
Collections and Generics



Generics are an essential ingredient in contemporary Java applications. Generics allow you to define classes in a type-agnostic kind of way. The most obvious manifestation of generics is in collection libraries, where you can create a list of strings, a list of persons, a list of bank accounts, and so on.

Contents

1. Overview of collections and generics
2. The need for generics
3. Using collections
4. Defining generic classes
5. Defining generic methods

Annex

- A. Going further with generics



Demo project: `DemoCollectionsGenerics`

2

Section 1 presents a general overview of collections and generics, to get you in the right mindset.

Section 2 emphasizes why generics are important. It describes the problem that generics are meant to resolve.

Section 3 shows how to use Java collection classes. The collection classes are a good example of how to use generics successfully.

Section 4 explains how you can define your own generic classes. This isn't as common as you might imagine, but it's worth knowing nonetheless.

Section 5 introduces the concept of generic methods, which are like type-agnostic algorithms.

The annex at the end of the chapter contains some additional reading on the detailed syntax and rules for generic classes and generic methods.

The demos for chapter are located in the `DemoCollectionsGenerics` project.

1. Overview of Collections

- Collections vs. arrays
- Simple code examples
- Common collection classes / interfaces
- Understanding the classes / interfaces



This section presents a relatively high-level overview of generics and collections, to set the scene before we dive into the technical details later in the chapter.

Collections vs. Arrays

- A collection is an object that holds a group of other objects
 - Some collection types (e.g. `ArrayList`) actually use arrays internally to store the data
- Differences between arrays and collections:
 - Arrays are a Java language feature
 - Collections are standard Java classes, in the `java.util` package
- Arrays are pretty basic
- Collections provide lots of useful methods to manipulate contents
- Arrays are fixed size
- Collections are resizable
- Arrays can store primitive types as well as objects
- Collections store only objects
- Arrays are processed using indexes
- Collections are usually processed without using indexes

4

Earlier in the course, we saw how to use arrays in Java. Arrays are a built-in language feature and allow you to store a fixed-size series of elements of a particular type. In contrast, collection are classes (defined in the `java.util` package), and are resizable. Collection classes are a lot more powerful than arrays, they provide many methods for managing the contents in the collection.

Simple Code Examples

- This code uses arrays:

```
public static void demoUsingArrays() {  
    String[] countries = new String[3];  
    countries[0] = "Norway";  
    countries[1] = "Sweden";  
    countries[2] = "Denmark";  
  
    for (int i = 0; i < countries.length; i++)  
        System.out.println(countries[i]);  
}
```

CollectionsVsArrays.java

- Equivalent code, using collections:

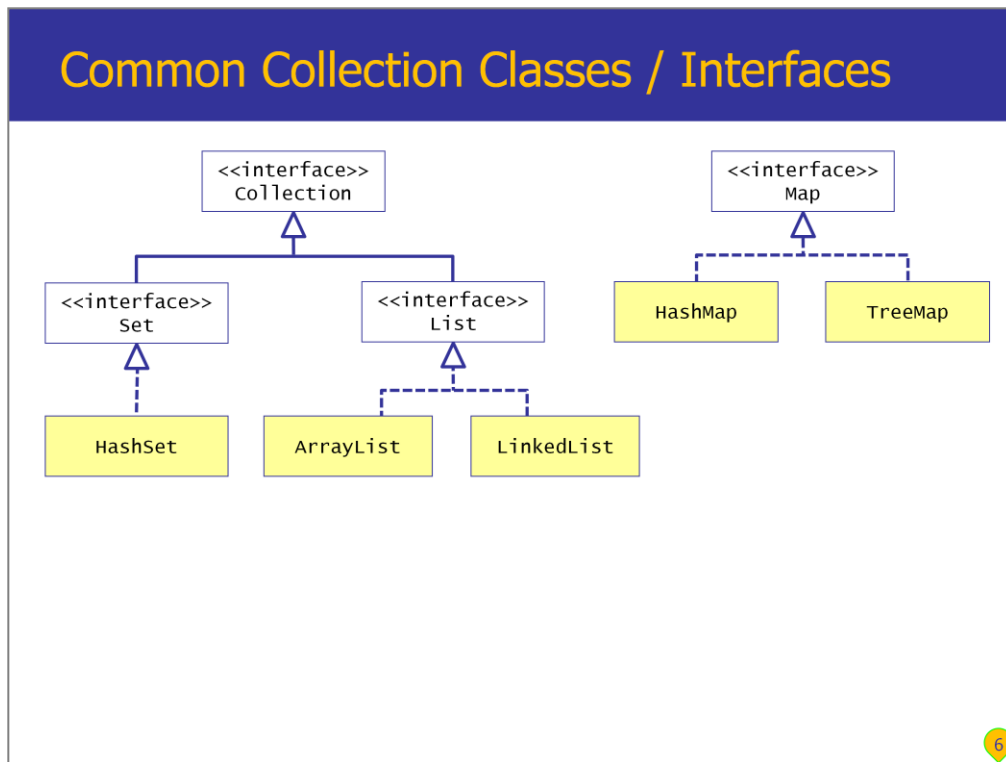
- See next section for discussion of <> generics syntax

```
import java.util.*;  
...  
public static void demoUsingCollections() {  
    ArrayList<String> countries = new ArrayList<String>();  
    countries.add("Norway");  
    countries.add("Sweden");  
    countries.add("Denmark");  
  
    for (String country: countries)  
        System.out.println(country);  
}
```

CollectionsVsArrays.java

5

This slide compares arrays and collections. The upper code box shows a fixed-size array of three strings, and the lower code box shows a resizable collection of strings. The lower code box uses the `ArrayList` collection class, which behaves like a resizable array in memory.



The Java SE library contains several different types of collection classes. These classes are organized into inheritance hierarchies and make use of interfaces to capture common behaviour.

The following slide summarizes the differences between all the collection classes.

Understanding the Classes / Interfaces

- **Collection**
 - Base interface for `Set` and `List` interfaces
- **Set**
 - Set-based collections guarantee element uniqueness
 - E.g. `HashSet` uses hash codes to ensure element uniqueness
- **List**
 - List-based collections store elements sequentially, by position
 - E.g. `ArrayList` stores elements internally as an array
 - E.g. `LinkedList` stores elements internally as a linked list
- **Map**
 - Map-based collections store key/value pairs
 - E.g. `HashMap` arranges items by using hash codes
 - E.g. `TreeMap` arranges items in a tree



Here's a quick comparison of the collection classes:

- `ArrayList` and `LinkedList` are sequential collections. They hold their elements in sequential order, and provide methods to get an element at a particular index position. These two classes both implement the `List` interface, which defines the common sequential-access behaviour. Internally, `ArrayList` holds its elements in contiguous memory, which makes it a good choice if you need to jump around accessing elements at random positions. In contrast, `LinkedList` holds its elements as a linked list, which makes it a very bad choice if you need random access. However, if you need to insert elements in the middle of the list, `LinkedList` is a better choice.
- `HashMap` and `TreeMap` are dictionaries. Each entry has a key and a value. You access entries by key, and obtain the corresponding value. These two classes both implement the `Map` interface, which defines the common lookup behaviour. Internally, `HashMap` uses hashing to help it locate keys quickly, whereas `TreeMap` organizes keys in a binary tree to achieve the same effect.
- `HashSet` is like a bag that doesn't allow duplicates. If you try to insert the same object more than once, only a single reference will be held. The elements are not stored in any particular order; to access elements, you just iterate through the set using a for-each loop.

2. The Need for Generics

- What are generics?
- Example of using raw types
- Problems with using raw types
- Example of using generic types
- Advantages of using generic types
- Type inference in Java SE 7

8

Collections make extensive use of generics, so that's the next topic on our agenda. This section describes what generics are, and explains why they matter.

What are Generics?

