Programming Languages and Concepts

Siegfried Benkner

Research Group Scientific Computing

Universität Wien

Contents

- **Collection Framework**

- **Generic Types and Methods**
- **Generics and Subtyping**
- **Bounded Type Parameters**
- **Unbounded and Bounded Wildcards**
- **Type Erasure**

See: https://docs.oracle.com/javase/tutorial/java/generics/index.html

Java Collection Framework

- The Java Collections Framework provides interfaces and classes for implementing collections (=containers).
- Collections are used to manage multiple elements into a single unit, i.e., to store, retrieve, manipulate, and communicate aggregate data based on frequently used data structures (lists, trees, hashmaps, …).
- Originally, collections were defined with elements of type `Object` so that they could hold objects of arbitrary types.
- Based on the concept of Generics (since Java 1.5), collections can be restricted to elements of a specific type.

See: Java Collections Tutorial; https://docs.oracle.com/javase/tutorial/collections/index.html

Java Collection Framework

- **Interfaces**
  - Allow collections to be manipulated independently of the details of their representation and implementation.

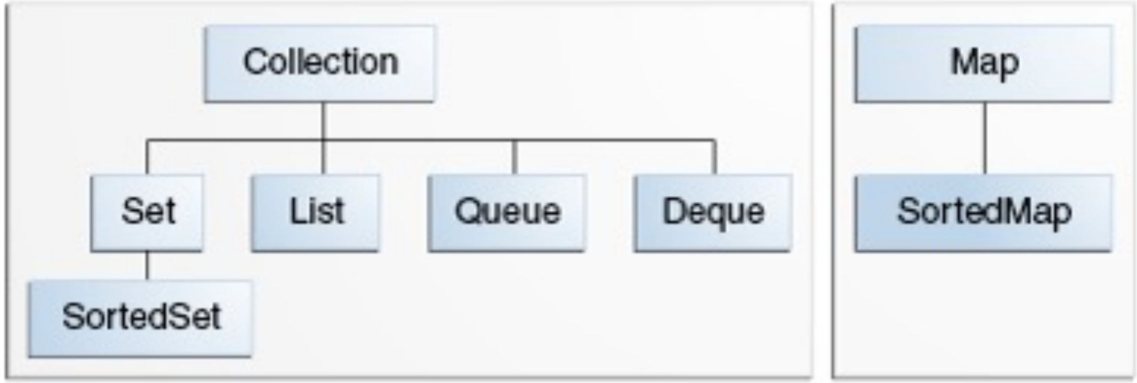
- **Implementations**
  - Concrete implementations of the collection interfaces.

  → Reusable data structures.

- **Algorithms**
  - Polymorphic methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces.

  → Reusable functionality.

Java Collection Framework - Interfaces



- `Collection` basic functionality used by all collections (e.g., add/remove)
  - `Set` does not allow duplicate elements.
  - `SortedSet` ordering of elements in the set.
  - `List` ordered with control over where each element is inserted
  - `Queue` (FIFO) additional insertion, extraction, and inspection operations.
  - `Deque` (LIFO, FIFO) double-ended queue

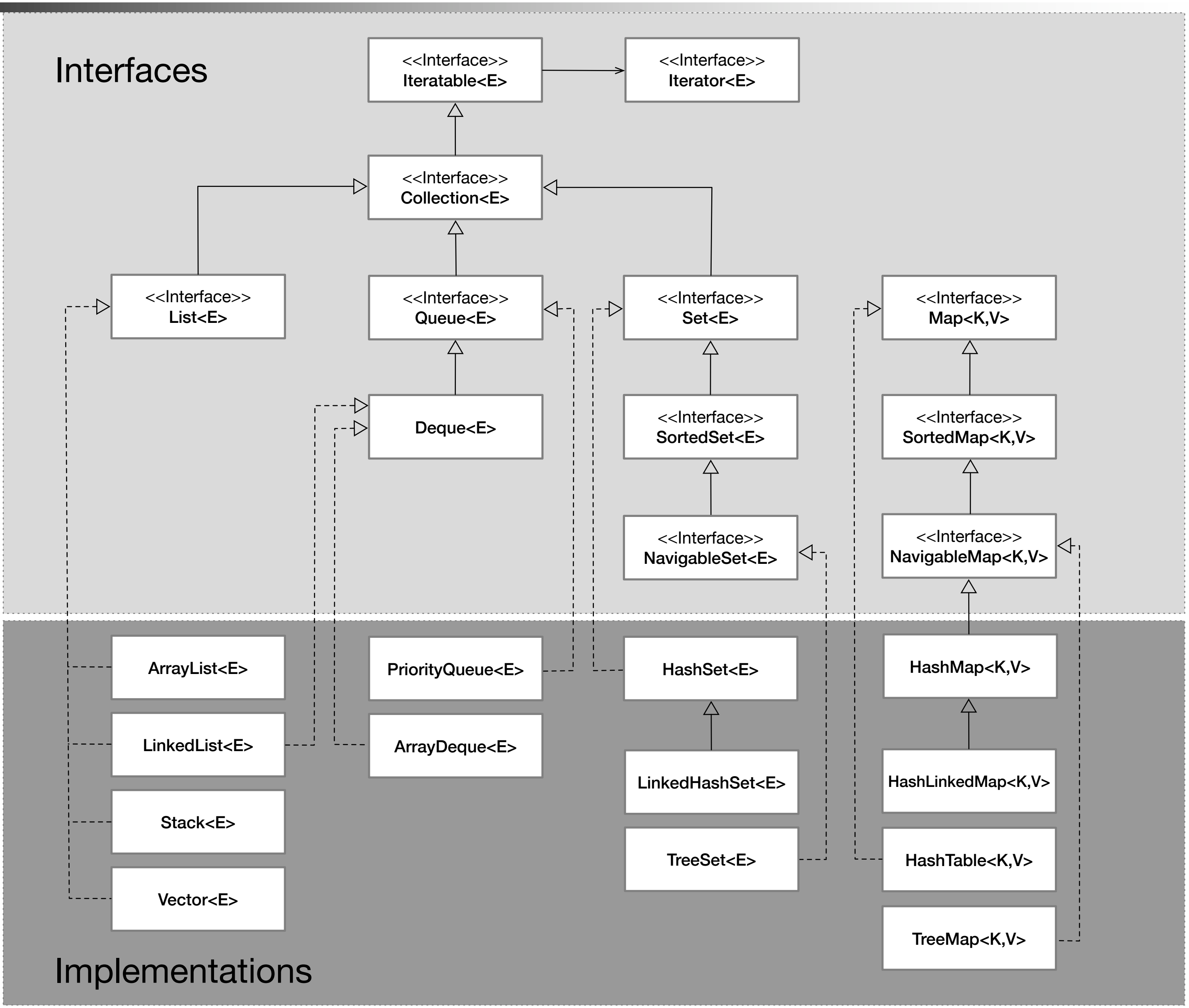
- `Map` maps keys and values similar to a hashtable.
  - `SortedMap`, key-value pairs in ascending order or in an order specified by a Comparator.

