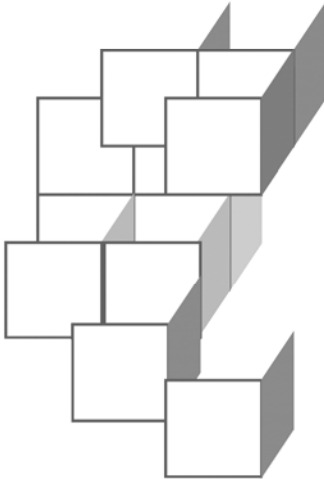
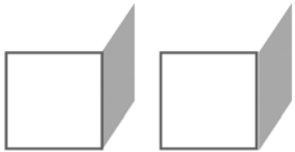




CHAPTER 10
COLLECTIONS



OBJECTIVES

After completing “Collections,” you will be able to:

- **Use the Collections API to manage dynamic collections of objects.**
 - Add values to a collection, including primitives.
 - Inspect values in a collection.
- **Choose between collection implementations for scalar collections and maps.**
- **Use iterators instead of directly manipulating collections, for better separation between interface and implementation.**
- **Take advantage of sorting and searching features in the API to optimize your own algorithms.**

Limitations of Arrays

- When managing series, sets, and groups of data, arrays are not always the best solution.
- Arrays capture snapshots of data fairly well.
- They do not excel where data is volatile – especially when the size of the data set can fluctuate.
 - To insert an element into an array is to slide every element above the insert point over some number of bytes in memory.
 - This assumes that the array has been allocated with extra space at the end; if it hasn't, then a new array must be allocated and all the values must be copied over to it.
- More generally, arrays expose low-level memory-management issues to the application programmer.
- When used as return values or parameters on public methods, arrays also expose something of a class' implementation strategy to the outside world.
- That is, if one wants to provide access to a private array, one has a few awkward choices:
 - Make the array itself available through an accessor method. The array could then be manipulated in ways the class designer wouldn't like.
 - Provide an interface for iteration over the array, with a **first** method, a **next** method, etc.
 - Return a deep copy of the array, which is inefficient.

The Collections API

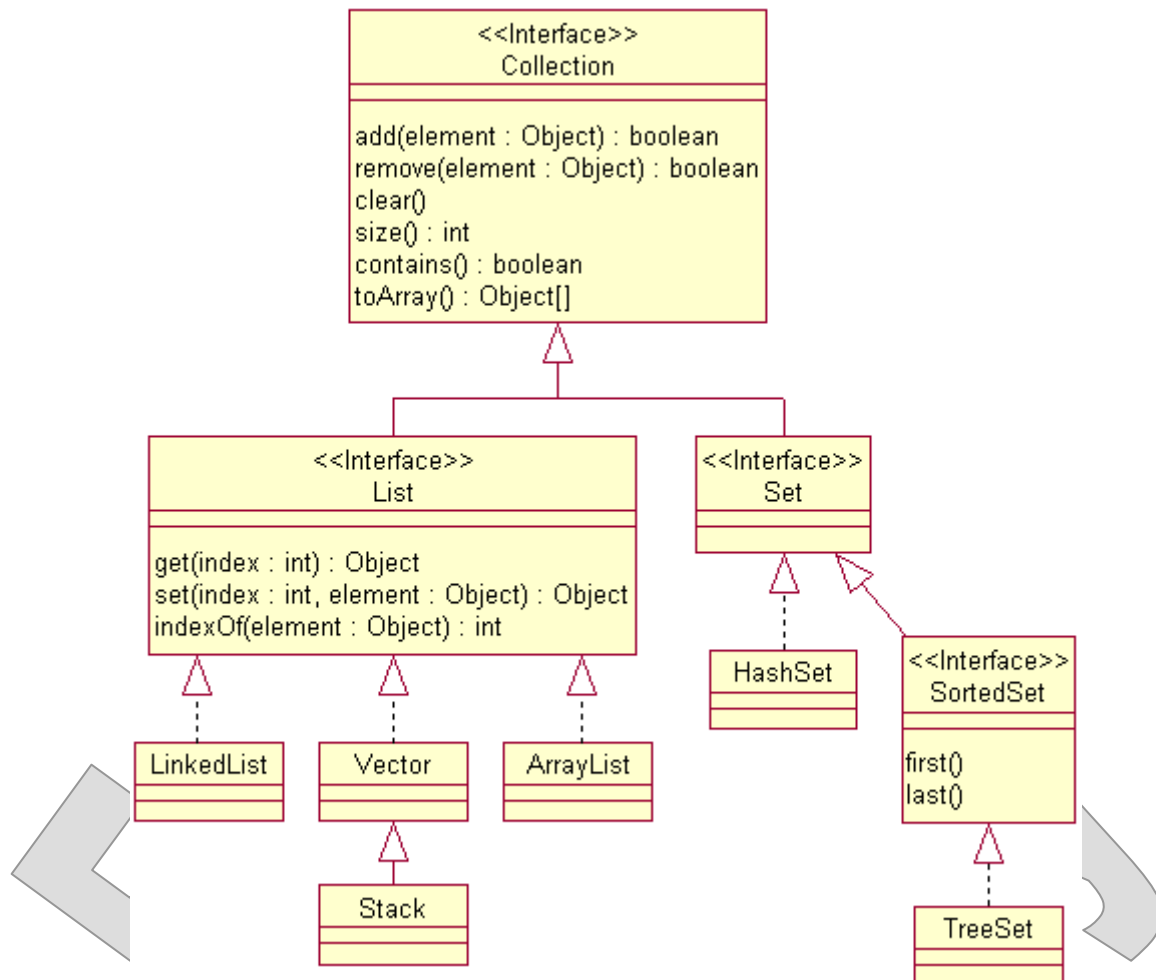
- The Java Core API includes a comprehensive set of dynamic collection classes called the **Collections API**.
 - The whole API is implemented in package **java.util**.
- The API includes:
 - Collections such as **Vector**, **LinkedList**, and **Stack**. Different implementations offer different performance characteristics.
 - **Maps** that index **values** under **keys**, including **HashMap**.
 - Variants of the above that assure **uniqueness** of elements, including **HashSet**.
 - Variants that assure that elements are always **ordered** by a comparator: **TreeSet** and **TreeMap**.
 - **Iterators** that abstract the ability to read and write the contents of a collection in loops, and isolate that ability from the underlying collection implementation.

