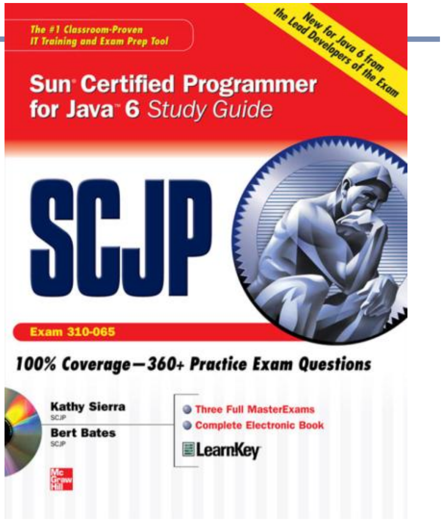
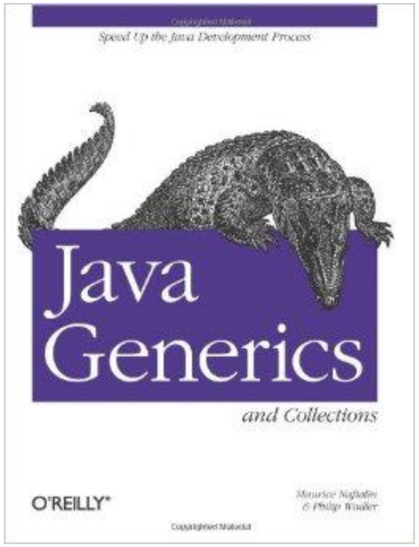
Java Generics

Maciej Kucharek, Paweł Łagan, Michał Dubel

Few words before we start

- This workshop session is heavily based on **Java Generics and Collections** by Maurice Naftalin & Philip Wadler
- The workshop just scratches the surface
- We strongly recommend you to read the book


The #1 Classroom-Proven IT Training and Exam Prep Tool
New for Java 6 from the Lead Developers of the Exam
Sun Certified Programmer for Java 6 Study Guide
SCJP
Exam 310-065
100% Coverage—360+ Practice Exam Questions
Kathy Sierra
Bert Bates
Three Full MasterExams
Complete Electronic Book
LearnKey


Speed Up the Java Development Process
Java Generics
and Collections
O'REILLY
Maurice Naftalin & Philip Wadler

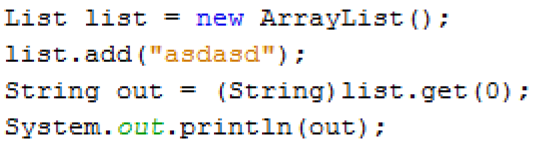
Agenda

- Introduction to Java Generics – definition
- Wrappers, Boxing and unboxing
- Subtyping and Wildcards
- Get and Put Principle
- Wildcard restrictions

Introduction to Generics in Java

Introduction to Generics in Java

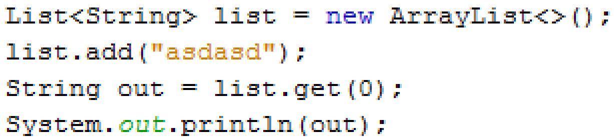
Legacy code



```
List list = new ArrayList();
list.add("asdasd");
String out = (String)list.get(0);
System.out.println(out);
```

Cast is required, without you’ll get compile-time error.

Generic version



```
List<String> list = new ArrayList<>();
list.add("asdasd");
String out = list.get(0);
System.out.println(out);
```

Cast is not required

Introduction to Generics in Java

Generics

An interface or class may be declared to take one or more type parameters, which are written in angle brackets and should be supplied when you declare a variable belonging to the interface or class or when you create a new instance of a class.

- interface List<E>
- class ArrayList<E>

where E is a type parameter

List<String> list = new ArrayList<>();

Diamond operator – type inference

Introduction to Generics in Java

Type parameter vs Type argument – what is the difference?

