**LESSON 7**

1. **Default Method :**

* is a fully implemented method within an interface, whose declaration begins with the keyword default.
* eliminate the need to create special classes that represent a default implementation of the interface

1. **Static Method:**

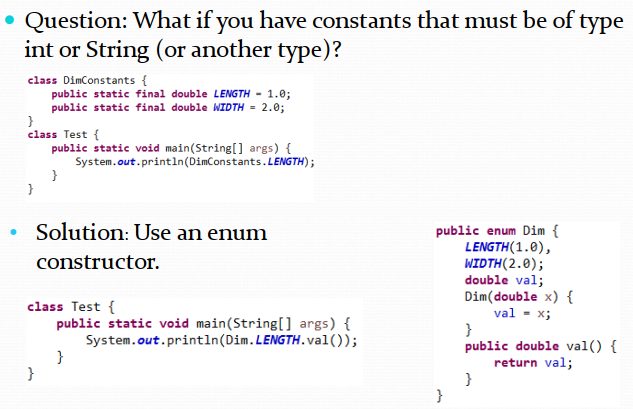
* is a fully implemented static method having the same characteristics as any static method in a class
* eliminate the need to create special utility classes that naturally belong with the interface

1. **Solution to evolving API problem**

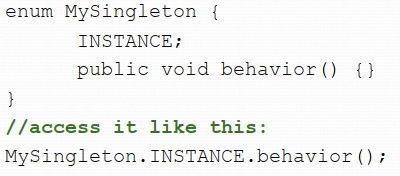
* When we add a new method to new interface with default implementation:
  + Legacy code won’t need to be required to implement new default methods, so existing code won’t be break
  + New functionality will be available for new client projects

1. **Application of default method:**

* Enums can inherit from other type which is interface
* Using enums as constants in an application
* Optimal, threadsafe implementation of the Singleton Pattern
* Enum is not allowed to inherit from any other class
* Enum can implement interface



* Enum implementation for singleton class

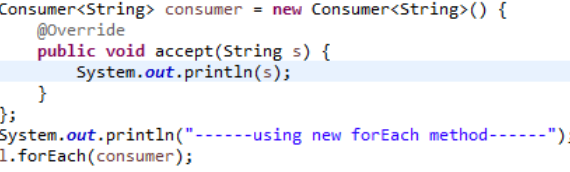


1. **Iterator:**

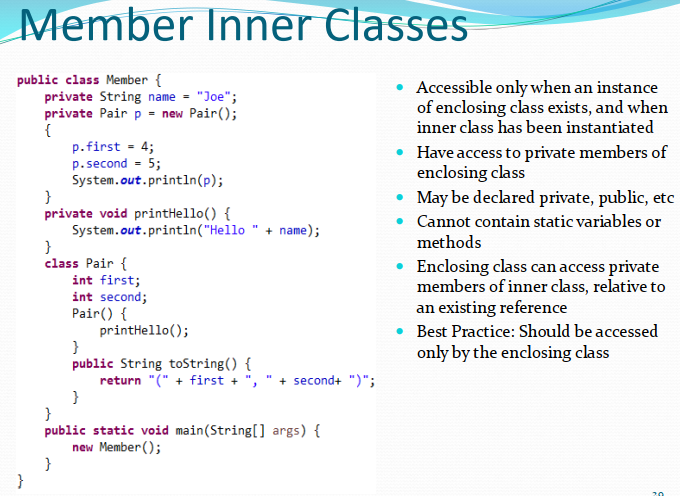
* Iterable interface supports iterate through a collection
* The only method in Iterable is iterator() -> return an iterator
* Iterator has two methods : hasnext(), next()