Collections vs. Arrays

- Working with the Collections API is vastly different than managing arrays – both from a strategic perspective and in the mundane matter of syntax.
- Arrays are **strongly typed**.
 - You tell the compiler what type of element is expected, and it enforces that when you try to assign values to elements.
 - You can define arrays of primitive-type elements.
- Collections are **weakly typed**.
 - There is one **Vector** class, and all its elements are of type **Object**. That means any object of any type can live there, and that you must downcast elements when you read them.
 - You cannot include primitive values in collections, although there is a way to wrap or “box” them.
- Arrays are generally faster, because they represent direct, random access to blocks of memory.
- Collections are objects with methods that must be called to read or write elements.
- Collections offer some functional advantages:
 - They are much easier to use for general-purpose coding, and really shine when the data is highly volatile – lots of adds, deletes, and direct changes over time.
 - The iterator design split helps hide implementation choices.

Collection Types

- The collection types are classified under the root interface **Collection**.



- Ordered collections implement **List**.
- Collections that assure element uniqueness implement **Set**.
- Sorted collections implement **SortedSet**.
- Other collection types are possible but would have to be implemented outside the Core API.

The Collection Interface

- All collections implement the interface **Collection**:

```
interface Collection
{
// Partial listing of methods:
    public int size ();
    public void clear ();
    public Object[] toArray ();
    public boolean add (Object);
    public boolean remove (Object);
    public boolean addAll (Collection);
    public Iterator iterator ();
}
```

- Thus they can all perform certain functions:
 - Add and remove elements
 - Clear to an empty set
 - Report their size
 - Convert their data to an array of **Objects**
- Additional properties of a given collection are defined by its implementation of one of the sub-interfaces of **Collection**.
 - Ordered collections implement the **List** interface.
 - Collections that forbid duplicates – that is, assure element uniqueness – implement **Set**.
 - Collections that order, or sort, their elements implement **SortedSet**.

The Vector Class

- One of the more popular concrete types in the API is the **Vector**, which provides random access to a scalar list of elements.
 - **Vector** and **ArrayList** are the two classes whose semantics are most like a raw array.
- ```
for (int n = 0; n < vec.size (); ++n)
 System.out.println ((String) vec.elementAt (n));
```
- Elements are ordered based on how they were added to the collection; that is, there is no implicit sorting.
  - Elements needn't be unique within the collection.
- **Vectors perform best in “random access” to their elements, as they are backed by arrays.**
    - Their weakness, as with arrays, is insertion and deletion.
  - Create a **Vector** with no arguments and begin adding elements to it.
  - You can also specify the initial capacity of the **Vector**.
  - **Capacity and size are not the same thing!**
    - Size is the number of elements currently in the collection.
    - Capacity is the current allocation of “slots” for elements.
    - Capacity is always greater than or equal to size, and it grows and shrinks by intervals larger than one, to avoid constant re-allocation of memory.