Interface List<**E**> {

}

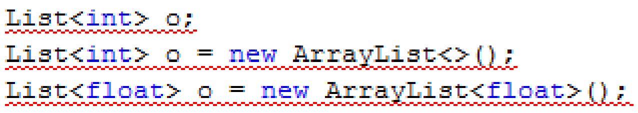
E is **a type parameter**

List<**String**> list = new ArrayList<>();

String is **a type argument**

Wrappers, Boxing and unboxing

Generics can't be used with primitive types



```
List<int> o;
List<int> o = new ArrayList<>();
List<float> o = new ArrayList<float>();
```

Compile-time error

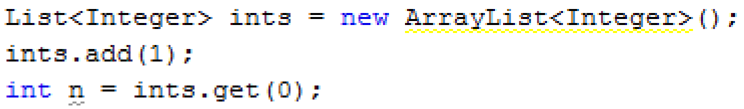
Primitive type	Wrapper class
boolean	Boolean
byte	Byte
char	Character
float	Float
int	Integer
long	Long
short	Short
double	Double

Wrappers, Boxing and unboxing

Boxing and unboxing

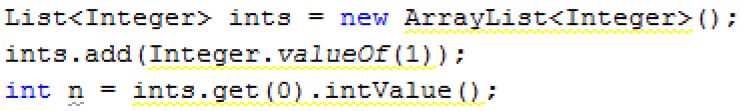
done automatically where appropriate

Example



```
List<Integer> ints = new ArrayList<Integer>();
ints.add(1);
int n = ints.get(0);
```

is equivalent to sequence:

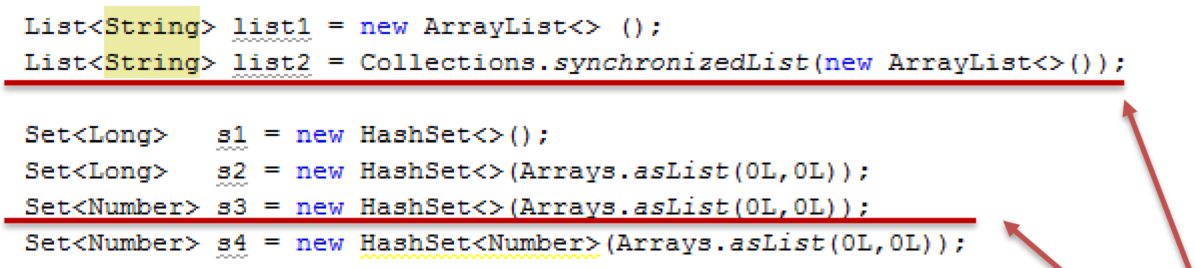


```
List<Integer> ints = new ArrayList<Integer>();
ints.add(Integer.valueOf(1));
int n = ints.get(0).intValue();
```

Introduction to Generics in Java

Type argument inference for instance creation expressions

The automatic deduction of the type arguments in a new-expression



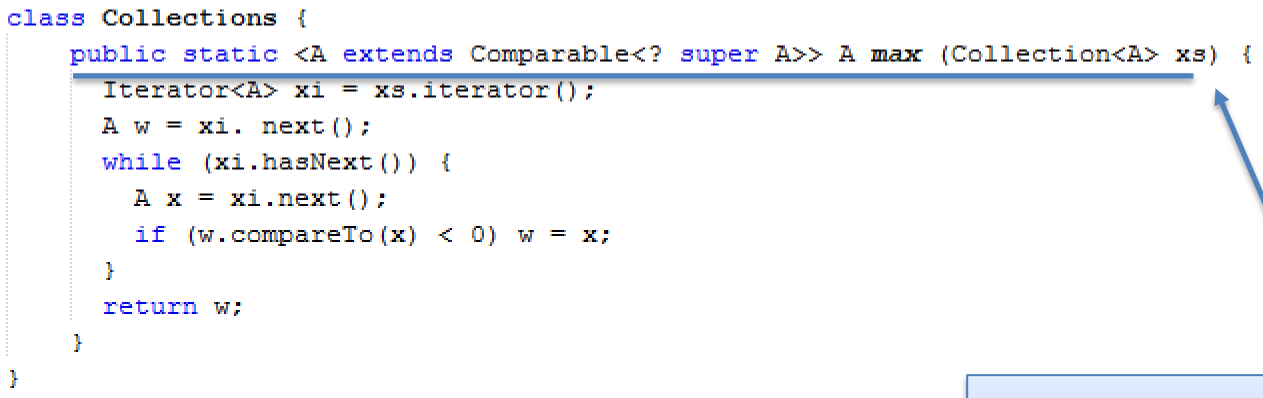
```
List<String> list1 = new ArrayList<> ();
List<String> list2 = Collections.synchronizedList(new ArrayList<>());

Set<Long>    s1 = new HashSet<>();
Set<Long>    s2 = new HashSet<>(Arrays.asList(0L,0L));
Set<Number> s3 = new HashSet<>(Arrays.asList(0L,0L));
Set<Number> s4 = new HashSet<Number>(Arrays.asList(0L,0L));
```