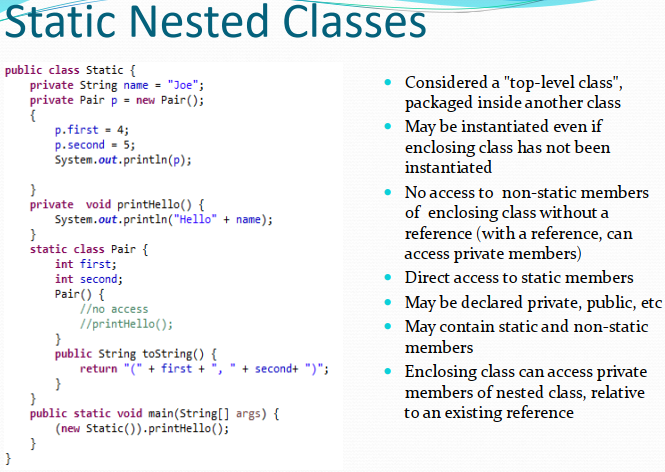
1. **Consumer:**

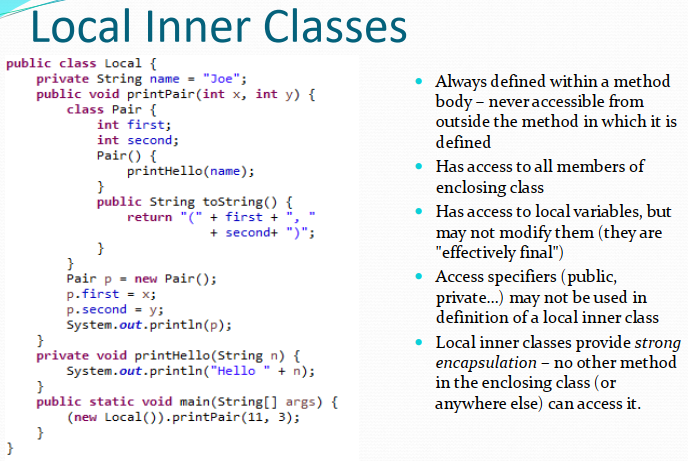
* Is a new interface in Java 8 with one abstract method accept a single argument that no return value

