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

Background

“In any nontrivial software project, bugs are simply a fact of life. Careful planning, programming, and testing can help reduce their pervasiveness, but somehow, somewhere, they'll always find a way to creep into your code. This becomes especially apparent as new features are introduced and your code base grows in size and complexity.

Fortunately, some bugs are easier to detect than others. Compile-time bugs, for example, can be detected early on; you can use the compiler's error messages to figure out what the problem is and fix it, right then and there. Runtime bugs, however, can be much more problematic; they don't always surface immediately, and when they do, it may be at a point in the program that is far removed from the actual cause of the problem.

Generics add stability to your code by making more of your bugs detectable at compile time…”

- From an Oracle tutorial at https://docs.oracle.com/javase/tutorial/java/generics/

Question: what does it tell us?

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
   ```

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())
   ```

3. *Enabling programmers to implement generic algorithms.*

   By using generics, programmers can implement generic algorithms that work on collections of different types, can be customized, and are type safe and easier to read.

   (There are many people in the industry who believe generics are a mistake:

   - Too many restrictions (https://docs.oracle.com/javase/tutorial/java/generics/restrictions.html)
   - Type erasure has performance issues
   - Too complicated than necessary
   - A lot of frustrations to new programmers)

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*.

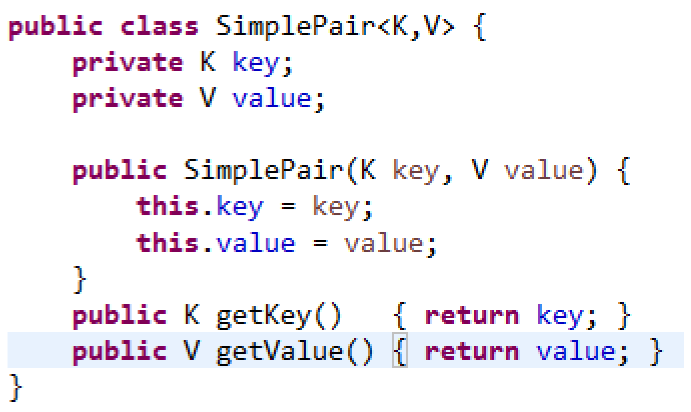
   Note: When raw types are used where a parametrized type is expected, the compiler issues a warning because the compile-time checks that can usually be done with parametrized types cannot be done with a raw type.

3. Commonly used type variables (Type Parameter Naming Conventions):

   - 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

```
/** or you can use javadoc comment to give people a better idea
 * @param <R> - row
 * @param <C> - column
 * @param <E> - cell element
 */
public class GenericTableCell<R, C, E> {

}
```

Creating Your Own Generic Class



```java
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; }
}
```

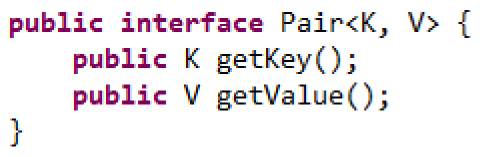
Notes:

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:

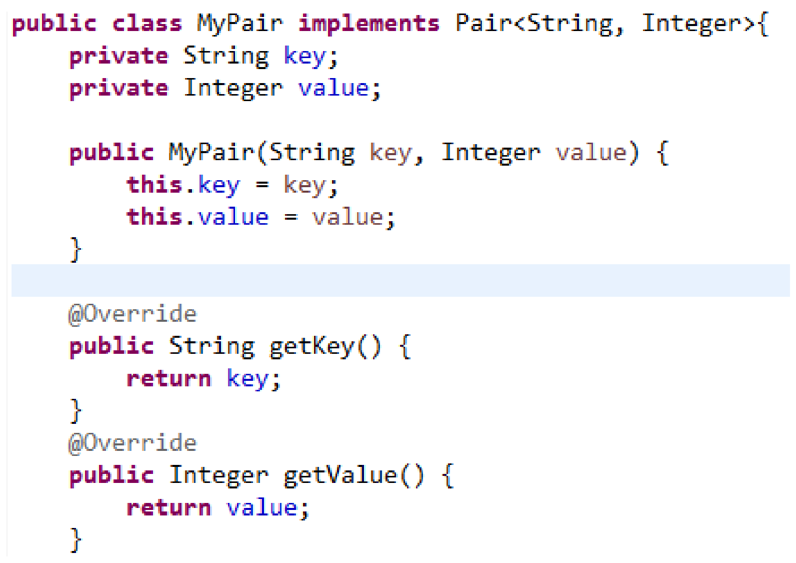
```java
SimplePair<String,String> pair = new SimplePair<>("Hello", "World");
String hello = pair.getKey(); //hello contains the String "Hello"
```