Compile-time error in Java 7, but are fine in Java 8

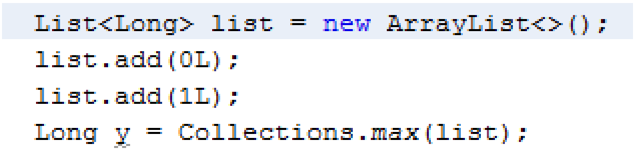
Introduction to Generics in Java

Type argument inference for instance creation expressions

The automatic deduction of the type arguments of a generic method at compile time.

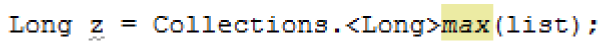


```
class Collections {
    public static <A extends Comparable<? super A>> A max (Collection<A> xs) {
      Iterator<A> xi = xs.iterator();
      A w = xi. next();
      while (xi.hasNext()) {
        A x = xi.next();
        if (w.compareTo(x) < 0) w = x;
      }
      return w;
    }
}
```



```
  List<Long> list = new ArrayList<>();
  list.add(0L);
  list.add(1L);
  Long y = Collections.max(list);
```

Compiler detects type of A



```
Long z = Collections.<Long>max(list);
```

Explicit type argument specification

Introduction to Generics in Java

Type Parameter Naming Conventions (Oracle)

- 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

Google Type Parameter Naming Convention:

- A single capital letter, optionally followed by a single numeral (such as E, T, X, T2)
- A name in the form used for classes, followed by the capital letter T (examples: RequestT, FooBarT).

Introduction to Generics in Java

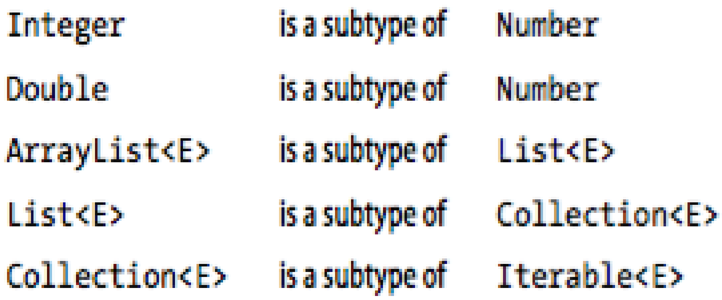
Advantages of Generics:

1. Stronger type checks at compile time.
2. Elimination of casts
3. No runtime overhead.
4. Enabling programmers to implement generic algorithms – write it once and have more free time
5. Reusable code means less bugs

Subtyping and Wildcards

Subtyping and the Substitution Principle

- Subtyping is a key feature of object-oriented languages such as Java.
- In Java, one type is a *subtype* of another if they are related by an extends or implements clause.
- Subtyping is transitive, meaning that if one type is a subtype of a second, the second is a subtype of third, then the first is a subtype of the third
- Examples:

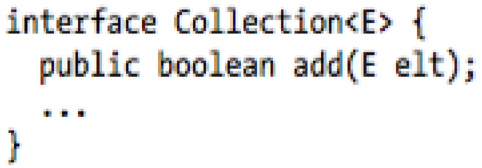


Integer	is a subtype of	Number
Double	is a subtype of	Number
ArrayList<E>	is a subtype of	List<E>
List<E>	is a subtype of	Collection<E>
Collection<E>	is a subtype of	Iterable<E>

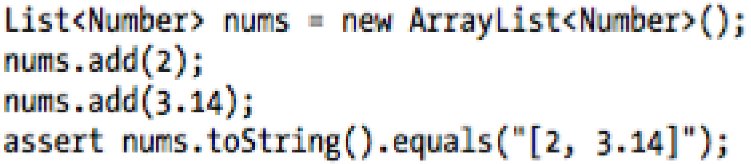
Conversely, **Number** is a *supertype* of **Integer** etc.