- Generics are type-safe classes or methods
 - Parameterized types (must be reference types, not primitives)
 - More efficient and type-safe than non-generic types
 - Enables collections (typically) to know the types of their objects
- Examples of non-generic ("raw") types
 - `LinkedList`
 - `HashMap`
- Examples of generic types
 - `LinkedList<String>`
 - `HashMap<Integer, String>`
- Note:
 - Generics were introduced in Java SE 1.5
 - Definitely the preferred approach nowadays

9

Java allows you to define generic classes and generic methods. Generic classes/methods are type-neutral; they use placeholder names representing the data type they work on. When you create an instance of a generic class, or when you call a generic method, the compiler deduces what actual type to use.

Example of Using Raw Types

- Consider the following example, which uses the raw types

```
import java.util.*;
...
public static void demoRawTypes() {

    System.out.println("\nDemonstrate the use of raw types.");

    Map players = new HashMap();
    players.put(6, "Ferrie Bodde");
    players.put(8, "Darren Pratley");
    players.put(22, "Angel Rangel");

    Set items = players.entrySet();
    Iterator iter = items.iterator();

    while (iter.hasNext()) {
        Map.Entry entry = (Map.Entry)iter.next();
        int number = (Integer)entry.getKey();
        String name = (String)entry.getValue();
        System.out.println(number + "\t: " + name);
    }
}
```

NeedForGenerics.java

10

To illustrate the need for generics, consider the code shown in the slide above. This uses the non-generic version of the `HashMap` collection class, which internally uses `Object` as its data type. (This non-generic version of `HashMap` is a predecessor of the generic `HashMap` class, which we used to use in the days before generics appeared in Java 1.5).

In the example in this slide, when we access elements in the collection, we must cast the items into the correct type. This is because the methods just return vanilla `Object` types.

Problems with Using Raw Types

- The map doesn't remember the types of the keys/values that it contains

- The map just stores `Object` references internally

```
Map players = new HashMap();
```

- This means the compiler cannot detect if client code inserts incorrect types of objects

- This will probably cause run-time errors later...

```
players.put("Twenty three", "Guillem Bauza");
```

- Client code has to cast objects on retrieval

- Because the retrieval methods just return `Object`
- The client code has to remember the real types

```
int number = (Integer)entry.getKey();  
String name = (String)entry.getValue();
```

11

There are numerous problems with the non-generic collection classes, as outlined in the slide here. These problems are all overcome when we use generic collections.

Example of Using Generic Types

- Consider the following example, which uses generic types

```
import java.util.*;
...
public static void demoGenericTypes() {

    System.out.println("\nDemonstrate the use of generic types.");

    Map<Integer, String> players = new HashMap<Integer, String>();
    players.put(6, "Ferrie Bodde");
    players.put(8, "Darren Pratley");
    players.put(22, "Angel Rangel");

    Set<Map.Entry<Integer, String>> items = players.entrySet();
    Iterator<Map.Entry<Integer, String>> iter = items.iterator();

    while (iter.hasNext()) {
        Map.Entry<Integer, String> entry = iter.next();
        int number = entry.getKey();
        String name = entry.getValue();
        System.out.println(number + "\t: " + name);
    }
}
```

NeedForGenerics.java

12

This slide shows the generic equivalent of the previous example.

Note that we now specify exactly what data types to use for the keys and values in the `HashMap`. Also notice that when we retrieve entries from the map, there's no need to do any casting now, because the methods return strongly-typed references rather than just returning `Object`.

Advantages of Using Generic Types

- The generic map remembers the types of the keys/values that it contains



- The map stores type-safe references internally

```
Map<Integer, String> players = new HashMap<Integer, String>();
```

- This means the compiler can detect if client code inserts incorrect types of objects

- No possibilities of run-time errors later...

```
players.put("Twenty three", "Guillem Bauza");
```

 **Compiler error** 

- Client code doesn't cast objects on retrieval

- Because the retrieval methods return type-safe types
- The client code doesn't have to remember the real types

```
int number = entry.getKey();  
String name = entry.getValue();
```

13

Here we see the benefits of using generics. All of the shortcomings we identified with the non-generic collections are addressed and overcome when we use generic collections.

