**Lesson 11  
Java 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

6. You can try to understand the optional parts of this file if you have time (JL).

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

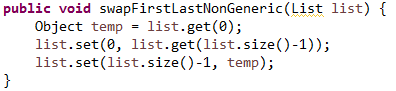
//You will get an IllegalCastException (JL)

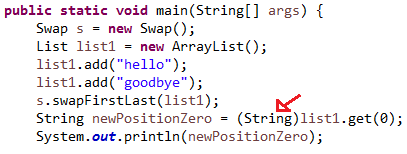
1. *Elimination of casts*.Downcasting is considered an “anti-pattern” in OO programming. Typically, downcasting should not be necessary (though there are plenty of exceptions to this rule); finding the right subtype should be accomplished with late binding.  
     
   Example of bad downcasting.  
   ClosedCurve[] closedCurves = //…populate with Triangles and Rectangles  
   if(closedCurves[0] instanceOf Triangle)   
   print( (Triangle)closedCurve[0].area());

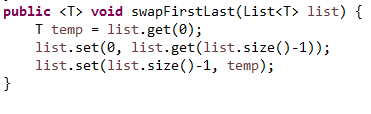
else

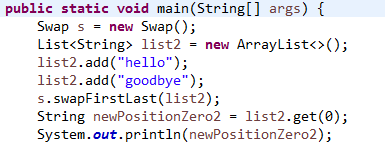
print((Rectangle)closedCurve[0].area())

1. *Supports creation of cleaner generic algorithms.*  
     
   Example *Task*: Swap elements in a list (*generic methods* discussed in upcoming slide)









**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*.**

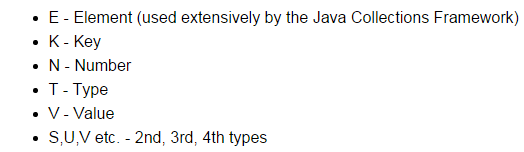
1. The delcaration

List<String> words;

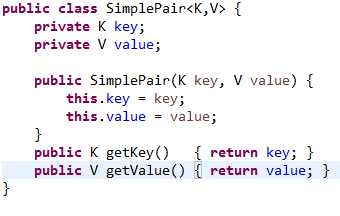
is called a *generic type invocation,* String is (in this context) a ***type argument***, and List<String> is called a ***parametrized type****.*Also, the class List, with the type argument removed, is called a ***raw type****.*Note: When raw types are used where a parametrized type is expected, the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

**USE THE BOARD!**

1. Commonly used type variables:



**Creating Your Own Generic Class**



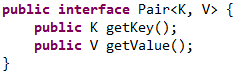
Notes:

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.

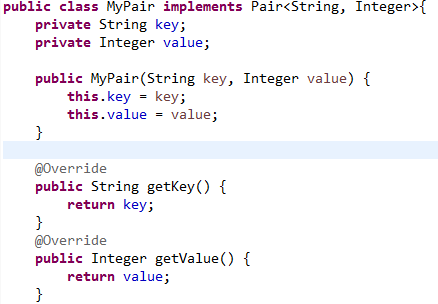
Usage Example:

SimplePair<String,String> pair = new SimplePair<>(“Hello”, “World”);  
 String hello = pair.getKey(); **//hello contains the String “Hello”**

**Implementing a Generic Interface, Extending a Generic Class**

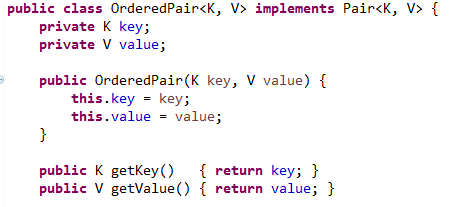


One way: Create a parametrized type implementation Concrete types put in explicitly.



**}**

Another way: Create a **generic class implementation**



See Demo: lesson11.lecture.generics.pairexamples

The same points apply for extending a generic class

Either**:**



Or**:**



**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!

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.

Benefits of this implemenation approach:

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

**The Downside of Java’s Implementation of Generics**

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

**WRITE ON BOARD!!**

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

1. *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;  
    }  
    }
2. *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:

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**

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

Consider the analogous situation with arrays: The following statement

new String[size]

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

new ArrayList<String>()

allocates a list, but does not store in the list any indication of the type of its elements.We say that Java *reifies* array component types but does not reify list element types (or other generic types). In the case of

new List<Integer>[] //not allowed

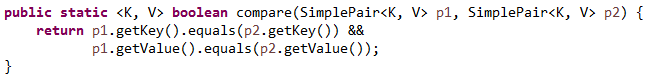
because the **List type does not store component type information**, the resulting array is unable to store component type information (which violates rules for arrays). We say *parametrized types* (as well as type variables) *are not reifiable.*

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

**Generic Methods**

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

The **syntax for a generic method** includes a type parameter, inside angle brackets, and appears before the method's return type. **For static generic methods**, the type parameter section must appear before the method's return type. Note that the following compare method that we wrote returns a boolean.