Substitution Principle

- a variable of a given type may be assigned a value of any subtype of that type, and a method with a parameter of a given type may be invoked with an argument of any subtype of that type.
- Example:



```
interface Collection<E> {
  public boolean add(E elt);
  ...
}
```



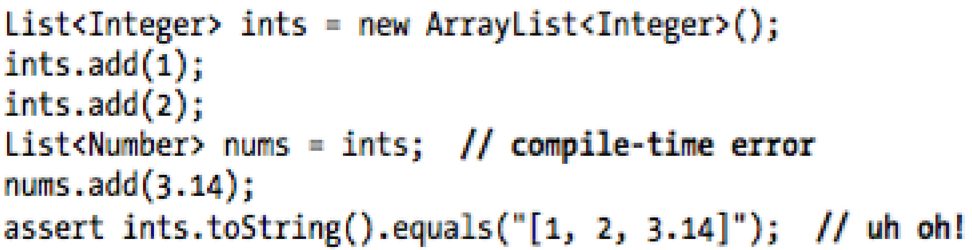
```
List<Number> nums = new ArrayList<Number>();
nums.add(2);
nums.add(3.14);
assert nums.toString().equals("[2, 3.14]");
```

ArrayList<Number> is a subtype of List<Number>

Integer and Double are subtypes of Number

Substitution Principle, cont.

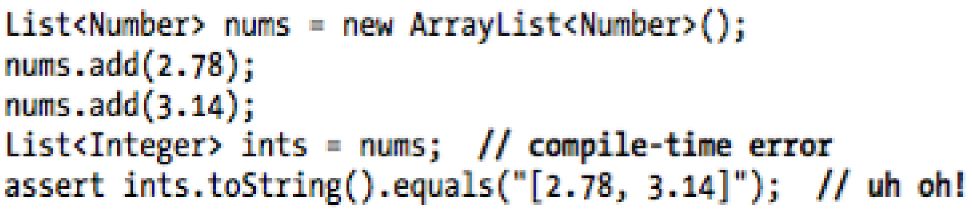
- However, notice the following:



```
List<Integer> ints = new ArrayList<Integer>();
ints.add(1);
ints.add(2);
List<Number> nums = ints;  // compile-time error
nums.add(3.14);
assert ints.toString().equals("[1, 2, 3.14]");  // uh oh!
```

List<Integer> is NOT a subtype of List<Number>

- What about the reverse?

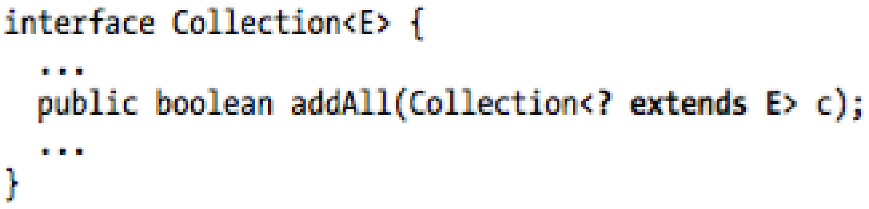


```
List<Number> nums = new ArrayList<Number>();
nums.add(2.78);
nums.add(3.14);
List<Integer> ints = nums;  // compile-time error
assert ints.toString().equals("[2.78, 3.14]");  // uh oh!
```

Nor the other way around

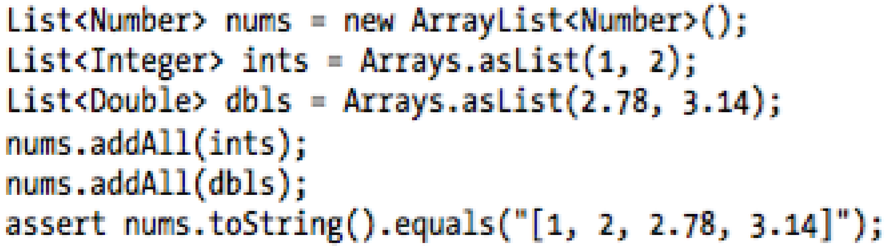
Wildcards with *extends*

- Most likely, till now you have encountered such a method definition:



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

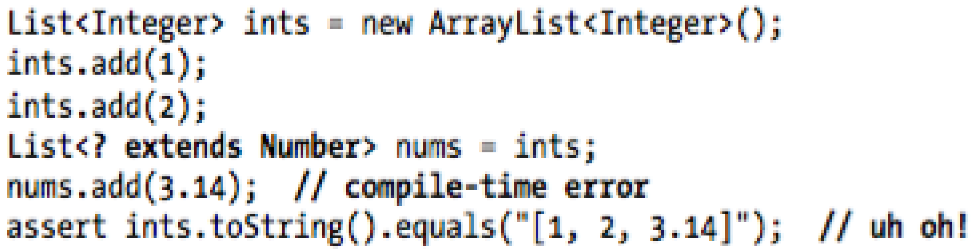
- The phrase in bold (“**? extends E**”) means you could add all members of a collection with elements of any type that is a *subtype* of E
- Example:



```
List<Number> nums = new ArrayList<Number>();
List<Integer> ints = Arrays.asList(1, 2);
List<Double> dbls = Arrays.asList(2.78, 3.14);
nums.addAll(ints);
nums.addAll(dbls);
assert nums.toString().equals("[1, 2, 2.78, 3.14]");
```

Declaring variables with wildcard

- We can also use wildcards when declaring variables, however:



```
List<Integer> ints = new ArrayList<Integer>();
ints.add(1);
ints.add(2);
List<? extends Number> nums = ints;
nums.add(3.14);  // compile-time error
assert ints.toString().equals("[1, 2, 3.14]");  // uh oh!
```

List<? extends Number> can be a List of ANY subtype of Number

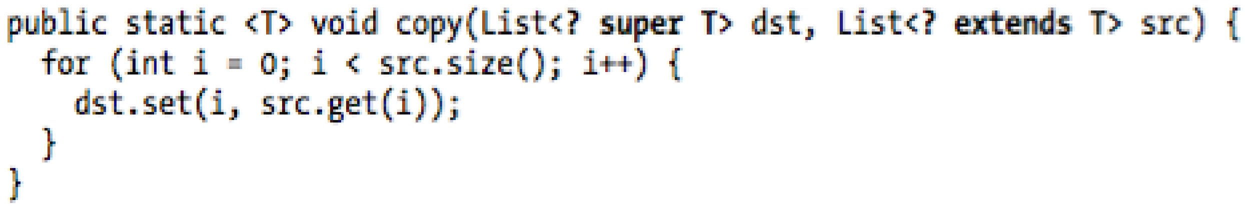
List<? extends Number> is, in fact, List<Integer>

- So what can we do about it?

super to the rescue!

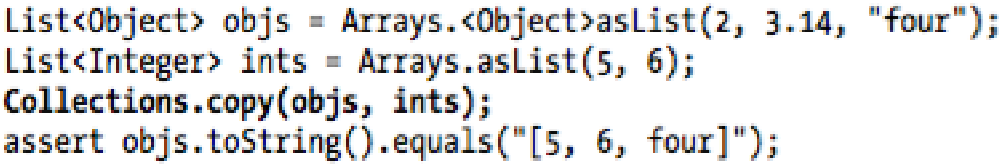
Wildcards with *super*

- Consider this method from **Collections** class:



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

- The phrase “**? super T**” means that the destination list may contain elements of any type that is a *supertype* of T
- Sample call:

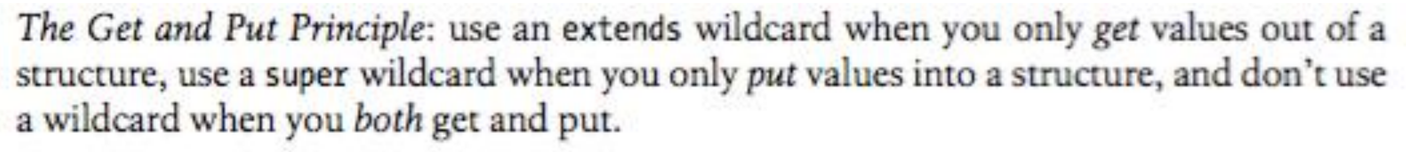


```
List<Object> objs = Arrays.<Object>asList(2, 3.14, "four");
List<Integer> ints = Arrays.asList(5, 6);
Collections.copy(objs, ints);
assert objs.toString().equals("[5, 6, four]");
```

The Get and Put Principle

The Get and Put Principle

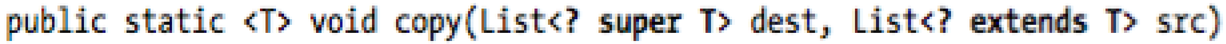
- It is a good practice to use wildcards whenever possible, as it makes your API flexible
- However, how do we know where to use *super* and where to use *extends*?