You should always use generic collections rather than non-generic collections.

Type Inference in Java SE 7

- Java SE 7 allows you to omit type information when you invoke a constructor

- Just use empty <> instead

```
Map<String, Integer> asiaDialCodes = new HashMap<>();
```

- Example:

- See `DemoGenericsTypeInference.java`

14

Java SE 7 simplifies generic usage slightly, by allowing you to omit the type parameters between <> when you call the constructor for a generic class. This can be a big help, especially when dealing with complicated type parameters!

3. Using Collections

- Using ArrayList
- Using LinkedList
- Using HashSet
- Using HashMap
- Using TreeMap

15

This section takes a closer look at the capabilities of the various collection classes in Java. We'll be looking at the generic versions of these classes, obviously.

Using ArrayList

- Constructors:
 - Default capacity
 - Specific capacity
 - From another Collection
- Some useful methods:
 - `add()`, `addAll()`, `set()`
 - `size()`, `get()`, `indexOf()`, `contains()`, `isEmpty()`
 - `remove()`, `removeAll()`, `clear()`
 - `removeFirst()`, `removeFirstOccurrence()`
 - `removeLast()`, `removeLastOccurrence()`
 - `trimToSize()`
- Example:
 - See `UsingCollections.java`, `demoArrayList()`

16

Here's a quick summary of the `ArrayList` class:

- Uses an array internally
- Automatically resizes when you add elements
- Fast for lookups, not so fast for insertions

For an example of how to use `ArrayList`, see the `demoArrayList()` method in the `usingCollections.java` source file.

For full details about the `ArrayList` class, see the following web site:

- <http://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html>

Using LinkedList

- Constructors :
 - Empty list
 - From another Collection
- Some useful methods, in addition to those of ArrayList:
 - `addFirst()`, `addLast()`
 - `descendingIterator()`
 - `getFirst()`, `getLast()`, `lastIndexOf()`
 - `peek()`, `peekFirst()`, `peekLast()`
 - `poll()`, `pollFirst()`, `pollLast()`
 - `push()`, `pop()`
- Example:
 - See `UsingCollections.java`, `demoLinkedList()`

17

Here's a quick summary of the `LinkedList` class:

- Uses a linked list internally
- Automatically resizes when you add elements
- Fast for linear traversal, not so fast for ad-hoc access
- More sophisticated than `ArrayList`

For an example of how to use `LinkedList`, see the `demoLinkedList()` method in the `usingCollections.java` source file.

For full details about the `LinkedList` class, see the following web site:

- <http://docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html>

Using HashSet

- Constructors:
 - Default capacity
 - Specific capacity, and optionally a specific load factor
 - From another Collection
- Some useful methods (not many):
 - `add()`, `addAll()`
 - `size()`, `contains()`, `isEmpty()`
 - `remove()`, `removeAll()`, `clear()`
- Example:
 - See `UsingCollections.java`, `demoHashSet()`

18

Here's a quick summary of the `HashSet` class:

- Uses an item's hash code to determine placement internally
- Is unordered
- Allows nulls
- Offers constant-time performance for basic operations

For an example of how to use `HashSet`, see the `demoHashSet()` method in the `UsingCollections.java` source file.

For full details about the `HashSet` class, see the following web site:

- <http://docs.oracle.com/javase/7/docs/api/java/util/HashSet.html>

Using HashMap

- Constructors:
 - Default capacity
 - Specific capacity, and optionally a specific load factor
 - From another Map
- Some useful methods:
 - `put()`, `putAll()`
 - `keySet()`, `values()`
 - `size()`, `containsKey()`, `containsValue()`, `isEmpty()`
 - `remove()`, `clear()`
- Example:
 - See `UsingCollections.java`, `demoHashMap()`

19

Here's a quick summary of the `HashMap` class:

- Stores key / value pairs (each pair is an `Entry<K,V>`)
- Is unsorted
- Allows null keys and values
- Offers constant-time performance for basic operations

For an example of how to use `HashMap`, see the `demoHashMap()` method in the `UsingCollections.java` source file.