## The `LinkedList` Class

---

- A radically different means of achieving a scalar collection is the **`LinkedList`**.
  - Each element in a linked list is discrete in memory.
  - It holds a pointer to the next element and the previous one.
- **Linked lists excel at insertion and deletion, because there is no need to shuffle any of the existing list elements when a new one is added.**
  - Rather, an existing link is broken and two new ones are formed.
  - Deletion is just the opposite process.
- **Iterating over a linked list is slower, however.**
  - The random seeking over a block of memory that the **`Vector`** offers is replaced by a walk from element one to element N.

## Building Collections

---

- **The first thing to remember about collections is that you must create them explicitly.**
  - This is true of arrays as well.
  - It is a common mistake to declare a reference to a **Vector** or **LinkedList** and just assume that the object is there and ready for action.
  - This doesn't work with any other classes, either!
- **Once a collection object is created, simply add elements to it.**
  - Use **add** to append the new element to the end.
  - Use insert methods to insert at a certain point in the collection. Note that **Collection** doesn't define an **insert**, but most subtypes offer their own semantics for this.
  - **remove** an element by identifying it. Many subtypes offer index-based remove methods as well.
- **Any Java object can be placed in any collection.**
  - Most collections will hold objects of the same type – these are **homogenous** collections.
  - **Heterogenous** collections are viable, though.
  - Often there's a bit of both: all elements will share a base type, but will vary over types derived from that base.

## Reading Elements

---

- Adding objects to a collection is a simple matter of calling **add**, or **insert**, and passing the object, as in:

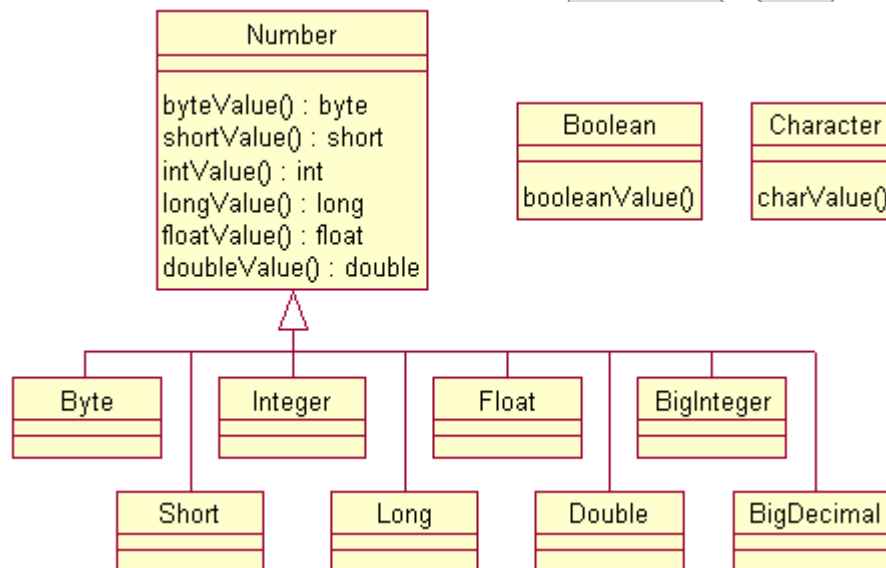
```
myVector.add (new MyClass (5));
myVector.add ("literal");
myVector.insertElement (second, 1);
```

- When reading objects from a collection, however, it's typically necessary to cast them back to their expected type.
  - All the methods of reading values from a collection will return a reference to **Object**.
  - This must be downcast to whatever type you're using.

```
MyClass first = (MyClass) myVector.elementAt (0);
Object second = myVector.elementAt (1);
String third = (String) myVector.elementAt (2);
```

## Collecting Primitive Values

- Primitive values cannot be placed directly in a collection.
- A library of classes is available in the **java.lang** package to wrap primitives:



- We’ve seen these in use for converting strings to numbers.
- They also serve the basic purpose of “boxing” primitive values: the primitive is held in an object of the corresponding wrapper type, and that can be placed in a collection.

```
vec.add (new Integer (5));
vec.add (new Double (45.5));
vec.add (new Boolean (false));
```

```
int y = ((Integer) vec.elementAt (0)).intValue ();
if (((Boolean) vec.elementAt (2)).booleanValue ())
 y += ((Number) vec.elementAt (1)).intValue ();
```