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 when you *both* get and put.

The Get and Put Principle, cont.

- Recall the previous example:

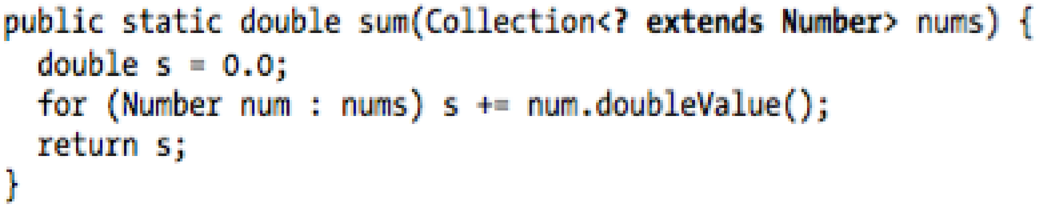


```
public static <T> void copy(List<? super T> dest, List<? extends T> src)
```

- We declare *List<? extends T> src* since we only **get** the values from *src* List
- We declare *List<? super T> dst* since we only **put** values into *dst* List

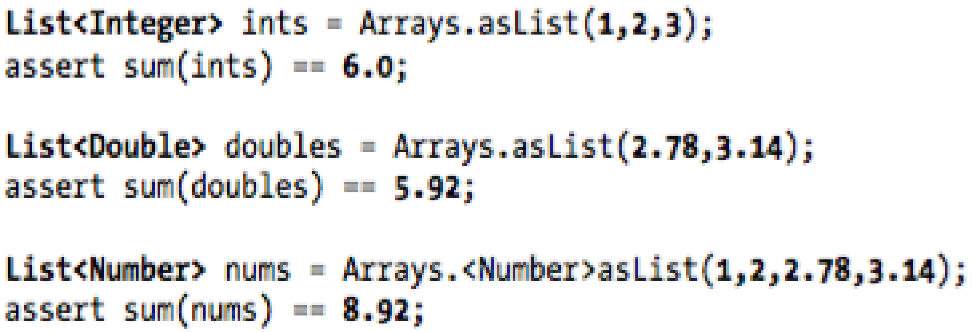
The Get and Put Principle, cont.

- Whenever you use an iterator, you get values out of a structure, so use *extends*:



```
public static double sum(Collection<? extends Number> nums) {
  double s = 0.0;
  for (Number num : nums) s += num.doubleValue();
  return s;
}
```

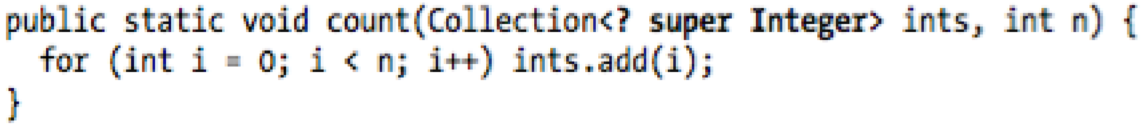
- As it uses *extends*, all the following calls are legal:



```
List<Integer> ints = Arrays.asList(1,2,3);
assert sum(ints) == 6.0;

List<Double> doubles = Arrays.asList(2.78,3.14);
assert sum(doubles) == 5.92;

List<Number> nums = Arrays.<Number>asList(1,2,2.78,3.14);
assert sum(nums) == 8.92;
```