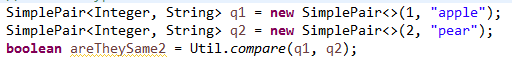


The complete syntax for invoking this method would be:



The generic type can always be inferred by the compiler, and can be left out.

Util must be our class (JL).

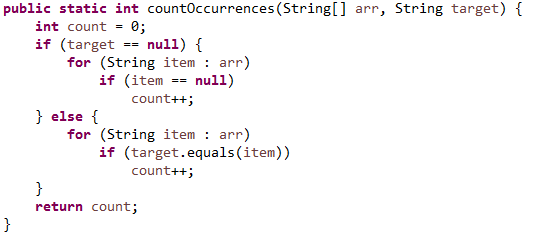


**Exercise SKIP THIS PAGE!**

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.)

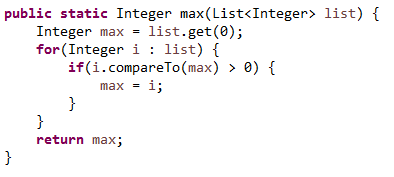
We start with the simple case of an array of Strings (shown below). Now how can this method be generalized to arbitrary types?

See demo lesson11.lecture.generics.countoccurrences

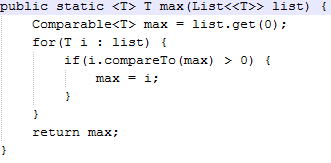


**OPTIONAL PAGE -Example: Finding the max**

**Problem:** Find the max value in a List.  
  
**Easy Case**: First try finding the max of a list of Integers:

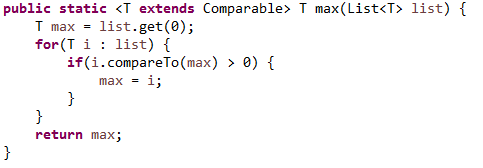


**Try to generalize** to an arbitrary type T (this first try doesn’t quite work…)



**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.*

*Now T must implement a compareTo method (JL).*

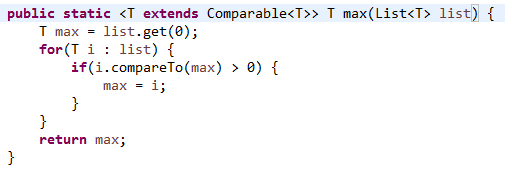


**Main Point**

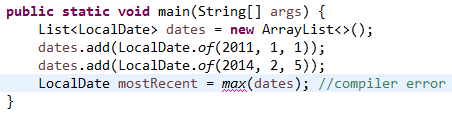
Generic methods make it possible to create general-purpose methods in Java by declaring and using one or more type variables in the method. This allows a user to make use of the method using any data type that is convenient, with full compiler support for type-checking. Likewise, when individual awareness 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.

**OPTIONAL PAGE -Finding the max (continued)**

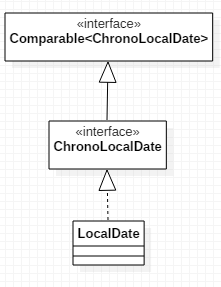
The Comparable interface is also generic. For a given class C, implementing the Comparable interface implies that comparisons will be done between a current instance of C and another instance; the other instance type is the type argument to use with Comparable. For example,   
String implements Comparable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



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



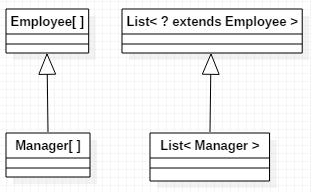
The Problem: LocalDate does not implement Comparable<LocalDate>. Instead, the relationship to Comparableis the following:



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.*

**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****.*
* For any subclass C of Employee, List<C> is a subclass of List<? extends Employee> .
* So, even though the following gives a compiler error:

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

**the following does work**:

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

(See demo lesson11.lecture.generics.extend)

**Applications of the ? extends Wildcard**

The Java Collection interface has an addAll method:

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

The extends wildcard in the definition makes the following possible (since not covariant JL):

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

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);  
 . . .  
}

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, not**

//**covariant**

CoVariant for Lists (JL).