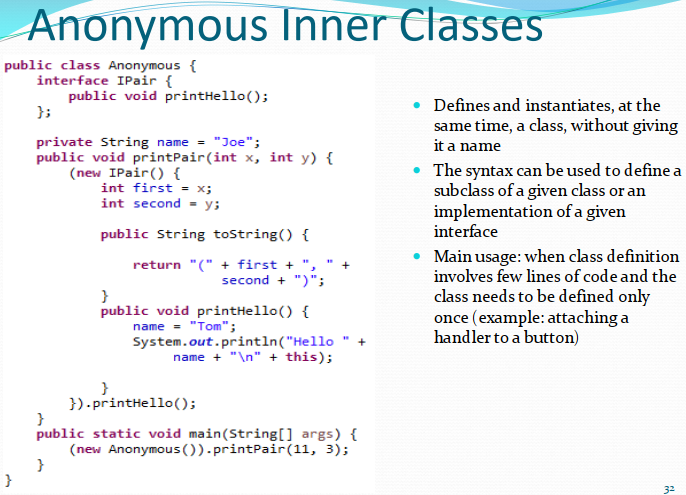


1. **Nested Classes**





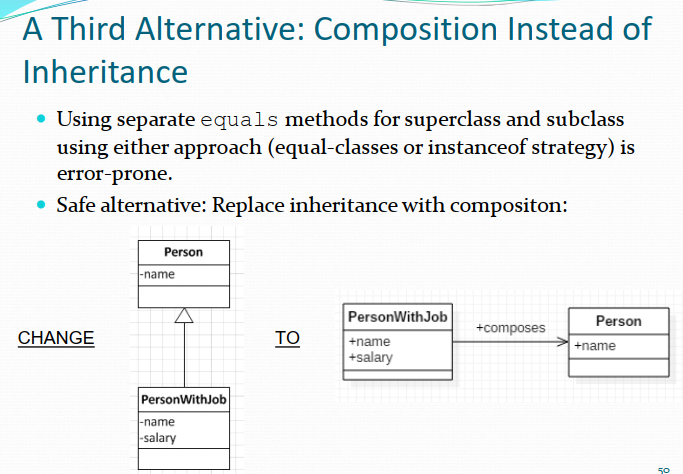


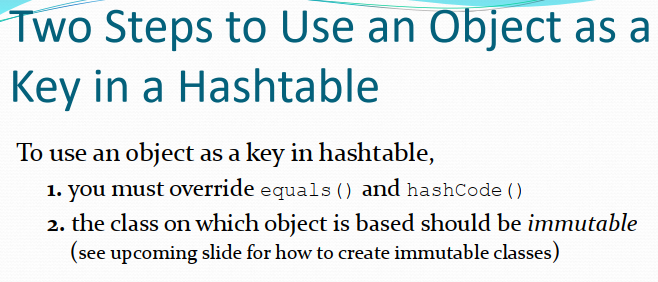


1. **Default method rules in Interface**

* If a class implements an interface with default method, that class inherit the default method or can override it
* Potential clash if :
  + Two interface has the same default method
  + Interface and super class has the same default method
* Static method doesn’t clash if two interface has the same name and implementation inside

1. **Object has 3 base methods : equal(Object e), hashcode,toString**





1. **A class is immutable if the data it stores cannot be modified once it is initialized**

* All fields should be private and final
* Return getter not setter
* Mark class final to not be inherited
* Mark getters don’t return mutable object

**LESSON 8**

1. **Programs are declarative (“what”) rather than imperative (“how”)**
2. **Side effect free function:**

* Do not change object state
* Have referential transparency
* No side effect

1. **Lamda expression is a block of code with a list of formal parameters and a body.**
2. **Functional interface is an interface that specify exactly one abstract method**
3. **Anonymous way:**

Collections.sort(obj, new Comparator<Person>(){

@override

Public int compare(Person p1, Person p2){

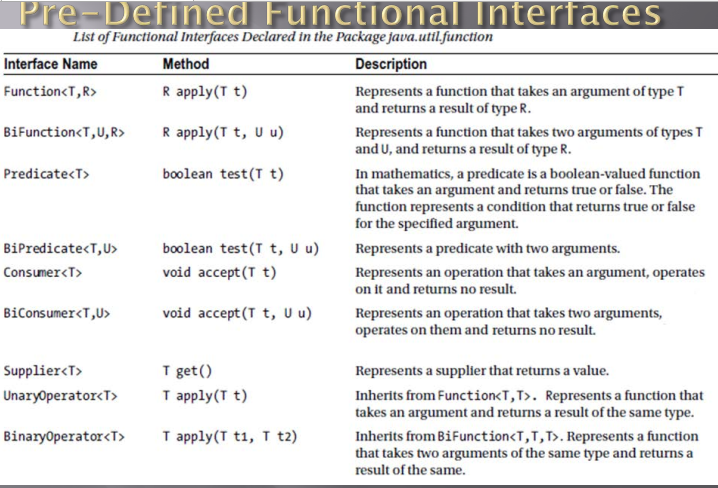
Return p1.getName().compareTo(p2.getName());

}

};)

*Using Lamda:*

Collections.sort(obj, (p1,p2) -> p1.getName().compareTo(p2.getName()));



1. **Consumer Interface :**

Consumer<String> consumer = **new** Consumer<String>() {

**public** **void** accept(String s) {

System.***out***.println(s);

}

};

Consumer<String> consumer1 = System.***out***::println;

Consumer<String> consumer2 = x -> System.***out***.println(x);

1. **Predicate check is number**

Predicate<String> numberOnly = x -> {

**if**(x == **null**) **return** **false**;

**return** x.chars().allMatch(Character::*isDigit*);

};

System.out.println(numberOnly.test(“abc”));