Implementing a Generic Interface, Extending a Generic Class



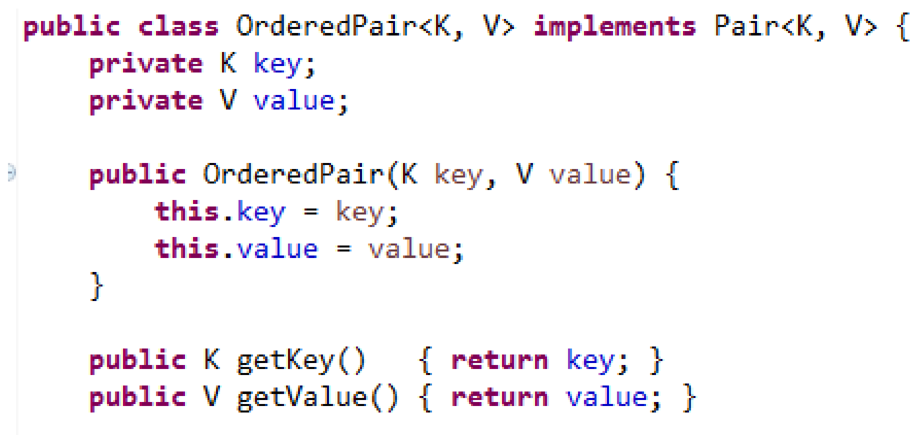
```
public interface Pair<K, V> {
    public K getKey();
    public V getValue();
}
```

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

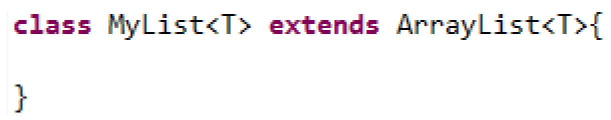


```
public class OrderedPair<K, V> implements Pair<K, V> {
    private K key;
    private V value;

    public OrderedPair(K key, V value) {
        this.key = key;
        this.value = value;
    }

    public K getKey()   { return key; }
    public V getValue() { return value; }
```

See Demo: `lesson11.lecture.generics.pairexamples`

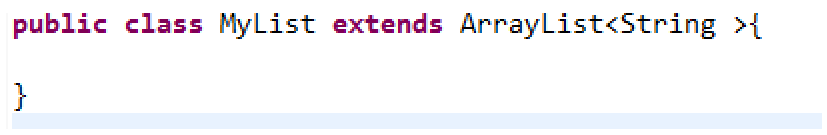
The same points apply for extending a generic class

Either:



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

}
```

Or:



```java
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!
```

1. Java 8 implements generics *by erasure* because the parametrized types like `List<String>`, `List<Integer>` and `List<List<Integer>>` are all represented at runtime by the single type `List`.
2. Java 8 compiler erases generic types by using raw types in bytecode. See the above code.
3. Generic type information is not discarded by the compiler but saved somewhere to let class loaders know it is a generic type.
4. The compiled code for generics will carry out the same downcasting as was required in pre-generics Java, if necessary. (cast insertion)
5. Generate bridge methods to preserve polymorphism in extended generic types.

Benefits of this implemenation approach:

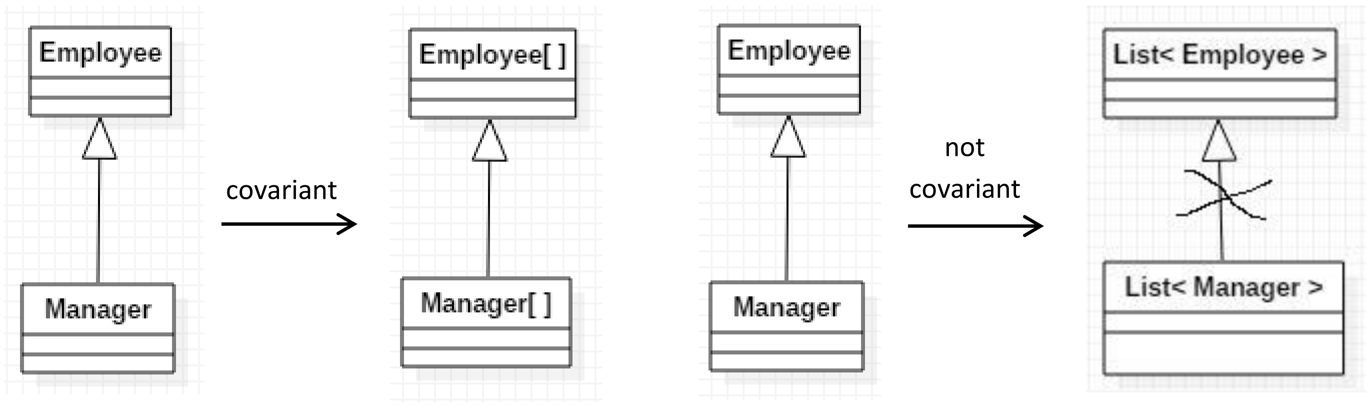
A. No increase in the number of types in the language (in C++, each parametrized type is a genuinely different type)
B. Backwards compatibilty with non-generic code – for instance, in both generic and non-generic code, there is, at runtime, only one type List, so legacy code and generic code can intermingle 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]
```



Optional: (Requires strong mathematical background) The real meaning of "covariant". A mathematical *category* is a collection of objects of the same type together with structure-preserving maps that map one object in the category to another. The *category of sets* has as its objects sets together with functions from one set to another. The collection Class of all classes (say Java classes) also forms a category; in this case, the "maps" between objects of this category are the arrows given by the *subclass relation.* Another category ClassArray is the collection of arrays having component type a Java class, like Employee[], Manager[], etc. Again the "maps" between these objects can be taken to be the subclass relation. The statement "array subtyping is covariant" means, technically speaking, that the transformation F: 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.

2. *Component type of an array may not 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;
   }
}
```

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

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

Example:

```
class 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
```

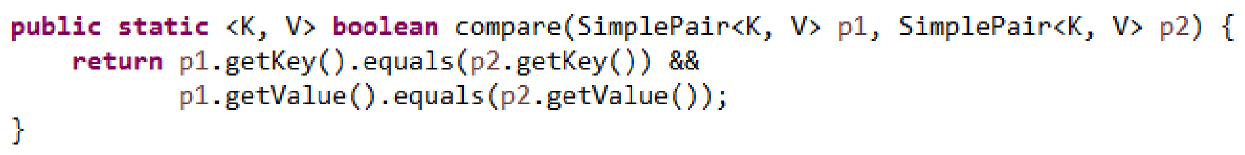
because the List type does not store component type information, the resulting array is unable to store component type information (which violates rules for arrays). We say *parametrized types* (as well as type variables) *are not reifiable.*

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

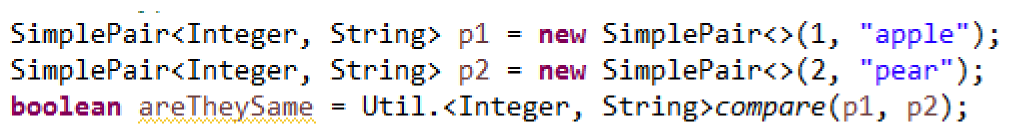
Generic methods are methods that introduce their own type parameters. This is similar to declaring a generic type, but the type parameter's scope is limited to the method where it is declared. Static and non-static generic methods are allowed, as well as generic class constructors.

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.



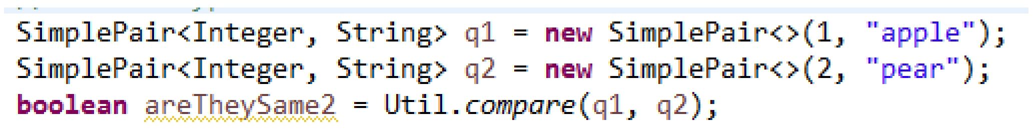
```
public static <K, V> boolean compare(SimplePair<K, V> p1, SimplePair<K, V> p2) {
    return p1.getKey().equals(p2.getKey()) &&
           p1.getValue().equals(p2.getValue());
}
```

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.



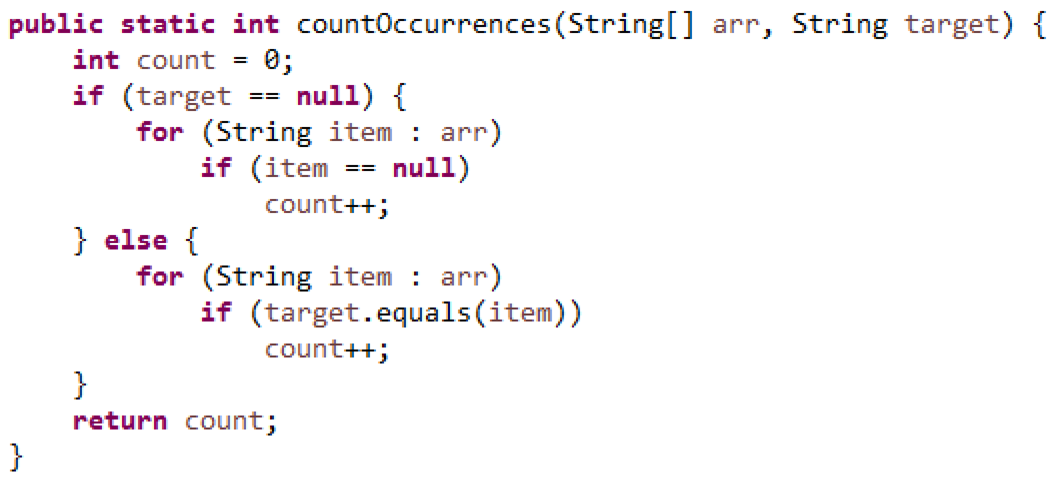
```
SimplePair<Integer, String> q1 = new SimplePair<>(1, "apple");
SimplePair<Integer, String> q2 = new SimplePair<>(2, "pear");
boolean areTheySame2 = Util.compare(q1, q2);
```

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 (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:

```java
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;
        }
    }
}
```

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*

```java
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;
}
```

Finding the max (continued)

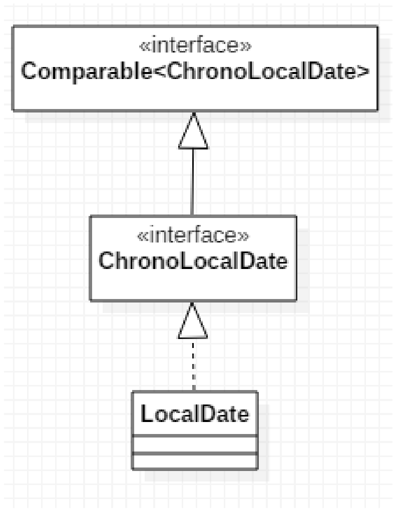
The Comparable interface is also generic. For a given class C, implementing the Comparable interface implies that comparisons will be done between a current instance of C and another instance; the other instance type is the type argument to use with Comparable. For example, String implements Comparable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

```java
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:

```java
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:



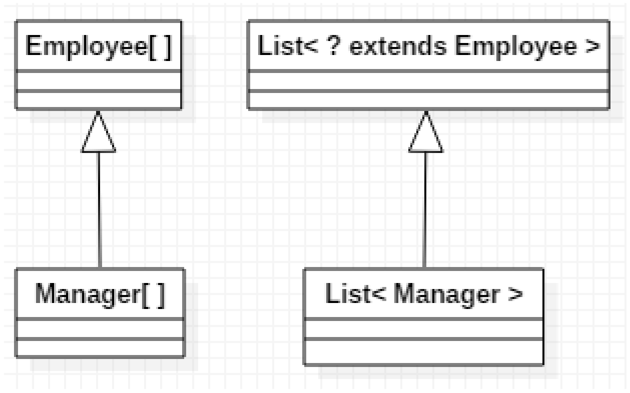
What is needed is a max function that accepts types T that implement not just Comparable<T>, but even Comparable<S> for any supertype of T.

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

One answer is 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:

```
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
```

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*. (Why not?)

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

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

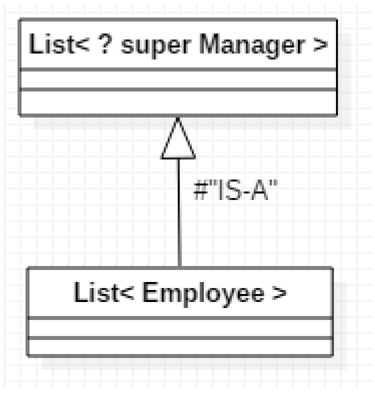
```
        remove, removeAll, retainAll
```

and also execute the static methods from `Collections`:

```
        sort, binarySearch, swap, shuffle
```

The `? super` Bounded Wildcard

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



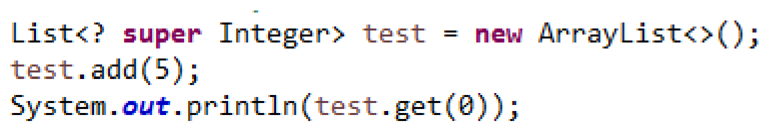
.

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

Limitations of the super Wildcard

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

Example:



```java
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 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  - 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. (simply because type ambiguity arises when you do the opposite operation)

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));
    }
}
```

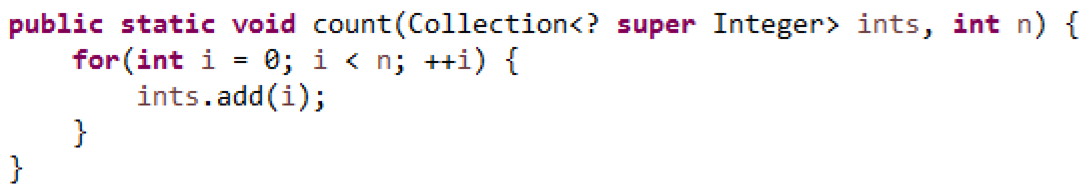
Note that we ***get*** from `source`, which is typed using `extends`, and we ***insert*** into `destination`, which is typed using `super`. It follows that any subtype of T may be *gotten* from `source`, and any supertype of T may be *inserted* into the `destination`.

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]
```

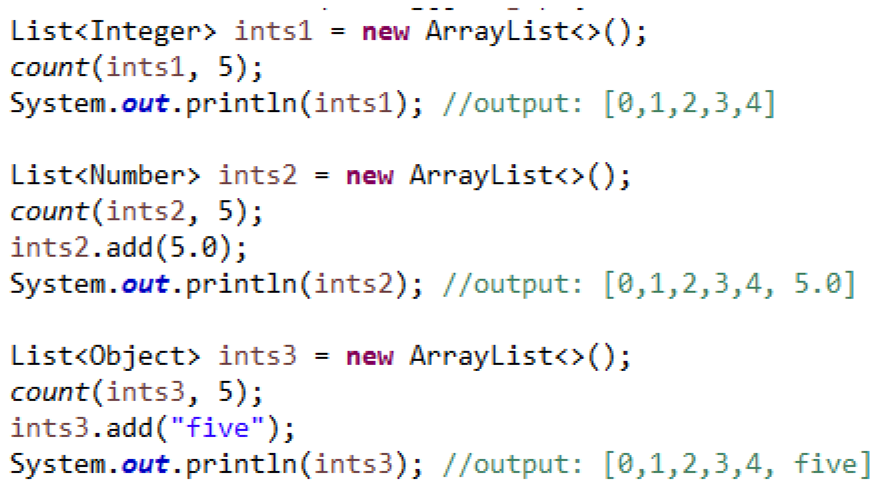
<u>Another Example (using ? super)</u> Whenever you use the `add` method for a `Collection`, you are inserting values, and so `? super` should be used.

