Lists, Maps, and Sets

Topics in This Section

- **Lists**
  - Ordered collections that can stretch or shrink
- **Using generic types**
  - Putting types in <angle brackets>
- **Maps**
  - Associating values with keys
- **Sets**
  - Unordered collections
- **A few techniques needed along the way**
  - Arrays.asList
  - static blocks
  - Preview of lambda expressions

Lists

Lists - Overview

- **Goal**
  - You want to make an ordered collection of objects
- **Problem: you cannot or do not want to use an array**
  - Even after you get the first few elements, you don't know how many more you will have
  - You want to easily add, remove, or search for elements later
- **Solution**
  - Use ArrayList or LinkedList: they stretch as you add elements to them
    - The two options give the same results for the same operations, but potentially differ in performance
- **Minor note**
  - Although the size of Java arrays must be known at the time that you allocate them, Java arrays are more flexible than C++ arrays since the size does not need to be a compile-time constant

ArrayList and LinkedList: Most-Basic Methods

- **Create empty list**
  - `List<Type> myList = new ArrayList<>();`
  - `List<Type> myList = new LinkedList<>();`
- **Add entry to list**
  - `myList.add(value);`
  - `myList.add(index, value);`
- **Retrieve nth element**

```
Type value = myList.get(index);
```

- **Print the list elements**

```
System.out.println(myList);
```

Simple Example: Summary

```
import java.util.*;

public class SimpleList {
  public static void main(String[] args) {
    List<String> words = new ArrayList<>();
    words.add("hi");
    words.add("hello");
    words.add("hola");
    System.out.println("First word: " + words.get(0));
    System.out.println("All words: " + words);
  }
}
```

```
First word: hi
All words: [hi, hello, hola]
```

Simple Example: Comments

```
import java.util.*;

public class SimpleList {
  public static void main(String[] args) {
    List<String> words = new ArrayList<>();
    words.add("hi");
    words.add("hello");
    words.add("hola");
    System.out.println("First word: " + words.get(0));
    System.out.println("All words: " + words);
  }
}
```

You don't usually worry too much about import statements. Just type in List<String> ... in your code, and when Eclipse puts a red X on the line about List being unrecognized, you click on the lightbulb or hit Control-1, and Eclipse offers to insert the imports for you. But, in this case, be careful, because there is also a List class in the java.awt package (it is a graphical list box). So, be sure you have java.util, not java.awt.

Some developers prefer to list classes one at a time, and do this:

```
import java.util.List;
import java.util.ArrayList;
```

There is no difference in performance, but some prefer to list all the classes individually for documentation, and others reason that no human looks at the import statements anyhow, and so they use the * form.

```
First word: hi
All words: [hi, hello, hola]
```

Simple Example: Comments

```
import java.util.*;

public class SimpleList {
  public static void main(String[] args) {
    List<String> words = new ArrayList<>();
    words.add("hi");
    words.add("hello");
    words.add("hola");
    System.out.println("First word: " +  words.get(0));
    System.out.println("All words: " + words);
  }
}
```

You say List (the interface type) instead of ArrayList (the concrete class) here so that you do not accidentally use one of the few methods that is specific to ArrayList. That way, if you decide to change to LinkedList for performance reasons, all the rest of your code stays the same, and the output is unchanged. And of course if you make a method that accepts a list, it is doubly important that you declare the parameter as List<Blah>, not ArrayList<Blah>, so that the same method can also be used for another type of List.

```
First word: hi
All words: [hi, hello, hola]
```

Simple Example: Comments

```
import java.util.*;

public class SimpleList {
  public static void main(String[] args) {
    List<String> words = new ArrayList<>();
    words.add("hi");
    words.add("hello");
    words.add("hola");
    System.out.println("First word: " + words.get(0));
    System.out.println("All words: " + words);
  }
}
```

Because you said List<String>, Java checks at compile time that all calls to "add" are Strings.

```
First word: hi
All words: [hi, hello, hola]
```

Simple Example: Comments

```
import java.util.*;

public class SimpleList {
  public static void main(String[] args) {
    List<String> words = new ArrayList<>();
    words.add("hi");
    words.add("hello");
    words.add("hola");
    System.out.println("First word: " + words.get(0));
    System.out.println("All words: " + words);
  }
}
```

Lists (but not arrays) have a builtin toString method that shows the List values separated by commas. So, you just pass the List variable to println.

```
First word: hi
All words: [hi, hello, hola]
```

The Arrays Utility Class

Arrays.asList

- **Idea**
  - Takes a number of individual entries or an array, and produces a List. Very useful for practice and testing, but not used often in real applications because you cannot modify (add to or remove from) the resultant list.