1. **Free Parameter is parameter that is not defined inside a block of code**
2. **Closure is a block of code on the right hand side of lamda expression**
3. **Comparator Interface**

**class** AccountComparator **implements** Comparator<Account> {

**public** **int** compare(Account a1,Account a2) {

**return** a1.getName().compareTo(a2.getName());

}

}

Comparator<Account> comparator1 = (a1,a2) -> a1.getName().compareTo(a2.getName());

Collections.*sort*(**new** ArrayList<Account>(), **new** AccountComparator());

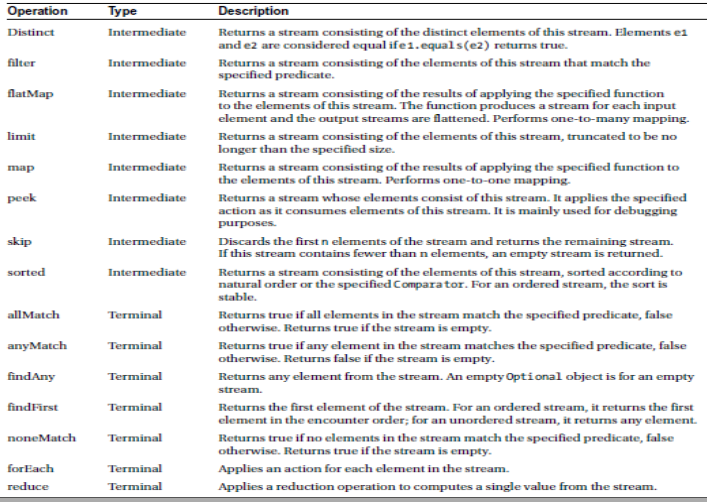
1. **A method reference** is a shorthand to create lamda expression using existing method : **classname::method**

**LESSON 9**

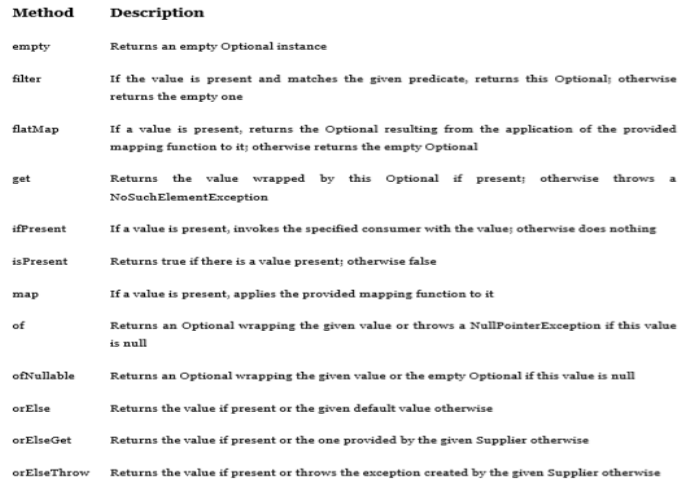
1. **Monad**

* Is a special data structure, available in some languages, that serves as a wrapper class, to support various operations.
* Support chaining operations, so that the output of each monad operation is another monad
* Stream and Optional are monads

1. **Intermediate is lazy operation, terminal is eager operation**
2. **Intermediate and Terminal Operation**



1. **Distinct and sorted** are two statefull lamda



1. **The difference between map and flatmap:**

* Function pass to map operation return a single value
* Function pass to flatmap operation return a stream

1. **Comparator:**

* Stream<String> longestFirst = words.stream().sorted(Comparator.comparing(String::length).reversed());
* Collection.sort(words, Comparator.comparing(String::length).thenComparing(CLASSNAME::MethodReference))

Stream().sorted(comparator or lamda with 2 parameters return int)

Collection.sort(list object, comparator or lamda with 2 parameters return int)

Comparator.comparing((Type t) -> t.getValue()) = comparator = Lamda with 2 paramter return int