Example:



```java
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:



```java
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]
```

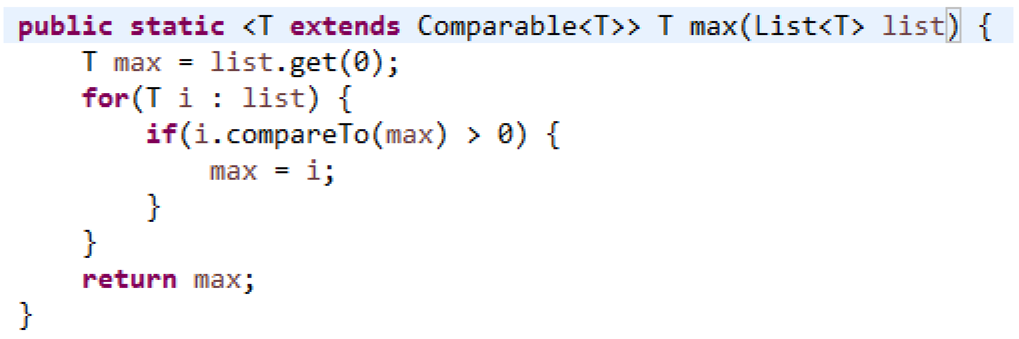
In the second call, `ints2` is of type `List<Number>` which “IS-A” `Collection<? super Integer>` (since `Number` is a superclass of `Integer`), so the `count` method can be called.

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.

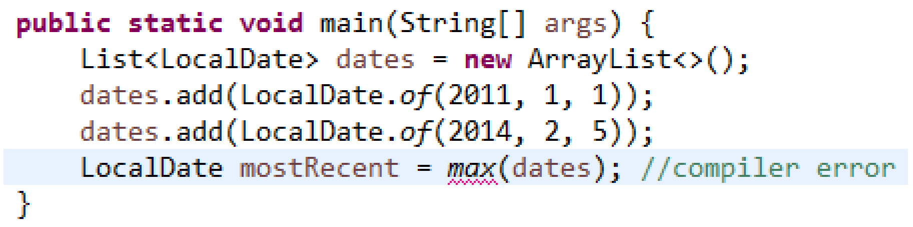
Another Example – improving implementation of the max function

We saw before that the following implementation of max was not general enough



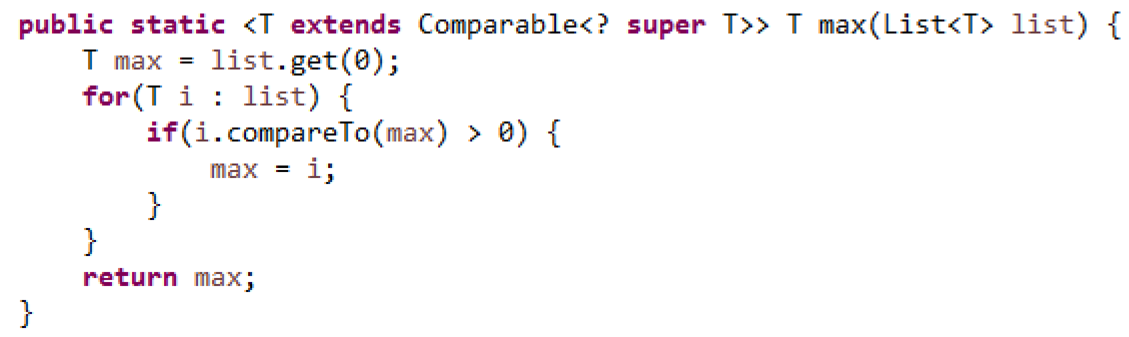
```
public static <T extends Comparable<T>> T max(List<T> list) {
    T max = list.get(0);
    for(T i : list) {
        if(i.compareTo(max) > 0) {
            max = i;
        }
    }
    return max;
}
```

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.

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).

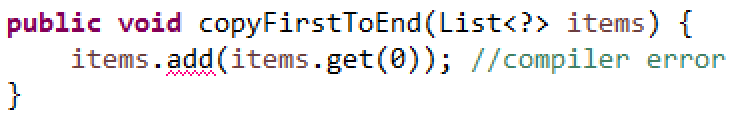
```
List<? super Integer> list = new ArrayList<>();
list.add(1);
list.add(2);
Object ob = list.get(0);
System.out.println(ob.toString());  //output:  1
```

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

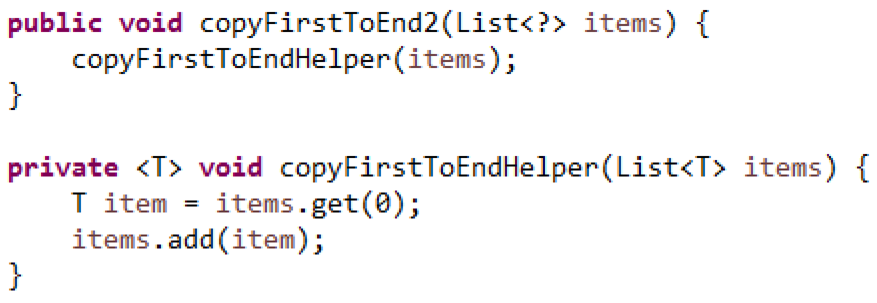
First Try:



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

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

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