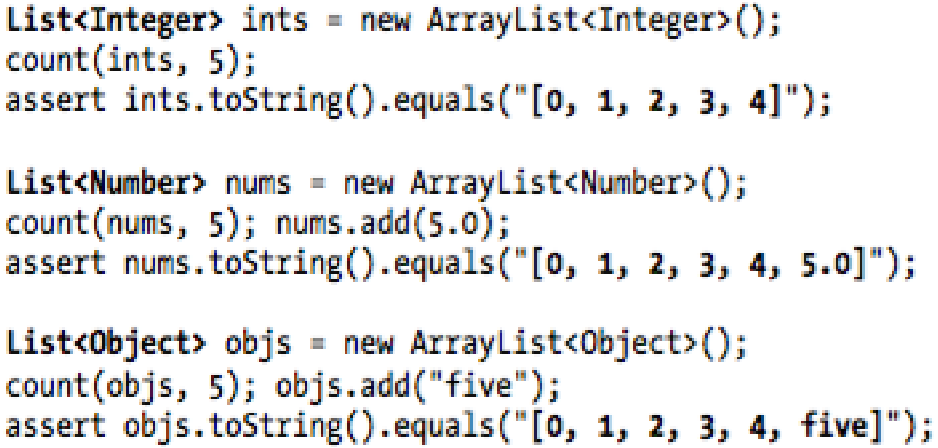
The Get and Put Principle, cont.

- Whenever you use the *add* method, you put values into the structure, so use *super*



```
public static void count(Collection<? super Integer> ints, int n) {
  for (int i = 0; i < n; i++) ints.add(i);
}
```

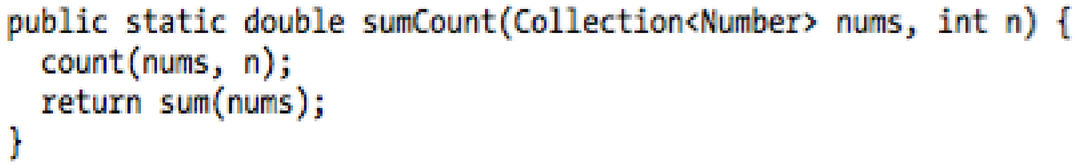
- As it uses *super*, all the following calls are legal:



```
List<Integer> ints = new ArrayList<Integer>();
count(ints, 5);
assert ints.toString().equals("[0, 1, 2, 3, 4]");

List<Number> nums = new ArrayList<Number>();
count(nums, 5); nums.add(5.0);
assert nums.toString().equals("[0, 1, 2, 3, 4, 5.0]");

List<Object> objs = new ArrayList<Object>();
count(objs, 5); objs.add("five");
assert objs.toString().equals("[0, 1, 2, 3, 4, five]");
```

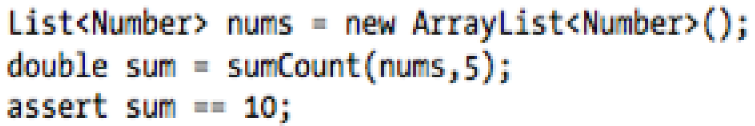
The Get and Put Principle, cont.

- Whenever you both put values into and get values out of the same structure, do not use a wildcard



```
public static double sumCount(Collection<Number> nums, int n) {
  count(nums, n);
  return sum(nums);
}
```

- Since there is no wildcard, the argument must be a collection of **Number**:



```
List<Number> nums = new ArrayList<Number>();
double sum = sumCount(nums,5);
assert sum == 10;
```

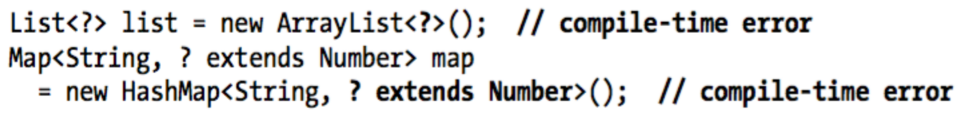
Restrictions on Wildcards

Restrictions on Wildcards

- Although very useful, there are some restrictions when using wildcards. Wildcards may not appear:
  - At the top level in class instance creation expression (*new*)
  - In explicit type parameters in generic method calls
  - In supertypes (*extends* and *implements*)