See the demo: lesson11.lecture.generics.addall

**Another Example Using addAll (Do This!)**

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 may be 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 =   
ints.add(1);  
ints.add(2);  
List<? extends Number> nums = ints;  
nums.add(3.14); //**compiler error**System.out.println(ints.toString()); //**output: [1, 2, 3.14]  
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 compiler error above is good since we do Not want to be able to add 3.14 to ints (JL).

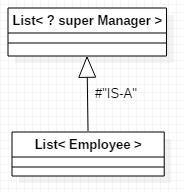
(Write the above line on the board.)

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(NOT adding anything new!)

**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.

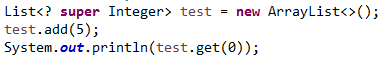
. 

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**. (Writing is easy, reading is inconvenient. JL.)

Example:



//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 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 - see below

**WRITE ON BOARD!**

**PECS - Producer Extends, Consumer Super**

**Producer (Supplier) uses ‘get’, Consumer uses ‘put’**

**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. 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);  
Integer val = sum(ints); //output: 6.0  
  
List<Double> doubles = Arrays.asList(2.78, 3.14);  
Double val = sum(doubles); //output 5.92

Another Example (from the Collections class)

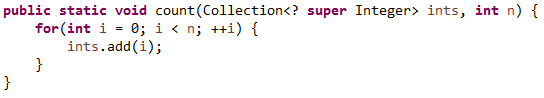
public static <T> void copy(List<? super T> destination, List<? extends T> source) {  
 for(int i = 0; I <source.size(); ++i) {  
 destination.set(i, source.get(i));  
 }  
 }

**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.**

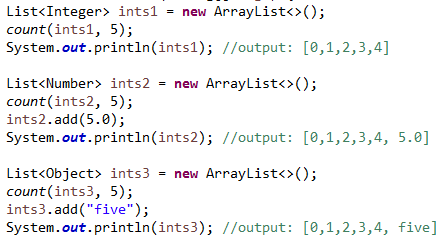
**(If we try it the other way around it does not work, i.e.,** copy(List<? **extends** T> destination, List<? **super** T> source) (JL))

*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) into the **wider type** (Object)  
 System.out.println(objs.toString()); //output: [5, 6, four] (because using a set above JL)

T is Integer here I believe (JL).**Another Example (using ? super)Whenever you use the add method for a Collection, you are inserting values, and so ? supershould be used.**

Example:  
  
 

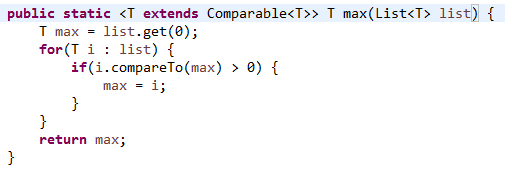
Since super was used, the following are legal:

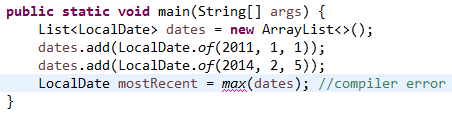


In the second call, ints2 is of type List<Number>which “IS-A” Collection<? super Integer> (since Number is a superclass of Integer), so the count method can be called.  
  
In the third call, ints3 is of type List<Object>which also “IS-A” Collection<? super Integer> (since Object is a superclass of Integer), so the count method can be called here too.

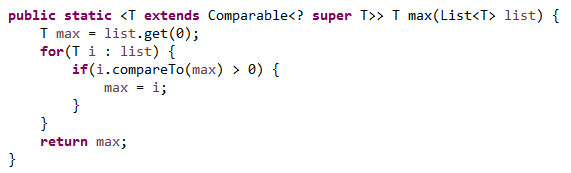
Note that the add methods shown here have nothing to do with the ?super declaration – you can add a double to a List<Number> and a String to a List<Object> for the usual reasons.