Java Collection Framework - Algorithms

- **Sorting**
- **Shuffling**
- **Routine Data Manipulation**
- **Searching**
- **Composition**
- **Finding Extreme Values**

Java Collection Framework

Program to interfaces!



Interfaces

<<Interface>> Iteratable<E>

<<Interface>> Iterator<E>

<<Interface>> Collection<E>

<<Interface>> List<E>

<<Interface>> Queue<E>

<<Interface>> Set<E>

<<Interface>> Map<K,V>

Deque<E>

<<Interface>> SortedSet<E>

<<Interface>> SortedMap<K,V>

<<Interface>> NavigableSet<E>

<<Interface>> NavigableMap<K,V>

ArrayList<E>

PriorityQueue<E>

HashSet<E>

HashMap<K,V>

LinkedList<E>

ArrayDeque<E>

LinkedHashSet<E>

HashLinkedMap<K,V>

Stack<E>

TreeSet<E>

HashTable<K,V>

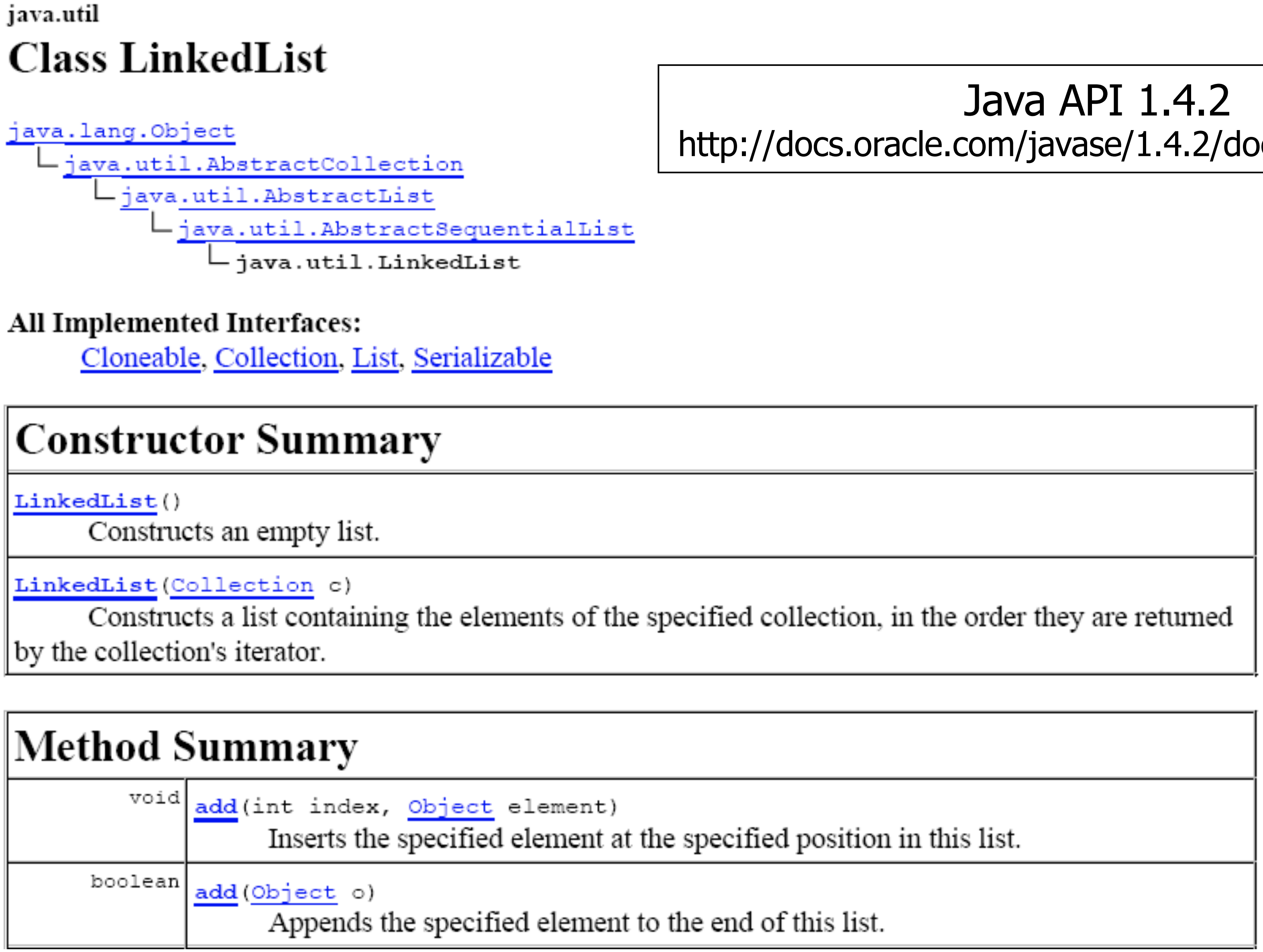
Vector<E>

TreeMap<K,V>

Implementations

Example: No Generics, Java 1.4

Java API 1.4.2
http://docs.oracle.com/javase/1.4.2/docs/api/index.html



java.util

Class LinkedList

```
java.lang.Object
  └java.util.AbstractCollection
      └java.util.AbstractList
          └java.util.AbstractSequentialList
              └java.util.LinkedList
```

All Implemented Interfaces:
Cloneable, Collection, List, Serializable

Constructor Summary
`LinkedList()` Constructs an empty list.
`LinkedList(Collection c)` Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.