For full details about the `HashMap` class, see the following web site:

- <http://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html>

Using TreeMap

- **TreeMap**
 - Stores key / value pairs (each pair is an `Entry<K,V>`)
 - Is sorted
 - Offers constant-time performance for basic operations
 - Is much more powerful than `HashMap`...
- Take a look in the JavaDoc and try out some of its functionality 😊

20

This slide summarizes the capabilities of `TreeMap`. It's quite similar to `HashMap` really!

For full details about the `TreeMap` class, see the following web site:

- <http://docs.oracle.com/javase/7/docs/api/java/util/TreeMap.html>

4. Defining Generic Classes

- Overview
- Defining a generic class
- Implementing methods in a generic class
- Using a generic class
- Additional techniques
- Erasures

21

In this section we describe how to define your own generic collection classes. We'll also see how to define generic methods in the next section.

Overview

- You can create your own generic types
 - Generic classes and generic interfaces
- A generic type has one or more type parameters
 - Enclosed in <> angled brackets
 - The name of the type parameter is insignificant
 - The type parameter always represents a reference type (class or interface), never a primitive type (e.g. int)
- For example:

```
class CyclicList<T> {  
    ...  
}
```

22

You can define your own generic classes, using the syntax shown in the slide. In this example, T is a type parameter; it's a placeholder for an actual type in your class definition.

Note that you can call the type parameter anything you like. We chose to call it T here, whereas the Java generic collections classes tend to use E.

Defining a Generic Class

- Here's a sample generic class

```
import java.util.*;
...
class CyclicList<T> {

    // Define members. Note we can use the type parameter here.
    private ArrayList<T> elements;
    private int currentPosition;
    private int maxElements;

    // Define a constructor.
    CyclicList(int size) {
        elements = new ArrayList<T>();
        currentPosition = 0;
        maxElements = size;

        // Pre-populate collection with nulls.
        for (int i = 0; i < maxElements; i++) {
            elements.add(null);
        }
    }
    ...
}
```

DefiningGenericTypes.java

23

Over this and the following slides, we show how to implement a generic class to represent a cyclic buffer, a special kind of collection where elements wrap around at the end and start filling at the beginning again.

Note the use of T here, rather than specific types. This means the class is applicable to any type, which obviously makes it more useful.

Implementing Methods in a Generic Class

- Methods in a generic class can use the type parameter

```
public void insert(T item) {
    elements.set(currentPosition, item);
    if (++currentPosition == maxElements) {
        currentPosition = 0;
    }
}

public T getItemAt(int position) throws IndexOutOfBoundsException {
    if (position >= maxElements) {
        throw new IndexOutOfBoundsException("Indexed beyond end of CyclicalList");
    }
    else {
        return elements.get(position);
    }
}

public void display() {
    for(T item: elements) {
        if (item == null) {
            System.out.print("[Null] ");
        }
        else {
            System.out.print(item.toString() + " ");
        }
    }
}
```

DefiningGenericTypes.java

24

We continue our definition of the `CyclicalList<T>` class, to show how to insert, access, and display elements in the collection. Basically this is regular code, except that it uses `T` rather than a specific type.

Using a Generic Class

- When you instantiate a generic class, you must specify a type parameter
 - The type parameter must be a class/interface
 - This is called "type substitution"
- When you use the generic class, it's typesafe!
 - No need for ugly casts

```
CyclicList<Integer> lotteryNumbers = new CyclicList<>(6);
lotteryNumbers.insert(19);
lotteryNumbers.insert(1);
lotteryNumbers.insert(2);
lotteryNumbers.insert(7);
```

```
int lotteryNumber0 = lotteryNumbers.getItemAt(0);
System.out.println("Lottery number 0 is: " + lotteryNumber0);
```

```
System.out.print("collection: ");
lotteryNumbers.display();
```

DefiningGenericTypes.java

```
Lottery number 0 is: 19
Collection: 19 1 2 7 [Null] [Null]
```

25

When you create an instance of a generic class, you must supply actual type parameters enclosed in <>. You must do this twice actually:

- When you create your object, e.g. `new CyclicList<Integer>`
- When you declare the variable, `CyclicList<Integer> lotteryNumbers`

Note that you cannot specify primitive types (such as `int`) for generic type parameters. You must specify the corresponding wrapper class instead (e.g. `Integer`).