- **Example: passing in individual entries**

```
List<String> wordList1 = Arrays.asList("word1", "word2", ...);
```

- **Example: passing in array**

```
String[] wordArray = { "word1", "word2", ...};
List<String> wordList2 = Arrays.asList(wordArray);
```

Other Static Methods in Arrays Class

- **Arrays.sort**
  - Sorts an array based on a comparison function. This is same comparison function shown later with the sort method of List.

```
Arrays.sort(someArray, comparisonFunction);
```

  - Details on comparison function (Comparator) in first tutorial section on lambdas, but the idea is that, given two elements, the function returns negative, positive, or zero depending whether first of the two should go first in sorted array, second should go first, or they are tied. See the example with list sorting in the upcoming subsection on advanced list methods. That example sorts the strings based on their lengths.
- **More**
  - Arrays.binarySearch, Arrays.copyOf, Arrays.deepEquals, Arrays.fill, Arrays.parallelSort, Arrays.setAll, Arrays.stream

Iterating Over List Elements

Overview

- **for(ElementType element: list) { ... }**
  - Since List implements Collection, and Collection implements Iterable, you can use the same loop style as we previously used for arrays
- **list.forEach(function)**
  - You can call a function on each element of the List
    - Very briefly shown on next slide; covered in detail in the first lecture on streams
- **list.stream()**
  - This lets you treat a List<ElementType> as a Stream<ElementType>
  - Streams have map, filter, reduce, and much more
    - Not covered here; covered in streams lectures

Example (both give same output)

- for(ElementType element: list) { ... }

```
public static void main(String[] args) {
  List<String> words = Arrays.asList("hi", "hello","hola");
  for(String word: words) {
    System.out.println(word);
  }
}
```

```
hi
hello
hola
```

- list.forEach(function)

```
public static void main(String[] args) {
  List<String> words = Arrays.asList("hi", "hello", "hola");
  words.forEach(System.out::println);
}
```

Read this as "pass each element of the List to println". System.out::println is a method reference, and method references are discussed in detail in the tutorial sections on lambda expressions.

The first section on streams compares and contrasts these two iteration approaches.

Motivation: ArrayList vs. LinkedList

Motivation: Lists vs. Arrays

- **Arrays**
  - You must know the final size at the time the array is built (but not necessarily at compile time, as in C and C++)
    - Cannot stretch or shrink
  - Only a few useful static methods apply to arrays (in Arrays class)
- **Lists**
  - Can stretch and shrink
  - Have several convenient methods (but less than Streams)
- **When to use Lists**
  - When, even after some elements are added, you are not sure how many more you will add later
    - In previous example, if you knew that you would never add more entries besides “hi”, “hello”, and “hola”, it might have been just as easy to use an array
  - You plan to use contains, replaceAll, forEach, or other List methods

Motivation: ArrayList vs. LinkedList

	Array with Copying (ArrayList)	List of Pointers (LinkedList)
Insert at beginning	O(N)	O(1)
Insert at end	O(1) if space O(N) if no space O(1) amortized time	O(1)
Access Nth element	O(1)	O(N)

Hang on to your hats! We will walk carefully through what this all means in the lecture. But the whole reason that Java supplies both LinkedList and ArrayList is because of the different performance characteristics, so understanding this slide is the key to deciding which class to use in real life.

Using
Generic Types

Overview

- **Using generics in existing classes**
  - Specify the type you want in <angle brackets>
    - Simple; covered here
- **Autoboxing**
  - You can declare the type as a wrapper type for numbers (e.g., List<Integer>), and Java will convert automatically if you insert primitives (e.g., int) or remove and assign to a primitive
    - Simple; covered here
- **Using generics in your classes and methods**
  - Lets you make your code more reusable
    - More difficult; covered in next tutorial section

Using Generic Types in Existing Classes

- **Find a data structure that accepts Object(s)**
  - ArrayList, LinkedList, HashMap, HashSet, Stack
- **Declare the data structure with the type(s) in angle brackets immediately after type name**
  - `List<String> names = new ArrayList<>();`
  - `Map<String,Person> employees = new HashMap<>();`
- **Insert objects of the appropriate type**
  - `names.add("Some String");`
  - `employees.put(person.getEmployeeId(), person);`
- **No typecast required on removal**
  - `String firstName = names.get(0);`
  - `Person p1 = employees.get("a1234");`

Lists: Modern Approach (Java 7 and Later)

```
public class RandomList {
  public static void main(String[] args) {
    List<String> entries = new ArrayList<>();
    double d;
    while((d = Math.random()) > 0.1) {
      entries.add("Value: " + d);
    }
    for(String entry: entries) {
      System.out.println(entry);
    }
  }
}
```

This is called the diamond operator. Because you declare the variable as List<String>, Java infers that the type here must be String, so you do not need to repeat the declaration. However, you must do ArrayList<>(), not ArrayList().