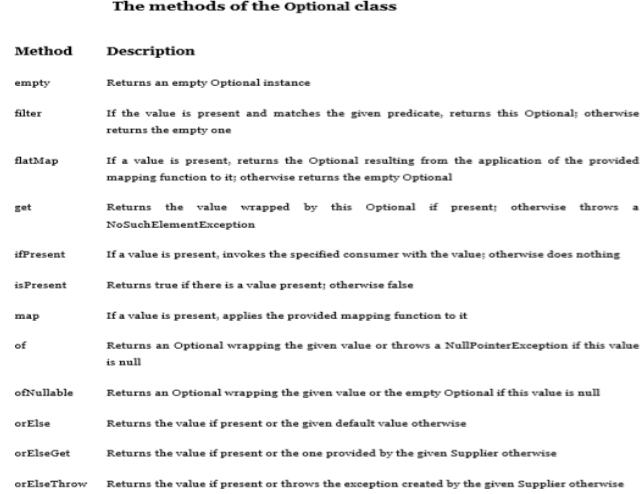
Ex:

Comparator.comparing((String s) -> s.length())

Comparator<String> comparator = (x,y) -> Integer.compare(x.length, y.length);

Stream.*of*("abc","a","bc").sorted(Comparator.*comparing*((String e) -> e.length()).reversed());

Stream.*of*("abc","a","bc").sorted(Comparator.*comparing*(String::length).reversed());

* Collections.*sort*(Stream.*of*("abc","a","bc").collect(Collectors.*toList*()),
* (x,y)-> Integer.*compare*(x.length(), y.length()));
* 

1. **Collecting result:**

* Stream.collect(Collectors.toList())
* Stream.collect(Collectors.joining(“ ”))
* IntSummaryStatistics summary = words.stream.collect(Collectors.summarizingInt(String::length));
* double averageWordLength = summary.getAverage();
* double maxWordLength = summary.getMax();

1. **Stream cannot be reused after use terminal operation**
2. **Primitive type stream**

* IntStream ints = IntStream.of(1, 2, 4, 8);
* IntStream ones = IntStream.generate(() -> 1); // infinite stream
* IntStream naturalNums = IntStream.iterate(1, n -> n+1); // infinite stream
* IntStream zeroToNinetyNine = IntStream.range(0, 100);
* IntStream zeroToHundred = IntStream.rangeClosed(0, 100); // Upper bound is included
* Stream<Integer> integers = IntStream.range(0, 100).boxed(); // convert to stream

1. **Create lamda library**

public static final BiFunction<List<Customer>, String, List<String>> NAMES\_IN\_CITY

= (list, searchStr)-> list.stream().filter(cust -> cust.getCity().startsWith(searchStr))

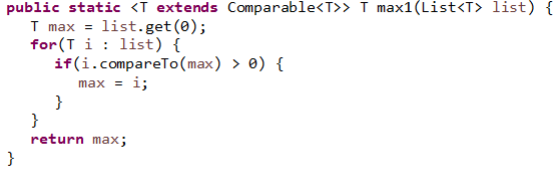
.map(cust -> cust.getName()).sorted().collect(Collectors.toList());

List<String> listStr = LambdaLibrary.NAMES\_IN\_CITY.apply(list, "Ma");

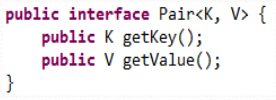
**LESSON 10**

1. **Benefit of Generic**:

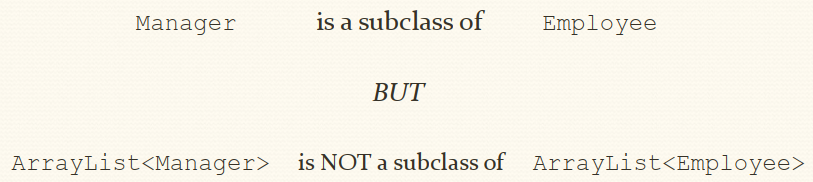
* Stronger type checks at compile time
* Reduce down casting : Ex : Employee emp = (Employee) Item.get(0)
* Support the most general possible API for methods that can be generalized