## Algorithmic Programming

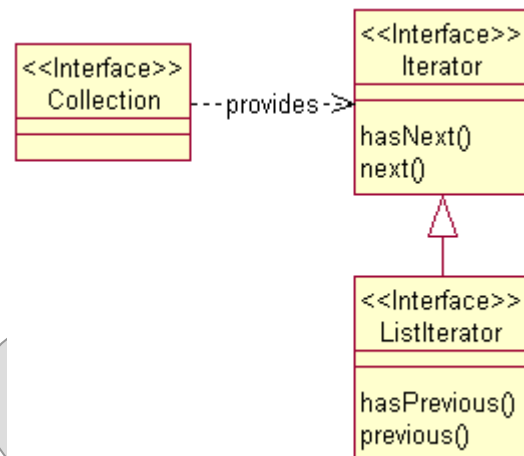
---

- Object-oriented theory has enabled software developers to decouple various concepts in solving complex problems.
- **Algorithmic programming** offers a different kind of decoupling.
- Certain algorithms occur frequently in a wide variety of programming solutions:
  - The **for-each** algorithm, which performs some function on every element in a collection – sometimes in a stateful manner such that an aggregate value or record is carried forward through the iteration
  - The **first-that** algorithm, which seeks from the beginning of a collection for the first item that meets some criteria
  - Various **sorting** algorithms
- The idea of algorithms is to parameterize the logic of iterating, searching, sorting, etc.
  - What do you want to do with (or to) each element?
  - What criteria do you want to seek out?
  - What criteria determine the appropriate sort order?
- More specific algorithms tend to crop up in application design, as well.

## The Iterator Interface

- There is a tension between the benefits of decoupling or parameterizing basic algorithms and the flexibility of different collection implementations with their own performance characteristics.
  - We want to be able to swap different collection types in and out, perhaps to tune performance after the application has been running for a while.
  - Algorithms isolate things like looping and sorting from application-specific behaviors.
  - We want them to be isolated from the collection implementation, as well.
- The **Iterator** interface isolates use of a collection from the collection class itself.

```
interface Iterator
{
 public void remove ();
 public boolean hasNext ();
 public Object next ();
}
```



- An iterator can be derived from a collection using the **iterator** method.
- Then, the common looping construct is:

```
while (iterator.hasNext ())
 doSomethingWith ((CorrectType) iterator.next ());
```

## Car Dealership

**EXAMPLE**

- A major revision of the car dealership has been broken out into a new tree of steps – see **Examples\CarsII\Step1**.
- There are no design changes in this revision.
- It's all implementation strategy: the arrays in **Dealership** have been replaced with **Vectors**.

```
Vector cars = new Vector ();
Vector usedCars = new Vector ();
Vector parts = new Vector ();
```

- **getAllCars** combines two of the vectors into a new one, and returns an iterator on that.

```
protected Iterator getAllCars ()
{
 Vector result = new Vector
 (cars.size () + usedCars.size ());
 result.addAll (cars);
 result.addAll (usedCars);
 return result.iterator ();
}
```

## Car Dealership

### EXAMPLE

- **listCars** runs one loop over this merged collection:

```
public String listCars ()
{
 StringBuffer result = new StringBuffer ();

 Iterator eachCar = getAllCars ();
 while (eachCar.hasNext ())
 {
 Car car = (Car) eachCar.next ();
 result.append (car.getVIN ().append (": ")
 .append (car.getShortName ().append
 (car instanceof UsedCar ? " -- USED" : ""))
 .append (endLine));
 }

 return result.toString ();
}
```

- The **findCar** methods use the same approach.

- The **DataManager** has been reworked to use the dynamic collections:

```
public static void prime (Dealership destination)
{
 destination.parts.add (new Part (...));
 destination.parts.add (new Part (...));
 ...
}
```

- Note that there is no longer a need to call out the size of the data set ahead of time – which was a slightly irritating extra point of maintenance:

```
Destination.parts = new Part [6];
```

## Better Management of Test Scores

**LAB 10A**

In this lab you will overhaul the **Scores** application (last seen in Lab 4C) to use the Collections API instead of various arrays. This will be a two-step process, as you will replace the primary collection of names, scores and grades in this lab, and then take a different approach to capturing occurrence statistics in Lab 10B.