Method Summary	
`void`	`add(int index, Object element)` Inserts the specified element at the specified position in this list.
`boolean`	`add(Object o)` Appends the specified element to the end of this list.

Example: No Generics, Java 1.4

```
Person p = new Person();
List ll = new LinkedList();   // create empty list
ll.add(p);                    // insert p into list

Arbeiter a = new Arbeiter();
ll.add(a);
...

for (Iterator i = ll.iterator(); i.hasNext(); ) {
     Object o = i.next();        // get next object from list
     if (o instanceof Person) {
        p = (Person) o;          // cast Object to Person
        ...
}
```

method add() of class List:
boolean add(Object o)

method next() of Interface Iterator:
Object next()

Generic Types

- A generic type is a type with formal type parameters.

- Type parameters provide a way to re-use the same code with different types.

```
public class LinkedList<E> ... {      // generic type with
                                      // type parameter E
```

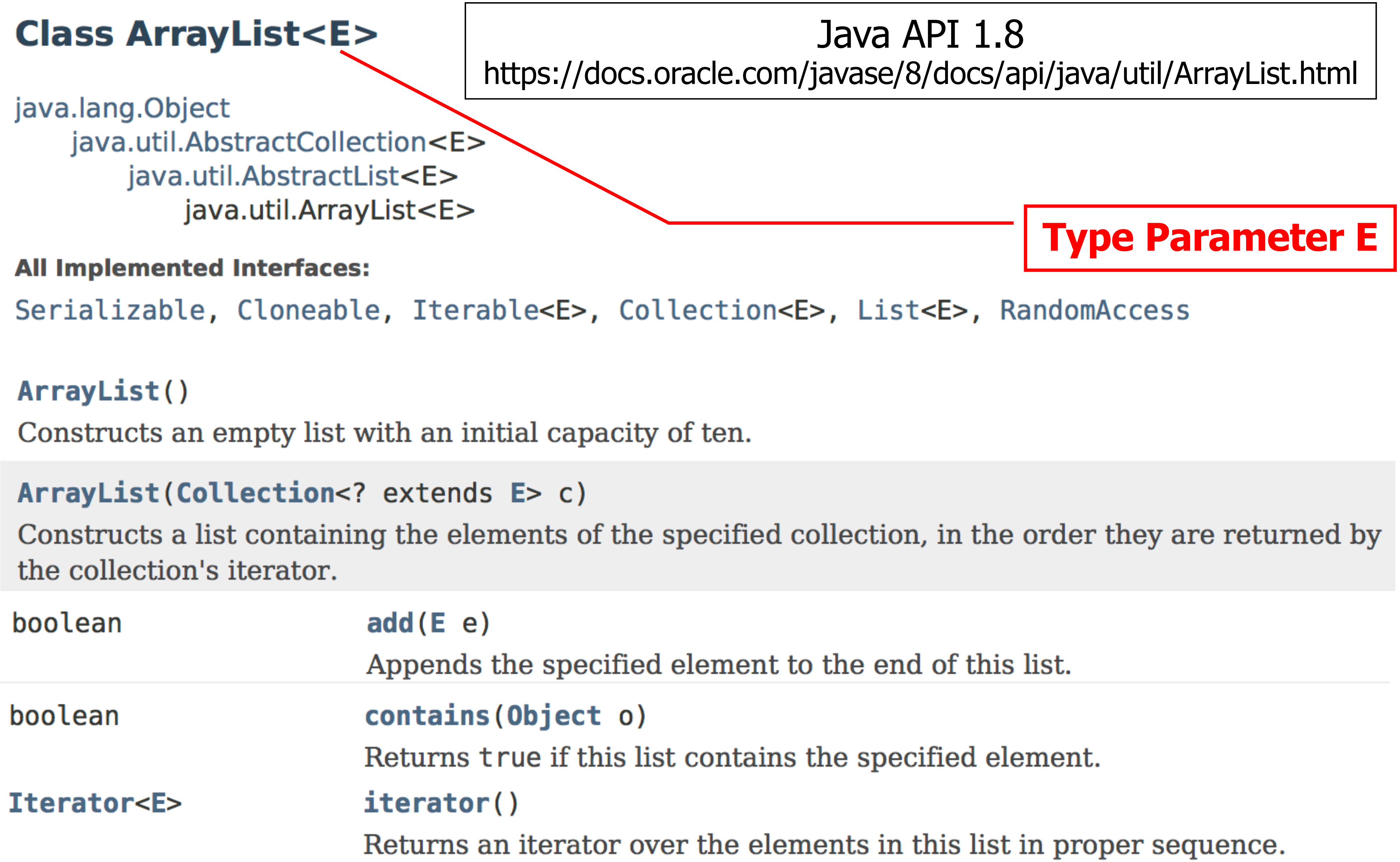
- By providing an actual type argument code can be restricted to that type.

```
List<Person> wl = new LinkedList<Person>(); // actual type
                                               // argument Person
```