Additional Techniques

- A generic type can have multiple type parameters

```
public class MyMap<K,V> {  
    ...  
}  
  
// Usage:  
MyMap<String, Date> birthdays = new MyMap<>();
```

- A generic type can contain a nested type

```
public class MyMap<K,V> {  
    static interface Entry<K,V> {...}  
    ...  
}  
  
// Usage:  
MyMap<String, Date> birthdays = new MyMap<>();  
MyMap.Entry<String,Date> entry;
```

26

You can define multiple type parameters in a generic class. The upper code box shows how to do this.

You can also define nested types inside a generic class. The lower code box shows how to do this. We'll discuss nested ("inner") classes later in the course.

Erasures

- Note that generics are a compile-time-only mechanism
 - Provides type information for the compiler, to make sure your code is type safe
- The compiler generates non-generic bytecode!
 - At run-time, type information is not available
 - E.g. if your code uses `Map<Integer,String>`, at run-time the bytecodes will use just `Map`
- This is called "erasure"
 - I.e. the compiler erases type information
 - Why? For compatibility with pre-generic code

27

To close this section, we point out what happens when you compile your code that contains generic types. The compiler erases type information in the Java byte codes; in other words, generics are a compile-time language mechanism, rather than a run-time JVM mechanism.

4. Defining Generic Methods

- Overview
- Implementing generic methods
- Invoking generic methods

28

As well as defining generic classes, you can also define generic methods. A generic method is like a type-agnostic algorithm.

Overview

- The previous section described generic types
 - A generic type contains type parameter(s) that affects the entire class/interface
- It's also possible to define generic methods
 - A generic method contains type parameter(s) that affect one particular method only
 - This means a non-generic class can contain a mixture of generic and non-generic methods
- Syntax for generic methods:

```
accessSpecifier <T> returnType methodName(methodArgs) {  
    ...  
}
```

29

From time to time, you might want to implement a method that can be flexible enough to deal with any kind of parameter. For example, you might want to write a `sort()` method that can theoretically sort any kind of array.

To achieve the appropriate degree of genericity, you can declare the method as a generic method. The code box in the slide shows the required syntax.

Implementing Generic Methods

- The following non-generic class contains two generic methods
 - For simplicity, the methods are `static` in this example

```
class MyUtilityClass {  
    public static <T> void displayType(T obj) {  
        System.out.println("Object type: " + obj.getClass().getName());  
    }  
  
    public static <T> void displayArray(T[] array) {  
        System.out.print("Array items: ");  
        for(T item: array) {  
            System.out.print(item.toString() + " ");  
        }  
    }  
}
```

DefiningGenericMethods.java

30

This example shows a class that contains two generic methods. The class itself isn't generic, because it doesn't need to be. Instead, we define both its methods as generic, so that they can display details for any type, and display an array of any type.

Invoking Generic Methods

- Invoke generic methods in the same way as you invoke non-generic methods
 - The argument types implicitly provide the type information

```
MyUtilityClass.displayType(new Date());  
  
Integer[] array = new Integer[] {10, 20, 30};  
MyUtilityClass.displayType(array[0]);  
MyUtilityClass.displayType(array);  
MyUtilityClass.displayArray(array);
```

DefiningGenericMethods.java

```
Object type: java.util.Date  
Object type: java.lang.Integer  
Object type: [Ljava.lang.Integer;  
Array items: 10 20 30
```

31

When you call a generic method, the compiler looks at the actual parameters you've passed into the method and deduces what type it's dealing with. You don't need to use any <> syntax explicitly in your call.

