyping is covariant.*)

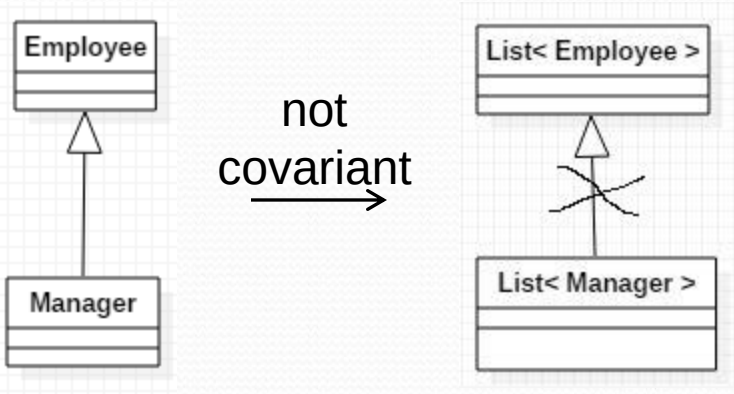
   Covariant: if they have the same subtype-supertype relationship as their type parameters.

   Example: If generic subtyping *were* covariant, there would be unfortunate consequences:

```
List<Integer> ints = new ArrayList<Integer>();
ints.add(1);
ints.add(2);
List<Number> nums = ints;        //compiler error
nums.add(3.14);
System.out.print(ints);   //output: [1, 2, 3.14]
```

The Downside of Java's Implementation of Generics (cont.)





The Downside of Java's Implementation of Generics (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 = new T[5];
```

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;
    }
}
```

The Downside of Java's Implementation of Generics (cont.)

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

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

Example: Anything wrong with this code?

```
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};
    }
 }
```

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.
- The syntax for a generic method includes a type parameter, inside angle brackets, and appears right 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);
```

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);
```

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. Method signature provided below:

```
public static int countOccurrences(String[] arr, String target)
```

Now how can this method be generalized to arbitrary types?

See demo lesson10.lecture.generics.countoccurrences

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: 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;
}
```

Demo: lesson10.lecture.generics.max

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.

Finding the max (cont.)

- 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:

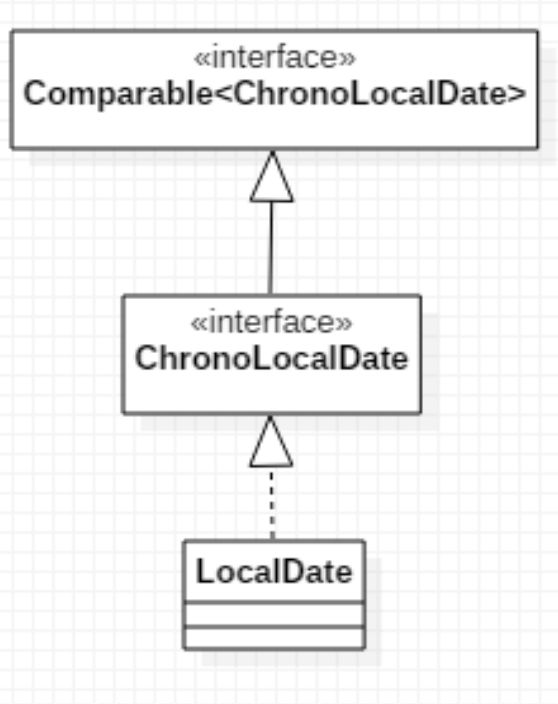
```
public static <T extends Comparable<T>> T max(List<T> list) {
    T max = list.get(0);
    for(T i : list) {
        if(i.compareTo(max) > 0) {
            max = i;
        }
    }
    return max;
}
```

- 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:



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`.

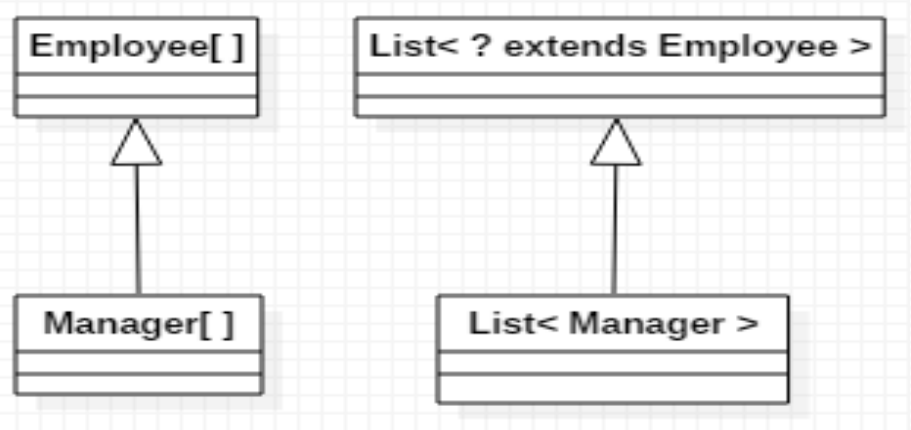
The answer lies in the use of *bounded wildcards.*

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 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 <u>does</u> work:

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

(See demo lesson10.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
```

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<Integer>();
ints.add(1);
ints.add(2);
List<? extends Number> nums = ints;
nums.add(3.14);          //compiler error
System.out.println(ints.toString()); //if add is allowed, output?
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:

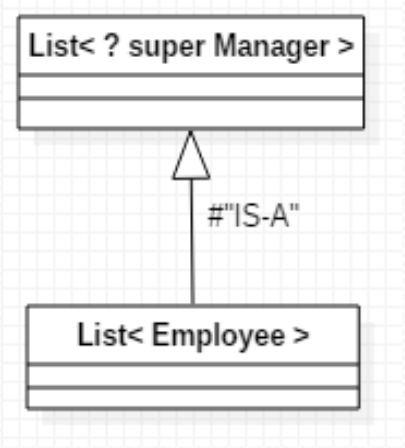
remove, removeAll, retainAll

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>`.

Applications of the ? super Wildcard

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.

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 erro
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 us a wildcard at all when you *both get and put* values.

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
```

Improving max() using bounded wildcards

We saw before that the following 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
}
```

We can ensure that the type T extends `Comparable<S>` for any supertype of T (which, as we saw before, is what is needed here) 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.

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 is 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

1. The wildcard ?, without the super or extends qualifier, is called the *unbounded wildcard*.
2. Collection<?> is an abbreviation for Collection<? extends Object>
3. Collection<?> is the supertype of all parametrized type Collections.

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: 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);
}
```

It may be helpful in this case to write the Predicate as an inner class, to see what is going on. In this example, T is `Manager` (since that's the type of the List we are starting with) and `Employee` is a supertype of T. (So, `Predicate<Employee>` IS-A `Predicate<? super Manager>`.)

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

Understanding Common Generic Signatures: from Stream API

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

Stream<T> sorted(Comparator<? super T> comparator);

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

public static <T> Stream<T> concat(Stream<? extends T>
a, Stream<? extends T> b)
```

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 demo lecture.generics.max.BoundedTypeVariable 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.