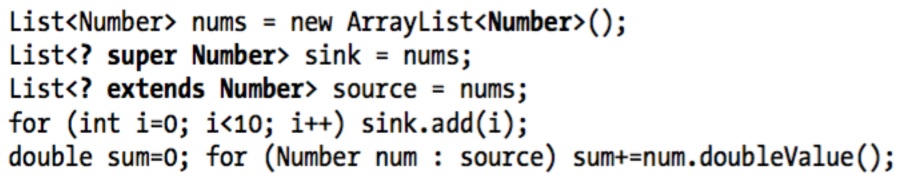
Instance Creation

- In a class instance expression, if the type is a parameterized type, then none of the type parameters may be wildcards



```
List<?> list = new ArrayList<?>();  // compile-time error
Map<String, ? extends Number> map
  = new HashMap<String, ? extends Number>();  // compile-time error
```

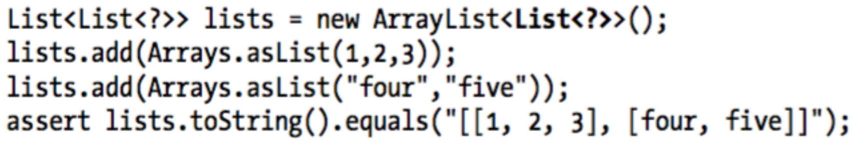
- It is, however, legal to use wildcard when declaring variables



```
List<Number> nums = new ArrayList<Number>();
List<? super Number> sink = nums;
List<? extends Number> source = nums;
for (int i=0; i<10; i++) sink.add(i);
double sum=0; for (Number num : source) sum+=num.doubleValue();
```

Instance Creation, cont.

- Also, only the top-level parameters in instance creation are prohibited
- Nested wildcards are permitted:



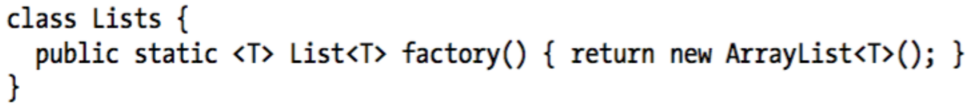
```
List<List<?>> lists = new ArrayList<List<?>>();
lists.add(Arrays.asList(1,2,3));
lists.add(Arrays.asList("four","five"));
assert lists.toString().equals("[[1, 2, 3], [four, five]]");
```

- The reason for that is even though the list of lists is created at a wildcard type, each individual list has a specific type (integers and strings, respectively)

Wildcard – ordinary type relationship is similar to interface – class relationship

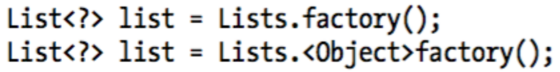
Generic Method Calls

- If a generic method call includes explicit type parameters, those parameters must not be wildcards
- Consider the following method:



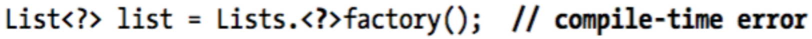
```
class Lists {
  public static <T> List<T> factory() { return new ArrayList<T>(); }
}
```

- It can be called either using implicit or explicit type parameter:



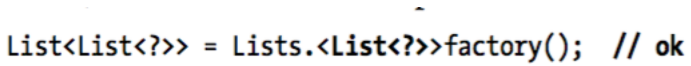
```
List<?> list = Lists.factory();
List<?> list = Lists.<Object>factory();
```

- However, using a wildcard as an explicit type parameter is not allowed



```
List<?> list = Lists.<?>factory();  // compile-time error
```

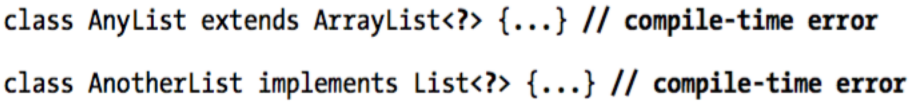
- Nested parameters are allowed though (see previous slides)



```
List<List<?>> = Lists.<List<?>>factory();  // ok
```

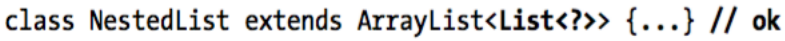
Supertypes

- When a class instance is created, it invokes the initializer for its supertype
- Hence, any restriction that applies to instance creation must also apply to supertypes
- When declaring a class, which supertype (or superinterface) has type parameters, these type cannot be wildcards:



```
class AnyList extends ArrayList<?> {...} // compile-time error

class AnotherList implements List<?> {...} // compile-time error
```

- Again, wildcards in nested type parameters are legal



```
class NestedList extends ArrayList<List<?>> {...} // ok
```