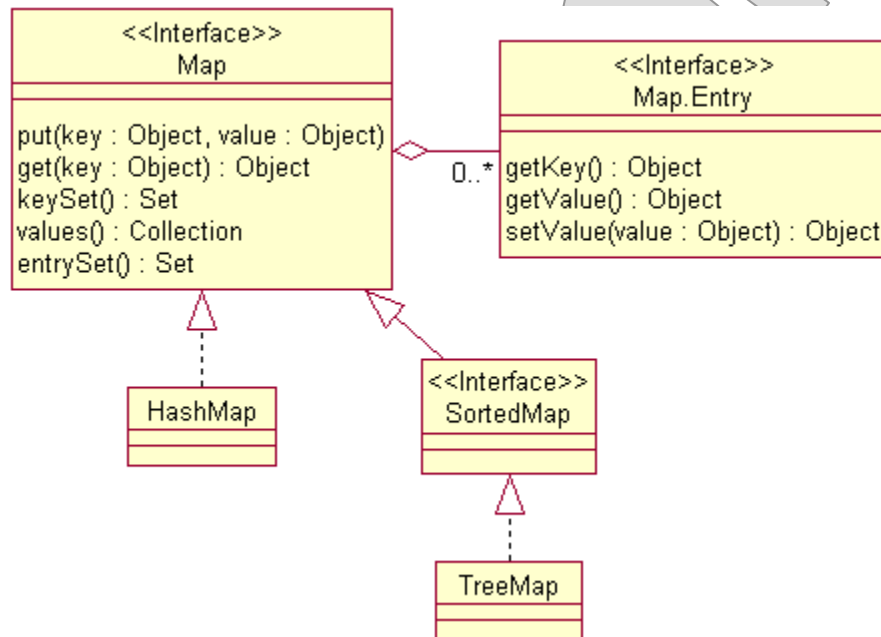
Detailed instructions are contained in the Lab 10A write-up at the end of the chapter.

Suggested time: 30-45 minutes.

Evaluation Only

## Maps

- Maps are not collections, per se.
- They are objects that implement the interface **Map** and thus map **key** objects to **value** objects.



- Keys must be unique.
- There can be only one value per key.

## The Map Interface

---

```
interface Map
{
 public int size ();
 public void clear ();
 public Object get (Object);
 public Object put (Object, Object);
 public Object remove (Object);
 public boolean containsKey (Object);
 public boolean containsValue (Object);
 public Collection values ();
 public Set entrySet ();
 public Set keySet ();
}
```

- Maps do provide **views** of their data as collections:
  - **keySet** provides just the keys.
  - **values** provides the values.
  - **entrySet** provides a set of entry objects, each of which is a pair of key and value.

## Post Offices

**EXAMPLE**

- The following code snippets exercise a **HashMap** object by storing the street addresses of US Postal Service offices in the Boston area, keyed by ZIP code:

```
Map map = new HashMap ();
map.put ("02130", "655 Centre Street");
map.put ("02131", "303 Washington Street");
map.put ("02118", "34 Stuart Street");
map.put ("02116", "92 Boylston Street");
map.put ("02107", "1 Franklin Street");

System.out.println (map.get ("02116"));

Iterator keys = map.keySet ().iterator ();
while (keys.hasNext ())
{
 String ZIP = (String) keys.next ();
 System.out.println (ZIP + ": " + map.get (ZIP));
}
```

## Sorted Collections

---

- Collections that implement **SortedSet** provide automatic insertion sorting whenever elements are added.
  - Thus the collection is always well ordered.
- Sorting is based on a definition of how two objects of a given type **compare**.
  - Strings are sorted alphabetically.
  - **Number** instances are compared numerically, including type conversions to compare numbers of different sizes.
- Most other classes require some sort of comparison criterion to be defined for them.
  - This can be done on the class itself by implementing **java.lang.Comparable**, which has one method **compareTo**.
  - Or a class can be responsible for making comparisons between objects of other classes, by implementing **Comparator**.
- The **SortedMap** interface defines similar semantics for maps.

## 50 States

**EXAMPLE**

- In **Examples\SortedSet**, the fifty state names primed into an array of strings are sorted in alphabetical order.
  - Recall that this was the subject of a challenge lab at the end of Chapter 4, and you were asked to implement various sorting algorithms by hand using only arrays.
  - This code uses the **TreeSet** class, which implements sorting with good performance over frequent inserts and deletions by storing elements in a binary tree.

```
String[] names = buildNamesArray ();
SortedSet set = new TreeSet ();
for (int n = 0; n < names.length; ++n)
 set.add (names[n]);
```

- It prints the sorted set using an iterator:

```
Iterator each = set.iterator ();
while (each.hasNext ())
 System.out.print (((String) each.next ()) + " ");
```

- Instead of a binary search algorithm using arrays, we simply call **contains** to search for a given string value:

```
String fullName = args[0];
for (int i = 1; i < args.length; ++i)
 fullName += " " + args[i];
```

```
System.out.println (fullName +
 (set.contains (fullName)
 ? " is a state."
 : " is not a state."));
```

## Gathering Statistics in a Map

**LAB 10B**

In this lab you will rebuild the latter part of the **Scores** application to get rid of the remaining two arrays, **possibleGrades** and **frequency**, and to replace them with a map.

Detailed instructions are contained in the Lab 10B write-up at the end of the chapter.