See: Java Generics Tutorial https://docs.oracle.com/javase/tutorial/java/generics/

Example: Generics (since Java 1.5)

Java API 1.8
https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html



Type Parameter E

Class ArrayList<E>

java.lang.Object
    java.util.AbstractCollection<E>
        java.util.AbstractList<E>
            java.util.ArrayList<E>

All Implemented Interfaces:

`Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess`

`ArrayList()`
Constructs an empty list with an initial capacity of ten.

`ArrayList(Collection<? extends E> c)`
Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.

`boolean`	**`add(E e)`** Appends the specified element to the end of this list.
`boolean`	**`contains(Object o)`** Returns `true` if this list contains the specified element.
`Iterator<E>`	**`iterator()`** Returns an iterator over the elements in this list in proper sequence.

Example: Generics

```
Book book = new Book(...);
List<Article> al = new ArrayList<>();  // diamond <>
al.add(book);                          // add book to list

//old-fashioned iterator
for (Iterator<Article> i = al.iterator(); i.hasNext(); ) {
     Article v = i.next();            // no cast required !!
     System.out.println(v.getId());
}

// new for "each" loop
for (Article a : al)
     System.out.println(a.getId());
```

method add() in List<E>:
boolean add(E e)

method iterator in List<E>:
Iterator<E> iterator()

method next() of interface Iterator<E>:
E next()

Generic Methods

- Generic methods are methods that introduce their own type parameters.
- 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.

```
<T> void fromArrayToCollection(T[] a, Collection<T> c) {
   for (T o : a) c.add(o);
}
```

https://docs.oracle.com/javase/tutorial/extra/generics/methods.html

Generic Methods

- **Type inference**

  When invoking generic methods, the compiler infers the most specific type argument based on the types of the actual arguments.

```
<T> void fromArrayToCollection(T[] a, Collection<T> c) {
   for (T o : a) c.add(o);
}
```

```
Collection<Object> co = new ArrayList<Object>();
Collection<String> cs = new ArrayList<String>();
Object[] oa = new Object[10];
String[] sa = new String[10];
fromArrayToCollection(oa, co);  // T → Object
fromArrayToCollection(sa, cs);  // T → String
```

https://docs.oracle.com/javase/tutorial/extra/generics/methods.html

Generics - Benefits

- Stronger type checks at compile time instead of runtime checks.

  A Java compiler applies strong type checking to generic code and issues errors if the code violates type safety.
- Programmers can implement generic algorithms that work on collections of different types, can be customized, and are type safe and easier to read.
- Elimination of explicit type casts.

```
List list = new ArrayList(); // without generics
list.add("hello");
String s = (String) list.get(0); // cast
```

```
List<String> list = new ArrayList<String>(); //with Generics
list.add("hello");
String s = list.get(0);          // no cast
```