Another Example – improving implementation of the max function  
  
We saw before that the following implementation of max was not general enough

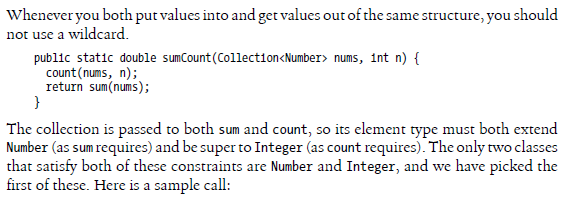
  
We encountered a compiler error here:

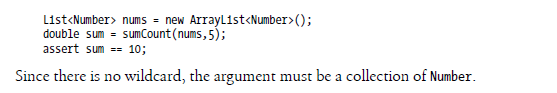


We can ensure that the type T extends Comparable<S> for any supertype of T (which, as we saw before, is what is needed here) we can use ?super

  
Using this version eliminates the earlier compiler error.

**OPTIONAL - When You Need to Do Both Put and Get**





.**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]

1. 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 ? 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**

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.
4. **Important application of the unbounded wildcard involves *wildcard capture****:*

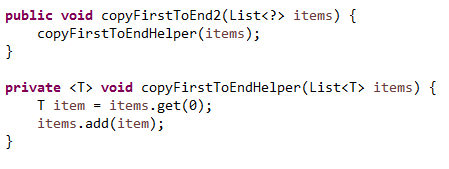
Example: Try to copy the 0th element of a general list to the end of the list

**First Try:**



Compiler error arises because we are trying to add to a List whose type involves the extends wildcard (We are doing an add and a get, JL).

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



Notes:

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

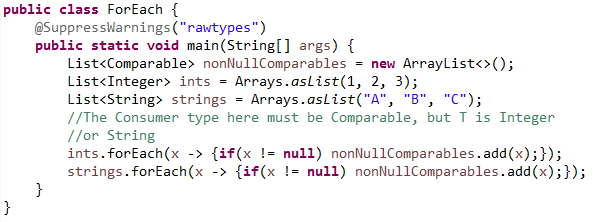
**Understanding Common Generic Signatures: forEach**

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 (Recheck) :



Scenario: I want to arrange integers, strings, and possibly other types of orderable objects into a single list, and arrange them in some order. So we create a List<Comparable> and add a list ints of integers to it, and another list strings of Strings to it, using the forEach method. This is possible because forEach accepts supertypes of the base type. For the ints list, the starting type is Integer; for the strings list, it is String. (**Both have an IS-A relationship with Comparable** - JL)  
  
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>.

The nonNullComparables.add(x) is the operation performed by the ‘accept’ method I believe. So, we have a Consumer<Comparable>. (JL).

Demo: lesson11.lecture.generics.signatures

**Understanding Common Generic Signatures: filter**

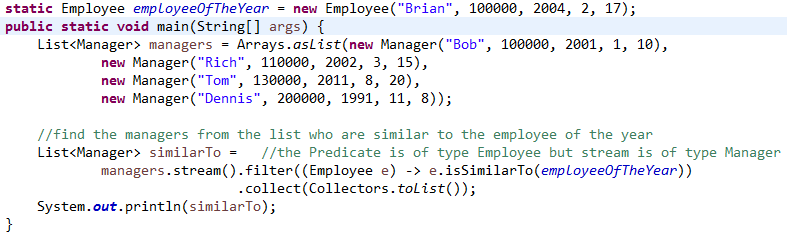
The filter method on a Stream<T> has this signature:

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

This means that tests (test method of Predicate (JL)) that are made on the elements of the Stream can be based on relationships in a supertype of T. Here is an example:

**Look at the below line “the Predicate is of type Employee but stream is of type Manager”.**

**T is a Manager here.**



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);  
 }  
}

**OPTIONAL Until I RECHECK!! - Understanding Common Generic Signatures: map**

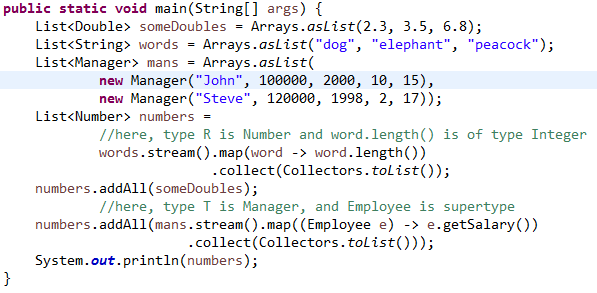
The map operation on Stream<T> has the following signature.

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.**

R - The element type of the new stream (From the Oracle API)!

Here is an example of both of these situations :



**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.

**For Homework Study The Parts of This File That I Emphasized!**

**Do the Threads mini-lesson!**

1. See demo lecture.generics.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, there are no type checks by the compiler. 2. Using generic Lists of Java 1.5 the compiler does automatic type-checking, and so the code is type-safe.   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. | arrow |