This tells Java your list will contain only strings. Java will check at *compile* time that all additions to the list are Strings. You can then use the simpler looping construct because Java knows ahead of time that all entries are Strings.

Representative Output

```
Value: 0.3626736654146729
Value: 0.7709503776509984
Value: 0.5290059300284917
Value: 0.5864932616619523
Value: 0.9863629365281706
Value: 0.5860043342250439
Value: 0.7855437155134938
```

Lists: Old Approach (Java 5 and 6)

```
public class RandomListOld {
  public static void main(String[] args) {
    List<String> entries = new ArrayList<String>();
    double d;
    while((d = Math.random()) > 0.1) {
      entries.add("Value: " + d);
    }
    for(String entry: entries) {
      System.out.println(entry);
    }
  }
}
```

Before Java 7, you had to repeat the type here.

Lists: Obsolete Approach (Java 1.4 and Earlier)

```
public class RandomListVeryOld {
  public static void main(String[] args) {
    List entries = new ArrayList();
    double d;
    while((d = Math.random()) > 0.1) {
      entries.add("Value: " + d);
    }
    String entry;
    for(int i=0; i<entries.size(); i++) {
      entry = (String)entries.get(i);
      System.out.println(entry);
    }
  }
}
```

Never do this in modern Java! The modern approach is more typesafe (checks insertions at compile time), more convenient (easier looping; no typecasts), and has the same performance (no extra type checking is performed at run time).

If you do not use generic types, you have to remember that inserts should be Strings. If you insert another type, it will compile, but then the typecast below will fail at *run* time.

If you do not use generic types, you have to use the less convenient looping format, then do an explicit typecast.

Autoboxing

Overview

- **Objects vs. primitives**
  - Primitives are stored directly. Objects use pointers (references).
  - Primitives start with lowercase letters (int, double, boolean, etc.). Object classes start with uppercase letters (Integer, Double, Boolean, String, etc.).
- **List, Map, etc. expect Object**
  - So you cannot actually store int, double, and other primitives. You cannot declare the List to take primitives (e.g., List<int> is illegal)
  - These data structures can only store wrapper types (Integer, Double, etc.)
- **Java converts automatically**
  - You can assign int to Integer or Integer to int
  - You can insert an int into a List<Integer>
  - You can assign an entry from a List<Integer> to an int

Example

```
public class Autoboxing {
  public static void main(String[] args) {
    List<Integer> nums = new ArrayList<>();
    int[] values = { 2, 4, 6 };
    for(int value: values) {
      nums.add(value);
    }
    System.out.println("List: " + nums);
    int secondNum = nums.get(1);
    System.out.println("Second number: " + secondNum);
  }
}
```

```
List: [2, 4, 6]
Second number: 4
```

More List Methods

ArrayList and LinkedList: More Basic Methods

- **Check if element is in list**
  - contains(element)
    - Uses equals, not ==
- **Remove elements**
  - remove(index)
    - Removes entry at that index. Error if index does not exist.
  - remove(element)
    - Removes *first* entry that matches (equals) the element. Nothing happens if there is no match.
  - removeAll(collection)
    - Calls remove (as above) on each member of collection
- **Find the number of elements**
  - size()

More Basic Methods: Code

```
public static void main(String[] args) {
  List<String> words = new ArrayList<>();
  // List<String> words = new LinkedList<>();
  words.add("hi");
  words.add("hello");
  words.add("hola");
  System.out.println("List: " + words);
  System.out.println("Num words: " + words.size());
  String secondWord = words.get(1);
  System.out.println("2nd word: " + secondWord);
  System.out.println("Contains 'hi'? " + words.contains("hi"));
  System.out.println("Contains 'bye'? " + words.contains("bye"));
  words.remove("hello");
  System.out.println("List: " + words);
  System.out.println("Num words: " + words.size());
}
```

More Basic Methods: Results

- **Output when using ArrayList**

```
List: [hi, hello, hola]
Num words: 3
2nd word: hello
Contains 'hi'? true
Contains 'bye'? false
List: [hi, hola]
Num words: 2
```

- **Output when using LinkedList (same!)**

```
List: [hi, hello, hola]
Num words: 3
2nd word: hello
Contains 'hi'? true
Contains 'bye'? false
List: [hi, hola]
Num words: 2
```