Generics – Subtyping

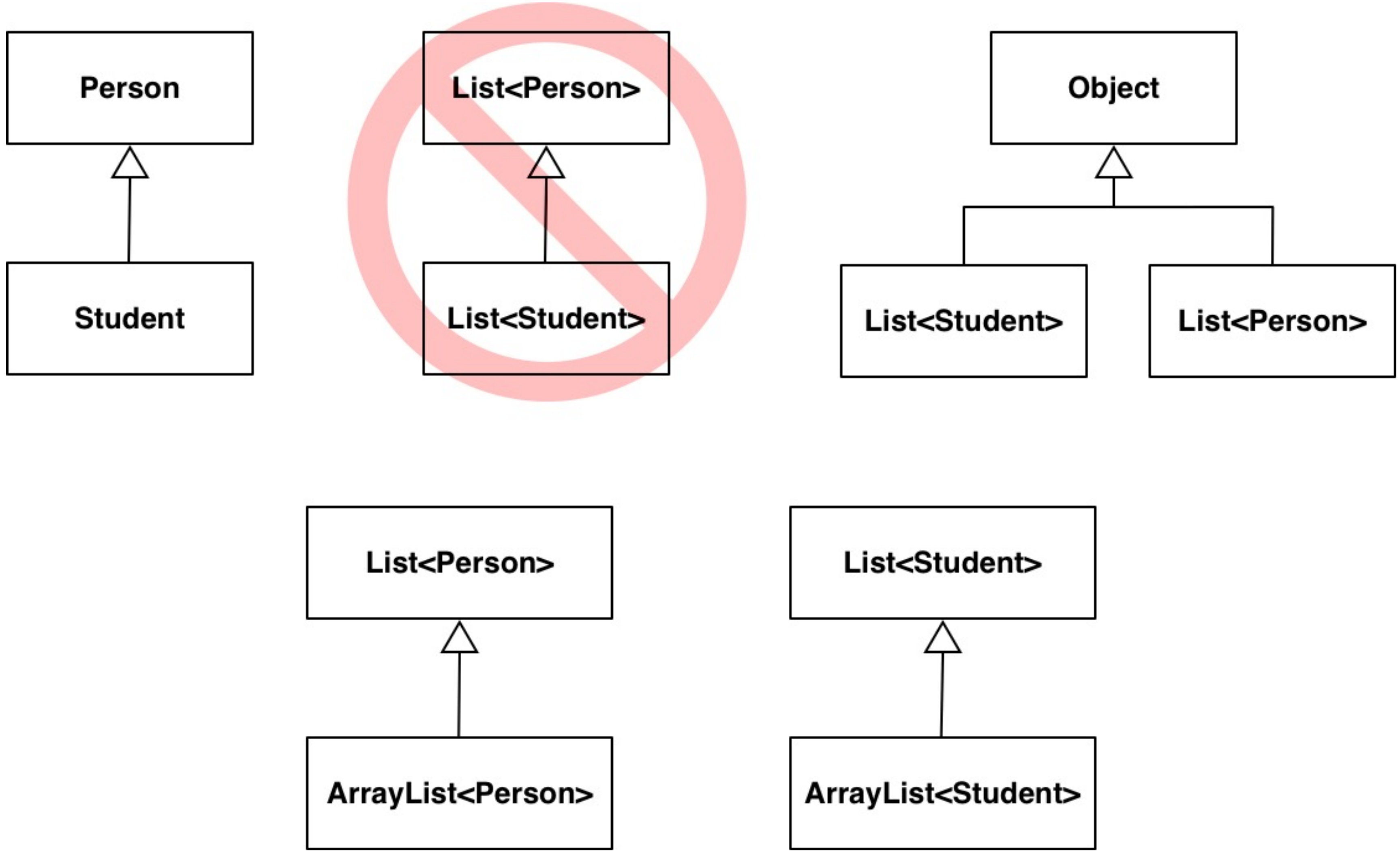
In Java, if `S` is a subtype (subclass or subinterface) of type `T`, and `G` is some generic type declaration, then `G<S>` is not a subtype of `G<T>`.

```
Person p; Student s;
...
p = s;   //okay, since Student is subclass of Person

List<Student> ls = new ArrayList<>();
List<Person> lp = li;
//Compiler Error
//Type mismatch: cannot convert from List<Student> to List<Person>
```

- This property of the Java type system is called Invariance.

Generics – Subtyping

- Type parameters in Java are invariant.



Generics – Subtyping

- Type parameters in Java are invariant, rather than covariant like arrays.

```
Person[] ap = new Person[100];
Student[] as = new Student[100];

ap = as;       // okay - arrays are covariant


List<Person> lp = new ArrayList<Person>();
List<Student> ls = new ArrayList<Student>();

lp = ls;       // Type mismatch: cannot convert from
               // List<Student> to List<Person>
```

Generics – Unbounded Wildcards

- The wildcard `?` represents an unknown type.
- It is useful for methods of a generic class that don't depend on the type parameter.
- It can be used as the type of a parameter, field, or local variable; as a return type (not recommended).
- Note: It is never used as a type argument for a generic method invocation, a generic class instance creation, or a supertype.

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

Generics – Unbounded Wildcards

```
public static void printList(List<Object> list) {
    for (Object elem : list) System.out.println(elem + " ");
    System.out.println();
}
```