Suggested time: 30 minutes.

Evaluation Only

## JAVA **Generic Types**

### 1.5

---

- One of the most heralded language enhancements included in the Java 1.5 release is **generic types**.
  - A generic is what UML calls a **parameterized type**: a type defined partially in terms of another type.
  - Historically, C++ popularized the use of parameterized types using class and function **templates**, and it is the C++ template syntax that is borrowed by Java 1.5 for generics.
- Generics address a certain pattern of design challenges, in which the responsibilities of a class or method are necessarily parameterized.
  - Does this sound familiar? We were just discussing **algorithmic programming**, and it was a desire for generic algorithms that drove the creation of the C++ standard template library, or STL.
  - For example a collection class can give us an iteration behavior, but only with the compromise of type safety: we have to iterate over **Object** references.
  - A generic collection type can give us the best of both worlds: generalized collection behavior such as iteration, add/remove, access to size, etc.; but also specificity to a given type to be collected.

## Using Generics

JAVA  
1.5

- A generic type is defined in terms of some other type, which it collects or on which it acts in some way, using angled brackets.
- Not surprisingly, generics have the greatest impact on the Collections API, so we'll focus on that API to illustrate some specifics:
  - An **ArrayList<Point>** is an array-list of **Point** objects (from the package **java.awt**).

```
ArrayList<Point> myList = new ArrayList<Point> ();
```

- The compiler can now catch the following mistake:

```
myList.add (new Dimension (5, 4));
```

- A type-specific collection object will provide a type-specific iterator:

```
Iterator<Point> each = myList.iterator ();
```

- Also, it is no longer necessary to downcast the results of various accessor methods such as **List.elementAt** or **Iterator.next**, because their method signatures are all parameterized as well:

```
while (each.hasNext ())
 each.next ().x = 5; // Direct use of Point.x
```

## JAVA **Simplified Iteration**

### 1.5

---

- Java 1.5 brings another enhancement, this one specific to the Collections API: a simplified “for-each” loop.

```
for (Point each : myList)
 each.x = each.x + 1;
```

- This applies to the older weakly-typed collections (which are retained in 1.5) and generics as well.
- Combined with generics it makes for quite elegant code!
  - Compare the code above to the typical approach in 1.4:

```
Iterator each = myList.iterator ();
while (each.hasNext ())
{
 Point point = (Point) each.next ();
 point.x = point.x + 1;
}
```

## Auto-Boxing

JAVA  
1.5

- Continuing the Java-1.5 hit parade: we come to the **auto-boxing** feature, which essentially removes the 1.4 requirements to:

- Promote a primitive value to an object, as in:

```
myIntegers.add (new Integer (myInt));
```

- Demote the object back to a primitive, as in:

```
int getItBack =
 ((Integer) myIntegers.elementAt (0))
 .intValue ();
```

- Instead of all this laborious explicit type conversion, Java 1.5 allows implicit conversion.

- The Java-1.5 approach to the above would be:

```
myIntegers.add (myInt);
int getItBack = myIntegers.elementAt (0);
```

- This and many of the other features discussed in this chapter are generally termed “ease-of-use” features.
  - In fact that’s the thrust of one of the primary JSRs behind the 1.5 release.
  - This is one of the clearest examples of a stronger ease-of-use philosophy in Java, as there was no real architectural reason not to do this in the first place.

**JAVA**  
**1.5** **Scores 1.5****EXAMPLE**

- In **Examples\Scores1.5** the student-grading **Scores** application has been rewritten for Java 1.5, taking advantage of
  - Generic collection types
  - The for-each construct
  - Auto-boxing
  - Formatted printing to the console – a feature we'll introduce after looking at the first three
- See the beginning of the **main** method in **Scores.java**:

```
Vector<Record> scores = new Vector<Record> (10);
scores.addElement (new Record ("Suzie Q", 76));
...
System.out.println ("Student Score Grade");
System.out.println ("----- ----- -----");
for (Record record : scores)
{
 int score = record.score;
 String grade = null;
 if (score >= 60)
 grade = "" + (char) (68 - (score - 60) / 10);
 else
 grade = "F";

 record.grade = grade;
 ...
}
```

## Scores 1.5

**EXAMPLE**JAVA  
1.5

- Note that a type can be multiply parameterized, as in map of strings keys and integer values:

```
Map<String,Integer> grades =
 new TreeMap<String,Integer> ();
int total = 0;
Student: for (Record record : scores)
{
 Integer frequency = grades.get (record.grade);
 if (frequency != null)
 grades.put (record.grade, frequency + 1);
 else
 grades.put (record.grade, 1);
 total += record.score;
}
```

- Notice too that we choose to use the **Integer** type in this case so that we can test for nullness and then either put in a new value or increment what was there.
- We can mix that usage in the call to **get** (which is, however, strongly-typed as an **Integer**, thanks to the type parameters) with the auto-boxing approach in our **put** calls.