ArrayList and LinkedList: Advanced Methods

- **Sort the list**
  - sort(comparisonFunction)
    - Same comparison function (Comparator) as with Arrays.sort, covered later
- **Modify every list element**
  - replaceAll(mappingFunction)
    - Same mapping function as with Stream.map, covered later
- **Remove matching elements**
  - removeIf(matchingFunction)
    - Same matching function as with Stream.filter, covered later
- **Do operation on each element**
  - forEach(sideEffectFunction)
    - Same function as with Stream.forEach, covered later
- **Build a Blah array from a list of Blah**
  - toArray(new Blah[0])

These methods are summarized here, but discussed in detail, with examples, in later section on Java 8 updates to Lists and Maps.

Advanced Methods: Code

```
public static void main(String[] args) {
  List<String> words = new ArrayList<>(); // or,  = new LinkedList<>();
  words.add("hi");
  words.add("hello");
  words.add("hola");
  System.out.println("List: " + words);
  words.sort((word1,word2) -> word1.length()-word2.length());
  System.out.println("List: " + words);
  words.replaceAll(word -> word.toUpperCase() + "!");
  System.out.println("List: " + words);
  words.removeIf(word -> word.contains("E"));
  System.out.println("List: " + words);
  words.forEach(word -> System.out.println("Word: " + word));
  String[] wordArray = words.toArray(new String[0]);
  for(String word: wordArray) {
    System.out.println("Word: " + word);
  }
}
```

Preview of Lambda Expressions (Covered in Detail Later)

```
public static void main(String[] args) {
  List<String> words = new ArrayList<>(); // or, = new LinkedList<>();
  words.add("hi");
  words.add("hello");
  words.add("hola");
  System.out.println("List: " + words);
  words.sort((word1,word2) -> word1.length()-word2.length());
  System.out.println("List: " + words);
  words.replaceAll(word -> word.toUpperCase() + "!");
  System.out.println("List: " + words);
  words.removeIf(word -> word.contains("E"));
  System.out.println("List: " + words);
  words.forEach(word -> System.out.println("Word: " + word));
  String[] wordArray = words.toArray(new String[0]);
  for(String word: wordArray) {
    System.out.println("Word: " + word);
  }
}
```

Read this as "replace each element with a value that was the old element passed through the function that takes a word, turns it into upper case, and puts an exclamation point on the end".

Read this as "sort by using a comparison function that takes two arguments and returns the difference of their lengths".

Read this as "remove any element that passes the test function that takes a word and checks if it contains an E".

Read this as "pass each element to the function that takes a word and prints 'Word: ' concatenated with that word".

Advanced Methods: Results

- **Output when using ArrayList or LinkedList**

```
List: [hi, hello, hola]
List: [hi, hola, hello]
List: [HI!, HOLA!, HELLO!]
List: [HI!, HOLA!]
Word: HI!
Word: HOLA!
Word: HI!
Word: HOLA!
```

Overview

- **Idea**
  - Map, most commonly HashMap, provides simple lookup tables
- **Instantiating empty Map**

```
Map<KeyType,ValueType> map = new HashMap<>();
```

- **Associating values with keys**

```
map.put(someKey, someValue);
```

- **Retrieving values for potential key**

```
ValueType result = map.getOrDefault(potentialKey, defaultVal);
```

  - Returns value associated with key if there is such a value, otherwise returns the default value

```
ValueType result = map.get(potentialKey)
```

  - Returns associated value if there is one, otherwise returns null. Same as map.getOrDefault(potentialKey, null)

Map Basics

Abbreviated Example

```
Map<String,Employee> employees = new HashMap<>();
Employee e1 = new Person("a1234", "Larry", "Ellison");
// ...
// Build more Person objects
employees.put(p1.getEmployeeId(), e1);
// ...
// Associate more employees with their ids
Employee match1 = employees.get("a1234");
// match1 is same instance as p1 above
Employee match2 = employees.get("q1234");
// match2 is null, assuming q1234 is not
// the id of any employee in the Map
```

HashMap: Performance

- **Inserting a value associated with a key**
  - Independent of the number of entries in the table, i.e., O(1).
- **Finding the value associated with a key**
  - Independent of the number of entries in the table, i.e., O(1).
- **Finding the key associated with a value**
  - Requires you to look at every entry, i.e., O(N)
- **Assumptions**
  - This performance makes some assumptions about the hash function, which is usually a safe assumption
    - But Java has other Map types with different performance characteristics and features