- `printList` attempts to print a list of any type, but it fails to achieve that goal. It cannot print `List<Integer>, List<String>`, …, because they are not subtypes of `List<Object>.`
- To write a generic printList method, `use List<?>`:

```
public static void printList(List<?> list) {
    for (Object elem : list) System.out.println(elem + " ");
    System.out.println();
}
```

https://docs.oracle.com/javase/tutorial/java/generics/unboundedWildcards.html

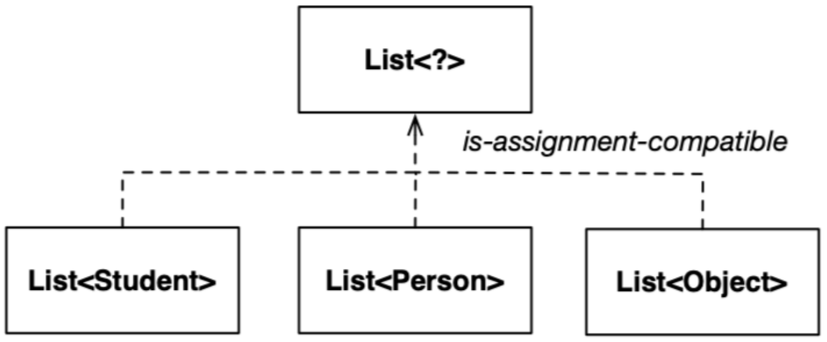
Example: Assignment Compatibility

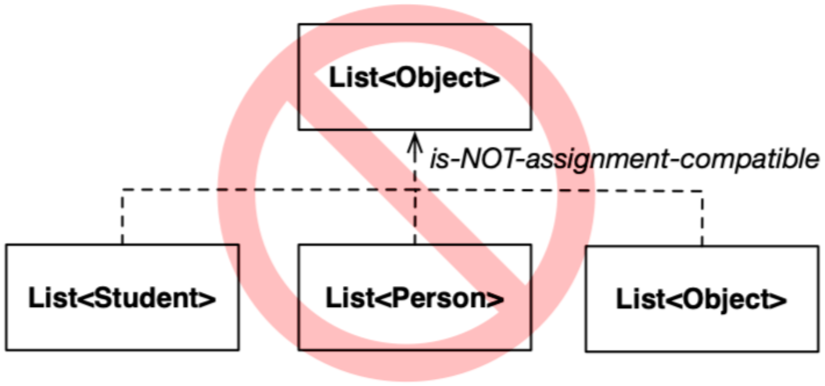
```
List<Person> lp;
List<Student> ls;
List<?> lw;
List l;
List<Object> lo;

lw = lp;
lw = ls;
lw = l;
lw = lo;

ls = lw;   // error
ls = lp;   // error
ls = l;    // unchecked
ls = lo;   // error

...
```

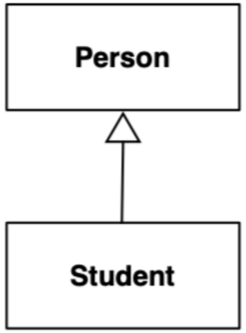
```
...

lp = lw;   // error
lp = ls;   // error
lp = l;    // unchecked
lp = lo;

l = lw;
l = lp;
l = ls;
l = lo;

lo = ls;   // error
lo = lp;   // error
lo = lw;   // error
lo = l;    // unchecked
```

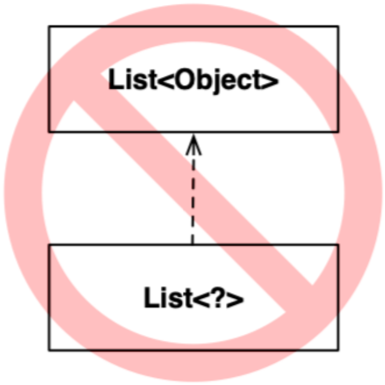
//unchecked → because of binary compatibility with pre-existing code

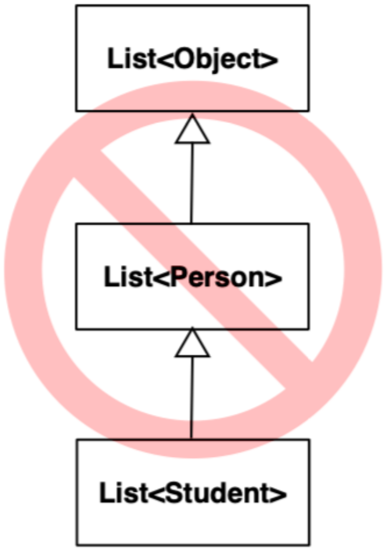
Example: Assignment Compatibility











Generics – Bounded Wildcards

- **Upper bounded wildcard**:

```
<? extends T>
```

Matches all types that are sub-types of T (including T)

- **Lower bounded wildcard**:

```
<? super T>
```

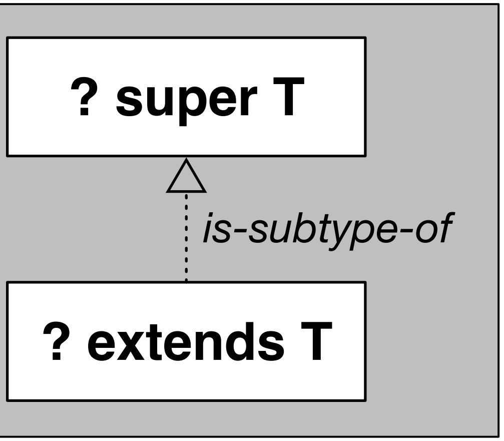
Matches all types that are super-types of T (including T)

For all type parameters T:

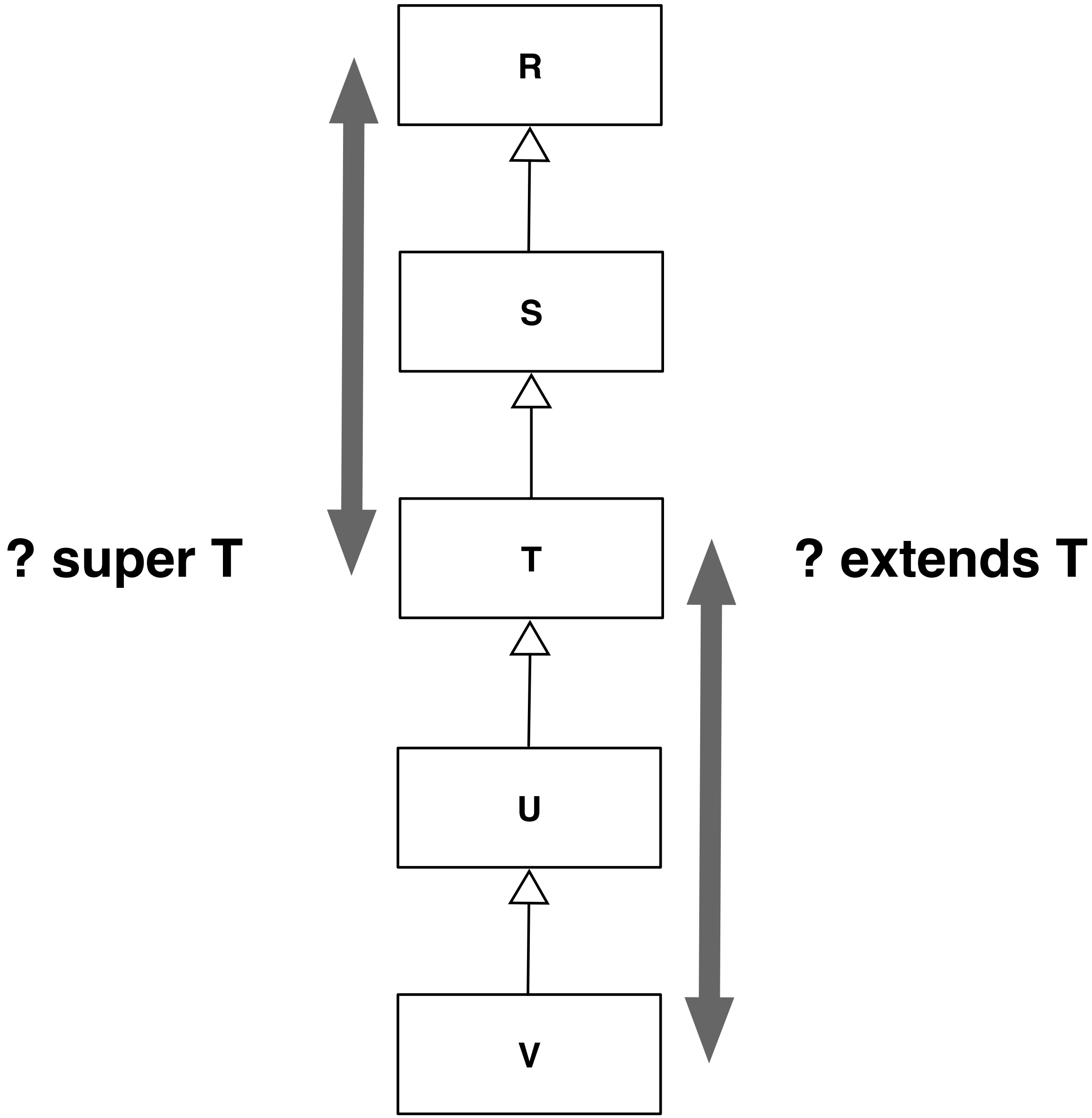
every Type C1 that my result from <? extends T>

is a sub-type of (or equal to)

every Type C2 resulting from <? super T>



Generics – Bounded Wildcards



R

S

T

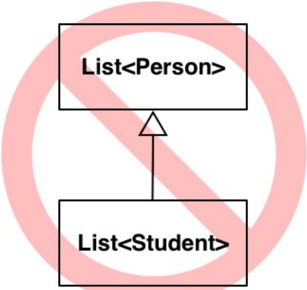
U

V

? super T

? extends T

Bounded Wildcards Guidelines



PECS principle: "producer extends consumer super"

- An "in" variable (producer) is defined with an upper bounded wildcard, using the extends keyword.
- An "out" variable (consumer) is defined with a lower bounded wildcard, using the super keyword.

```
Example: java.util.Collections.copy()
static <T> void copy(List<? super T> dest, List<? extends T> src)
Copies all of the elements from one list into another.
```

```
List<Person> pl = ...
List<Student> sl = ...
Collections.copy(pl, sl);     // ok
Collections.copy(sl, pl);     // compiler error
```

https://docs.oracle.com/javase/tutorial/java/generics/wildcardGuidelines.html

Generics – Bounded Wildcards

- Bounded wildcards are useful in situations where only partial knowledge about the type argument of a generic type is needed, but where unbounded wildcards carry too little type information.

```
public class Collections { ...
  public static <T> void copy
  (List<? super T> dest, List<? extends T> src) {

      ...
      for (int i=0; i<src.size(); i++)
        dest.set(i,src.get(i));

      ...
```

- Destination list must be capable of holding the elements from the source list.
- The destination list is required to have an element type with a lower bound T.
- The source list must have an element type with an upper bound T.