```
public void copyFirstToEnd2(List<?> items) {
    copyFirstToEndHelper(items);
}

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

Notes:

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

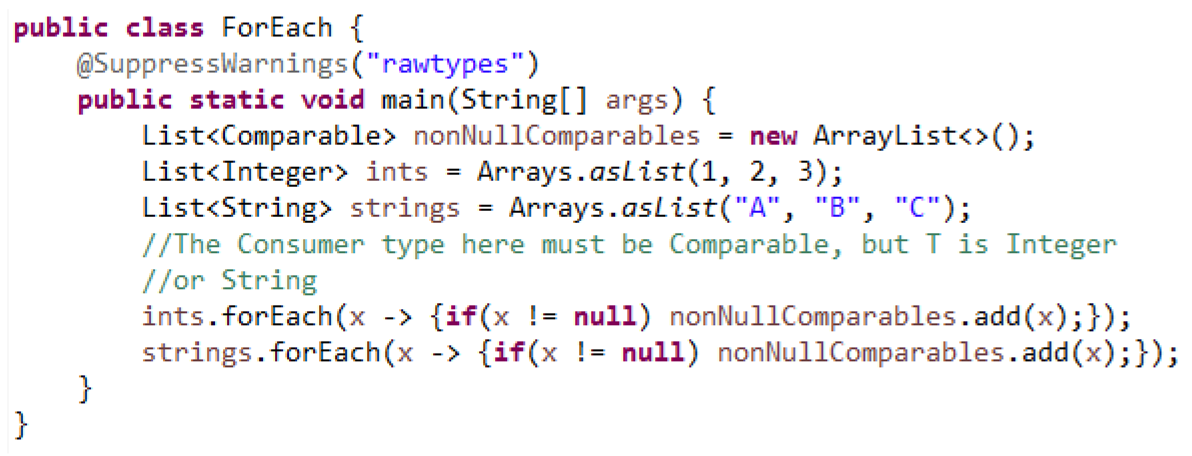
Understanding Common Generic Signatures: `forEach`

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 paticular `Collection` type T.

Here is an example:



```java
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);});
    }
}
```

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 of integers to it, and another list 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.

In more detail: When we add the Integers 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 using a Consumer<Comparable> (which is a Consumer<? super String>) as an argument to forEach as it traverses a List<String>.