Ex: Get max value in list 

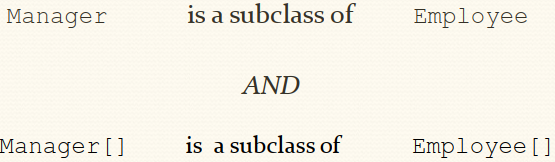
1. **Generic Interface:**

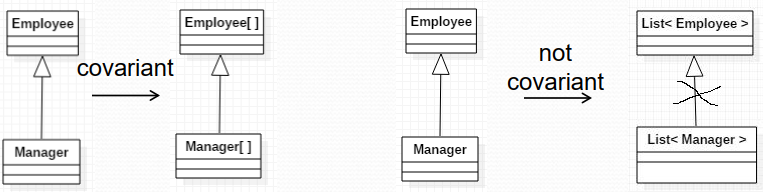


1. **Generic Subtyping is Not Covariant**

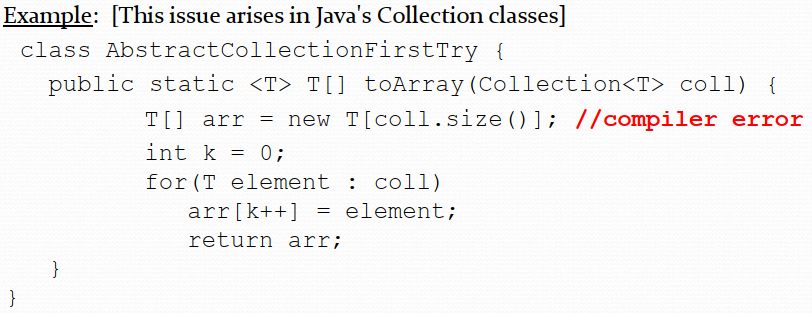


1. Array Subtyping is Covariant

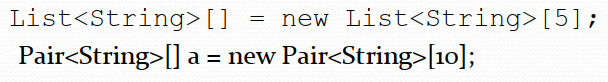




1. **Component type of array is not allowed as type variable**

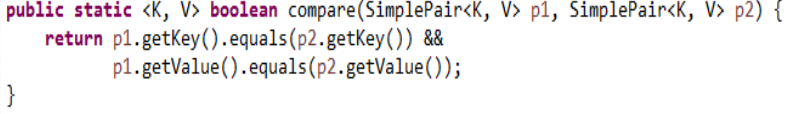


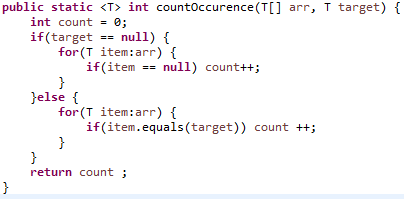
1. **Component type of array is not allowed as parametrized type**



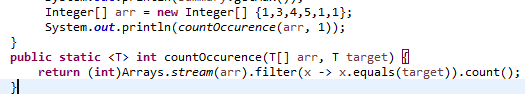
1. **Reifiable Type :** is type whose type information is fully available at runtime. This includes primitives, non-generic type, raw types and invocation of unbound wildcards
2. **Non-Reifiable Type** is type where information is fully removed at compile –time by erasure
3. **Generic method :** is method that introduce their own type parameter. It is similar to generic type but scope is just inside method where it is declared.