Quick Aside: Static Blocks

- **Idea**
  - Code that runs when class is loaded, not when instance is built (as with constructor). Used to initialize static variables that require multiple steps to build their values.
- **Syntax**

```
public class Blah
  private static Map<KeyType,ValType> map = new HashMap<>();
  static {
    map.put(key1, value1);
    map.put(key2, value2)
  }
  ...
}
```

StateMap: Finding Abbreviations for State Name

```
public class StateMap {
  private static String[][] stateArray =
    { { "Alabama", "AL" },
      { "Alaska", "AK" },
      ...
    };

  private static Map<String,String> stateMap = new HashMap<>();

  static {
    for(String[] state: stateArray) {
      stateMap.put(state[0], state[1]);
    }
  }
```

StateMap (Continued)

```
  public static String abbreviation(String stateName) {
    return(stateMap.getOrDefault(stateName, "[None]"));
  }

  public static Map<String,String> getStateMap() {
    return(stateMap);
  }

  public static String[][] getStateArray() {
    return(stateArray);
  }
}
```

MapTest

```
public class MapTest {
  public static void main(String[] args) {
    String[] testNames = {
      "Maryland", "California", "New York",
      "Utopia", "Confusion", "Awareness"
    };
    for(String name: testNames) {
      System.out.printf("Abbreviation for '%s' is '%s'.%n",
                        name, StateMap.abbreviation(name));
    }
  }
}
```

```
Abbreviation for 'Maryland' is 'MD'.
Abbreviation for 'California' is 'CA'.
Abbreviation for 'New York' is 'NY'.
Abbreviation for 'Utopia' is '[None]'.
Abbreviation for 'Confusion' is '[None]'.
Abbreviation for 'Awareness' is '[None]'.
```

More Map Methods

A Few Other Simple Map Methods

- **keySet()**
  - Returns a Set of the keys
  - A Set is like a Map with boolean values (i.e., does key exist or not?)
- **values()**
  - Returns a Collection of the values
  - You can loop over a Collection just as with a List, but you cannot access entries by index
- **size()**
  - Returns the number of entries

A Few Advanced Map Methods

These methods are summarized here, but discussed in detail, with examples, in later section on Java 8 updates to Lists and Maps.

- **forEach(function)**
  - For each entry, passes key and value to the bi-function
    - map.forEach((k,v) -> System.out.printf("(%s,%s)%n", k, v);
- **computeIfAbsent(key, function)**
  - If there is no value associated with the key, pass the key to the function and store the result in the Map
- **merge(key, value, function)**
  - If there is no value associated with the key, inserts the value. Otherwise replaces existing value with the function applied to old and new values
    - map.merge(key, message, String::concat)
- **Notes**
  - Lambda functions (like the argument to forEach), method references (String::concat), and printf covered in later sections

Sets

Overview

- **Idea**
  - An unordered collection of values with no repeats
    - Similar to a Map where the values corresponding to each key are just boolean true
    - You just ask (true or false) if value exists
- **Instantiating empty Set**

```
Set<EntryType> set = new HashSet<>();
```

- **Adding entries to the Set**

```
set.add(someValue);
```

- **Testing if entry is in the Set**

```
if (set.contains(possibleValue)) { ... }
```

WebLanguages: Set of Programming Languages

```
public class WebLanguages {
  private static String[] languageArray =
    {"java", "javascript", "ruby", "c#", "python"};
  private static Set<String> webLanguages = new HashSet<>();

  static {
    for(String language: languageArray) {
      webLanguages.add(language);
    }
  }

  public static boolean isWebLanguage(String language) {
    return(webLanguages.contains(language.toLowerCase()));
  }
}
```

SetTest

```
public class SetTest {
  public static void main(String[] args) {
    String[] testNames = {
      "Java", "JavaScript", "C#",
      "COBOL", "FORTRAN", "Assembly"
    };
    for(String name: testNames) {
      System.out.printf("%s is a Web language? %s.%n",
                        name, WebLanguages.isWebLanguage(name));
    }
  }
}
```

```
Java is a Web language? true.
JavaScript is a Web language? true.
C# is a Web language? true.
COBOL is a Web language? false.
FORTRAN is a Web language? false.
Assembly is a Web language? false.
```

Summary

- **Lists**

```
List<EntryType> list = new ArrayList<>();
list.add(someEntry);
EntryType entry = list.get(someIndex);
System.out.println(list);
for(EntryType e: list) { doSomethingWith(e); }
```

- **Maps**

```
Map<KeyType,ValType> map = new HashMap<>();
map.put(someKey, someValue);
ValType result = map.getOrDefault(key, default);
```

- **Sets**

```
Set<EntryType> set = new HashSet<>();
set.add(someEntry);
if (set.contains(possibleEntry)) { ... }
```