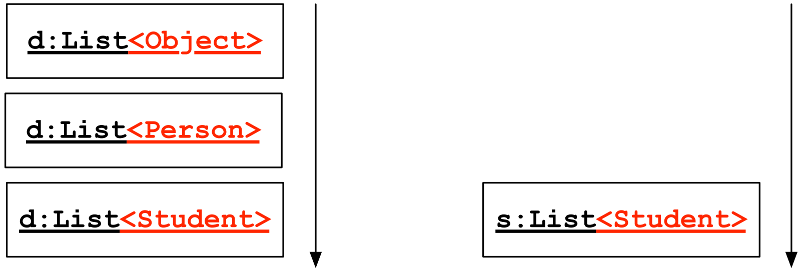
See: `java.util.Collections`

Wildcards Guidelines - Example

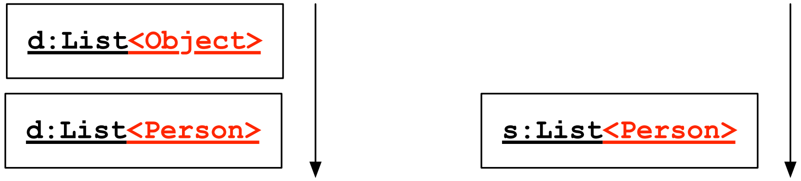
Callee

```
class Collections {
…
     static <T> void copy(List<? super T> dest, List<? extends T> src) { ... }
```

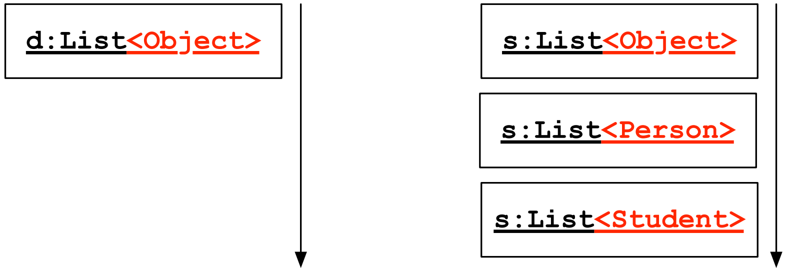
Caller



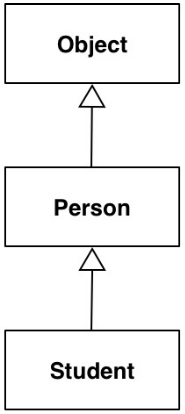
```
Collections.<Student>copy(List<? super Student> dest, List<? extends Student> src)
```

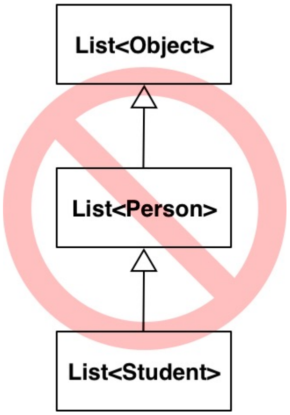


```
Collections.<Person>copy(List<? super Person> dest, List<? extends Person> src)
```



```
Collections.<Object>copy(List<? super Object> dest, List<? extends Object> src)
```





Wildcards Guidelines - Example

```
List<? super Person> lSp1 = new ArrayList<Person>();
List<? super Person> lSp2 = new ArrayList<Student>();    // Error
List<? super Person> lSp3 = new ArrayList<Object>();

List<? extends Person> lEp1 = new ArrayList<Person>();
List<? extends Person> lEp2 = new ArrayList<Student>();
List<? extends Person> lEp3 = new ArrayList<Object>();   // Error
```

Type Erasure

- With Java generic types the type information is discarded by the compiler and it is not available at run time.
- This process is called type erasure:
  - if the type parameter is unbounded, replace it with Object
  - if the type parameter is bounded replace it with first bound
- Main reason: binary compatibility with pre-existing code.

```
class Node<T> {                      //before type erasure
    private T data;
    private Node<T> next;

    public Node(T data, Node<T> next) {
        this.data = data;
        this.next = next;
    }
```

Type Erasure

- With Java generic types the type information is discarded by the compiler and it is not available at run time.
- This process is called type erasure:
  - if the type parameter is unbounded, replace it with Object
  - if the type parameter is bounded replace it with first bound
- Main reason: binary compatibility with pre-existing code.

```
class Node {                         //after type erasure
    private Object data;
    private Node next;

    public Node(Object data, Node next) {
        this.data = data;
        this.next = next;
    }
```

Type Erasure

Before type erasure:

```
public class Node<T extends Comparable<T>> { // bounded
   private T data;
   private Node<T> next;

   public Node(T data, Node<T> next) {
      ...
   }
```

After type erasure:

```
public class Node {
   private Comparable data;
   private Node next;

   public Node(Comparable data, Node next) {
      ...
   }
```

Generics – Concluding Remarks

- Generics enable types to be parameterized (with other types) when defining classes, interfaces and methods.
- They enable generic data structures/algorithms that work with different types.
- Generics facilitate type safety through better compile time checks.

- In C++, generic programming usually relies on templates.
- As opposed to Java generics, C++ templates are not erased; the C++ compiler generates code for each different instantiation of template parameters.

```
template<class T>
class Stack {
   ...
   public: void push(T) { ... };
   ...
```