**Ex:**

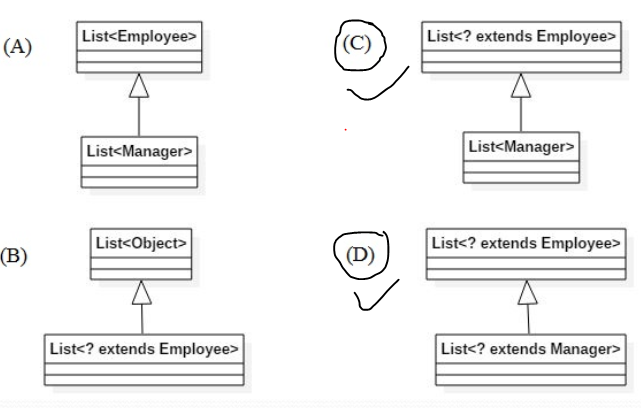


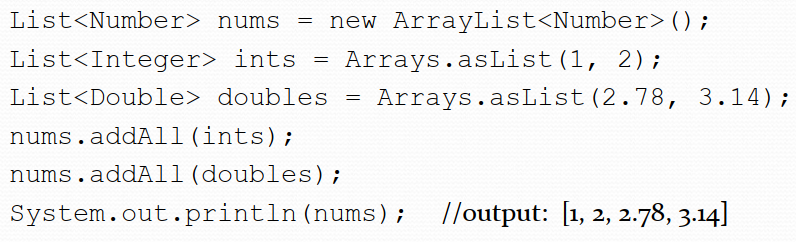
1. **Count occurrence generic method**

**Use lamda to revise:**



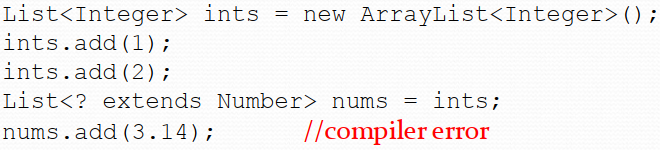
1. **Bounded wild card**





1. **Limitation of <? extend class>**

When the extends wildcard is used to define parametrized type, the type cannot be used for adding new element



1. **? super bounded 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

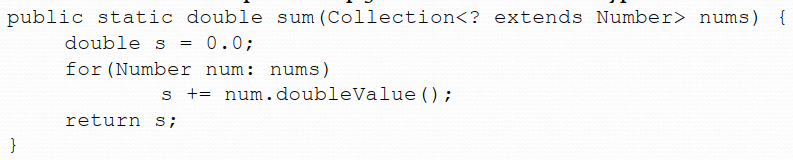
1. **Get and Put wildcard principal**

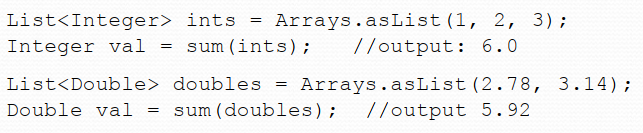
Use an extends wildcard when you only get values out of a structure. Use a super wildcard when you only put values into a structure. And don’t use a wildcard at all

when you both get and put values.

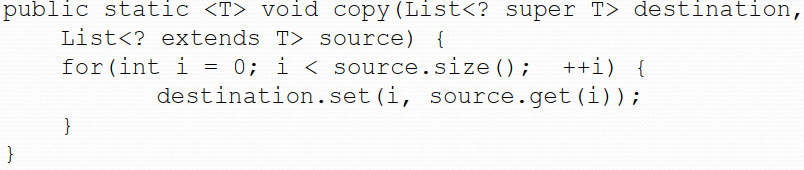
Ex :

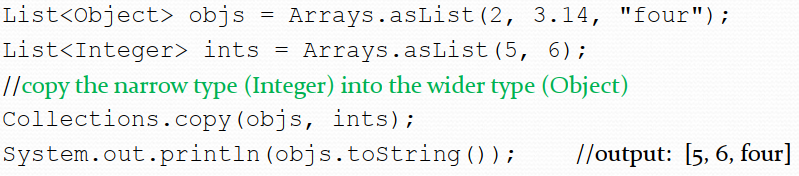
? extends





? super





1. **Helper Method**