Any Questions?



32

Annex: Going Further with Generics

- Specifying a hierarchy argument
- Generic and subtyping
- Using wildcards

33

This annex section provides some more technical information about the syntax for generics. Feel free to browse this section if you have time, and ask the instructor if you have any questions.

Specifying a Hierarchy Argument (1)

- A type parameter can specify that substitution types must be a specific type (or subtype)

- General syntax for a constrained type parameter:

```
<T1 extends T2 & T3 & T4>
```

- This means the type parameter T1 must extend (or implement) all of the other types
- Notice in particular the & notation!

- Reasons for specifying a hierarchy argument:

- Ensures baseline functionality of type parameter
- Enables generic class/method to utilize baseline functionality

Specifying a Hierarchy Argument (2)

- Example of specifying a hierarchy argument:

```
class Statistics<T extends Number> {  
    private T[] numbers;  
  
    public Statistics(T[] nos) {  
        numbers = nos;  
    }  
  
    public double calcAverage() {  
        double total = 0.0;  
        for (T number: numbers) {  
            total += number.doubleValue();  
        }  
        return total/numbers.length;  
    }  
}  
  
// Usage:  
Integer[] ints = new Integer[] {43, 42, 34};  
Statistics<Integer> ages = new Statistics<>(ints);  
System.out.println("Average age: " + ages.calcAverage());
```

GoingFurtherWithGenerics.java

Generics and Subtyping (1)

- Generic types are substitutable if the type parameter is the same
 - E.g. an `ArrayList<Number>` object can be assigned to a `List<Number>` variable
 - An `ArrayList<Number>` "is a kind of" `List<Number>`

```
// The following statement is OK  
List<Number> mylist = new ArrayList<>();
```

```
// The following statements are OK. Integer, Double, Long etc are subtypes of Number  
mylist.add(100);  
mylist.add(3.14);  
mylist.add(300L);
```

Generics and Subtyping (2)

- Generic types are not substitutable if the type parameter is different
 - E.g. an `ArrayList<Integer>` is not a kind of `List<Number>`
 - Even though `Integer` is a kind of `Number`!!!

```
// This statement is OK
List<Integer> li = new ArrayList<>();

// This statement causes compiler error: can't assign List<Integer> to List<Number>.
List<Number> ln = li;

// If it was ok, the following could happen...
ln.add(3.14);           // We could add any kind of Number into ln (therefore li).
Integer i = li.get(0);  // What would happen here, then???
```

Using Wildcards (1)

- There are three ways to use wildcards (?)
 - To denote a family of subtypes of some type Type (this is the most useful wildcard syntax):

```
? extends Type
```

- To denote a family of supertypes of type Type:

```
? super Type
```

- To denote the set of all types:

```
?
```

Using Wildcards (2)

? extends Type

■ Example of ? extends Type

- Use wildcards to specify a list-of-any-kind-of-number

```
public double calcAverage(List<? extends Number> numbers) {  
    double total = 0.0;  
    for (Number n: numbers) {  
        total += n.doubleValue();  
    }  
    return total / numbers.size();  
}
```

Using Wildcards (3)

? super Type

■ Example of ? super Type

- Use wildcards to sort a list into ascending order
- All elements in the list must implement the Comparable interface

```
public static <T extends Comparable<? super T>> void sort(List<T> list) {  
    // Convert the list to an array.  
    Object a[] = list.toArray();  
  
    // Sort the array (this will call compareTo() on array elements).  
    Arrays.sort(a);  
  
    // Set each item in the list to be the (sorted) array element.  
    ListIterator<T> i = list.listIterator();  
    for(int j=0; j < a.length; j++) {  
        i.next();  
        i.set((T)a[j]);  
    }  
}
```


Using Wildcards (4)

?

- Example of ?

- Use wildcards to specify a collection of any type

```
public static void printCollection(Collection<?> c) {  
    for (Object item : c) {  
        System.out.println(item);  
    }  
}
```

41