## JAVA **Variable Argument Lists**

### 1.5

---

- Java 1.5 also brings out the ability to define a method as taking a variable number of parameters.
  - Define a method whose last (or only) parameter type is followed immediately by an ellipsis of three dots, as in:

```
public void concatenate (String... components);
```

- Call the method normally, with one or more arguments of the appropriate type:

```
String hi = concatenate ("Hello", " ", " ", "Java!");
```

- **This is a great convenience when operating on an unknown number of arguments.**
  - The traditional solution has been to accept an array of objects.
  - In fact, this is still happening, under the covers, but now the Java compiler is building the array for the invocation.
  - The variable list of arguments is still treated as an array by the declaring method, in its own code.

## String and Stream Formatting

JAVA  
1.5

- Java's ability to format output has grown gradually from rather humble beginnings.
  - Originally, C-style formatted output (along the lines of **printf** and related functions in the standard library) was considered old-school and not object-oriented enough for Java.
  - Object-oriented solutions came along in later releases, primarily in the **java.text** package, which provides utilities such as **DecimalFormat** and **DateFormat**.
- Convenient formatted output in the C tradition is finally available in Java 1.5.
- The center of this implementation is the new **Formatter** class in **java.util**.
  - Its **format** method takes a formatting string and a variable list of arguments of **Object** type.
  - It will replace **fields** found in the formatting string with values in the argument list, and then convert and format them according to the field directives.

```
Formatter.format ("%3d squared is %6d.", 5, 25)
```

produces

```
5 squared is 25.
```

- Convenience methods with similar signatures are now found on the **System.out** and **System.err** streams, and on the **String** class itself.

## SUMMARY

- **Dynamic collections can make many coding tasks much easier, and the resulting code more maintainable.**
- **Working with the Collections API gives this benefit, but also means tracking type information yourself, and doing a lot of downcasting, because the collections are all weakly typed.**
- **Different collection types offer different performance characteristics, and each excels at a certain pattern of usage.**
- **Iterators offer a layer of objects that isolate the semantics of traversing collections from the underlying implementation.**
- **Maps provide an easy way to build collections of name/value pairs; these are also weakly typed.**
- **Sorted sets and maps assure that all values or keys are kept in their natural ascending order.**



4. Change the rest of the loop to get and set values on **record**:

```

for (int student = 0; student < scores.size (); ++student)
{
 Record record = (Record) scores.elementAt (student);
 int score = record.score;
 String grade = null;
 if (score >= 60)
 grade = "" + (char) (68 - (score - 60) / 10);
 else
 grade = "F";

 record.grade = grade;
 System.out.print (record.name);
 for (int fill = record.name.length (); fill < 24; ++fill)
 System.out.print (" ");
 System.out.println (" " + score + " " + grade);
}

```

5. Skip down to the nested loop that populates the **frequency** array. Make the same change to the outer loop (the one labeled **Student**;) as you did at the top of the method.
6. In testing for a matching grade inside the inner loop, change **grades[student]** to the grade that's been stored for the record at index **student** – take a moment to be sure you understand the usage here, including the typecast and the read of field **grade**:

```

if (((Record) scores.elementAt (student)).grade
 .equals (possibleGrades[g]))

```

7. Make a similar change to the code that accumulates the **score** in **total**.

8. Build and test at this point. You should get the same behavior as in the completion of Lab 4C:

**build**

**run**

| Student         | Score | Grade |
|-----------------|-------|-------|
| Suzie Q         | 76    | C     |
| Peggy Fosnacht  | 91    | A     |
| Boy George      | 80    | B     |
| Flea            | 55    | F     |
| Captain Hook    | 71    | C     |
| Nelson Mandela  | 98    | A     |
| The Mighty Thor | 70    | C     |
| Oedipa Maas     | 88    | B     |
| Uncle Sam       | 69    | D     |
| The Tick        | 60    | D     |

Statistics:

There were 2 As given.  
 There were 2 Bs given.  
 There were 3 Cs given.  
 There were 2 Ds given.  
 There were 1 Fs given.