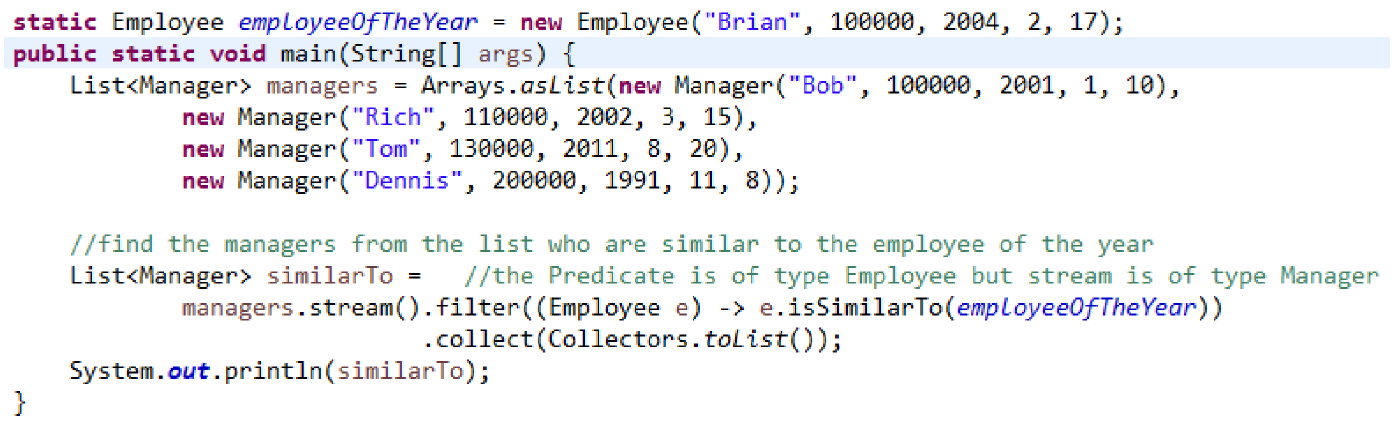
Demo: `lesson11.lecture.generics.signatures`

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 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);
   }
}
```

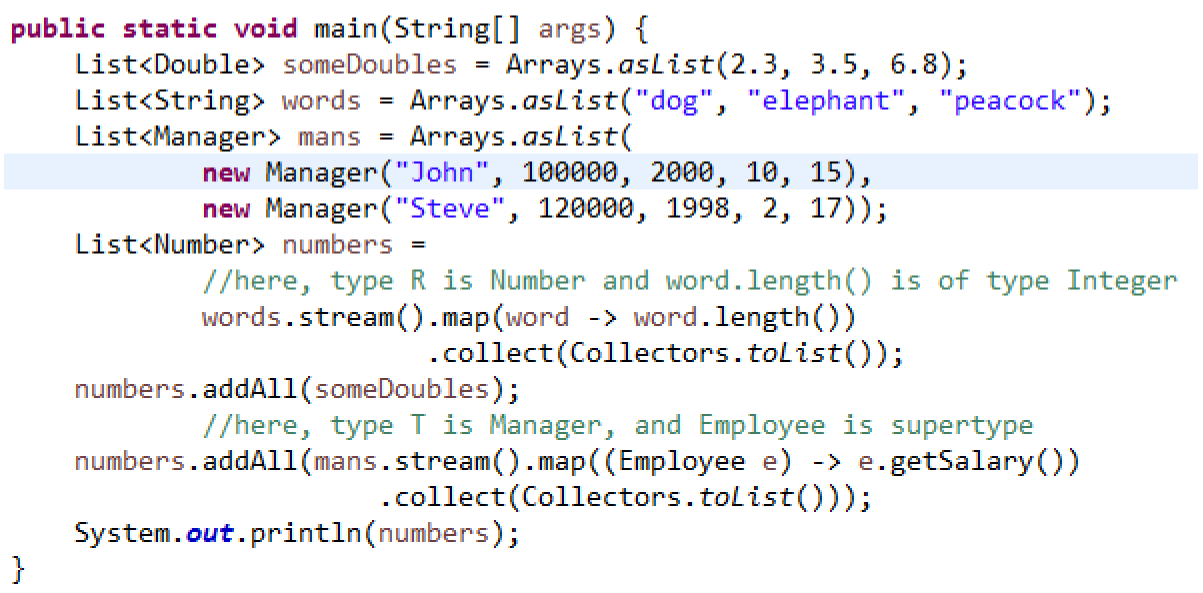
Understanding Common Generic Signatures: `map`

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 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.

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.

See demo lecture.generics.BoundedTypeVariable for a development of examples leading to the most general possible version.

Questions:

1. **Does the benefit of bounded wildcard outweigh its complexity?**
2. **As a Java developer, when do you think you should/want to use generics?**