The mean score was 75.8

### Optional Steps:

9. After computing the grades, but before generating the statistics, add a middle section that performs an insertion sort on the **scores**, placing them in descending order in a new **Vector** called **sorted**.
10. To do this, build a nested loop: the outer loop (label it **Source**;) will iterate over **scores**, as usual, and the inner loop will iterate over **sorted**.

```
Source: for (int student = 0; student < scores.size (); ++student)
{
 Destination: for (int s = 0; s < sorted.size (); ++s)
 {
 }
}
```

11. At the top of the outer loop (that is, right before the inner loop) get a reference **record** to the **Record** object at that point in **scores**.
12. Do the same thing at the top of the inner loop – call the reference **test** and get it from **sorted**.
13. If the **record.score** is greater than the **test.score**, call **sorted.insertElementAt**, passing **record** and the index **s**, and then **continue Source**;
14. If the inner loop falls through, then **record** belongs at the end of the growing **sorted** vector; pass it in a call to **sorted.addElement**.
15. After the nested loops, print a new header and iterate over **sorted**, printing a formatted line for each record, just as you do in the original code.

16. Build and test, and your complete output should be:

**build**

**run**

| Student         | Score | Grade |
|-----------------|-------|-------|
| Suzie Q         | 76    | C     |
| Peggy Fosnacht  | 91    | A     |
| Boy George      | 80    | B     |
| Flea            | 55    | F     |
| Captain Hook    | 71    | C     |
| Nelson Mandela  | 98    | A     |
| The Mighty Thor | 70    | C     |
| Oedipa Maas     | 88    | B     |
| Uncle Sam       | 69    | D     |
| The Tick        | 60    | D     |

| Student         | Score | Grade |
|-----------------|-------|-------|
| Nelson Mandela  | 98    | A     |
| Peggy Fosnacht  | 91    | A     |
| Oedipa Maas     | 88    | B     |
| Boy George      | 80    | B     |
| Suzie Q         | 76    | C     |
| Captain Hook    | 71    | C     |
| The Mighty Thor | 70    | C     |
| Uncle Sam       | 69    | D     |
| The Tick        | 60    | D     |
| Flea            | 55    | F     |

Statistics:

There were 2 As given.

There were 2 Bs given.

There were 3 Cs given.

There were 2 Ds given.

There were 1 Fs given.

The mean score was 75.8

# Gathering Statistics in a Map

**LAB 10B**

## Introduction

In this lab you will rebuild the latter part of the **Scores** application to get rid of the remaining two arrays, **possibleGrades** and **frequency**, and to replace them with a map.

**Suggested Time:** 30 minutes

**Root Directory:** Capstone\Java

**Directories:**     **Labs\Lab10B**                     (do your work here)  
                       **Examples\Scores\Step6**         (backup copy of starter files)  
                       **Examples\Scores\Step7**         (contains lab solution)

**Files:**             **Scores.java**

## Instructions

1. Open **Scores.java** and start by removing the two arrays **possibleGrades** and **frequency**, near the bottom of the **main** method. (Note one of the advantages of this mapping approach is that you will not need to define a list of possible grades ahead of time; the map will begin to accumulate whatever grades do occur.)
2. In their place, create a new **HashMap** and call it **grades**.
3. Rip out the inner loop labeled **Grade:**, except for the last line that accumulates values in **total**.
4. Begin the new code for the **Student:** loop by getting a reference **record** to the **scores** element at index **student**.
5. Create an **Integer** reference called **frequency** and initialize it by looking up any frequency entry that may exist under the current record's grade:

```
Integer frequency = (Integer) grades.get (record.grade);
```

6. Either there will be something there already for this grade, or there won't. If **frequency** is **null**, then this is the first encounter with this grade, so call **grades.put**, passing **record.grade** and a new **Integer** built with the value 1.
7. Otherwise, call **grades.put** and pass **record.grade** and a new **Integer** whose value is **frequency.intValue () + 1**. Note that this will overwrite the value you just looked up, which means you've incremented the existing count.
8. After the code that prints the "Statistics:" header to the console, replace the loop over **possibleGrades** with an iteration over **grades.keySet**.

9. For each key, get the key value out of the iterator and print a line of text formatted to show the grade and the number of times it occurred, which will be `grades.get(key)`. Notice that you don't need to downcast the returned object to an `Integer` in this case, because all that will be done with it is string conversion, and that can be called on any `Object`.
10. Build and test, and you should see that you're gathering statistics accurately. However, you've lost the A-through-F ordering that was in place before thanks to the manual ordering of `possibleGrades`.

```
Statistics:
There were 2 Ds given.
There were 2 As given.
There were 1 Fs given.
There were 3 Cs given.
There were 2 Bs given.
```

11. How can you get sorting into your code? This couldn't be much simpler: change `HashMap` to `TreeMap`, build, and test again. This map type automatically sorts as keys are added, so:

```
Statistics:
There were 2 As given.
There were 2 Bs given.
There were 3 Cs given.
There were 2 Ds given.
There were 1 Fs given.
```