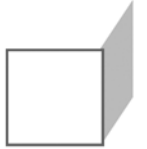
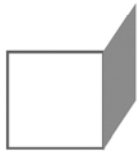
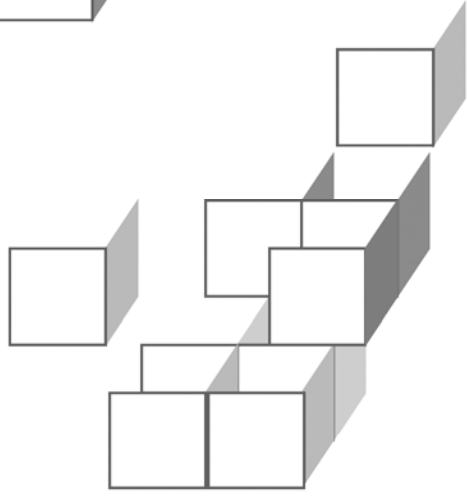
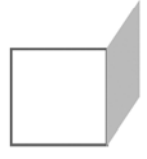
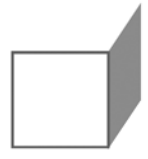




**CHAPTER 7**  
**GENERIC, COLLECTIONS, AND**  
**ALGORITHMS**



## OBJECTIVES

*After completing “Generics, Collections, and Algorithms,” you will be able to:*

- Explain the advantages of **generics** in the Java language, and use generic types in your code.
- Use the Collections API to manage dynamic collections of objects.
  - Add values to a collection, including primitives.
  - Inspect values in a collection.
- Choose between Core API implementations for lists, sets, 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.
- **To wit: 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.

## Dynamic Collections

---

- A **dynamic collection** is a class which can hold zero, one, or many instances of some other class(es).
- By contrast to arrays, dynamic collections:
  - **Hide the data** storage away from the caller, preserving the OO principle of (you guessed it) data hiding
  - Use **more memory and processing**, because they are instances of classes, with all the usual overhead
  - **Handle insertion, deletion, and resizing better** – even though many implementations will be backed by arrays at some level
  - Can themselves be hidden behind **iterators**, which fosters general-purpose **algorithmic programming**
  - Can offer rich **object-oriented interfaces** to their underlying data, and thus can be easier for client code to use
  - Can take advantage of OO **polymorphism**, such that a client knows only the interface and the implementation can be tuned for certain performance advantages
- **Java has always had dynamic collections.**
- **Through Java 1.4, they were only available in a weakly-typed form.**
  - That is, the collection classes – such as **Vector**, **LinkedList**, **HashMap** – managed **Objects**, and not any specific type.
  - This was a weakness compared to strongly-typed arrays.

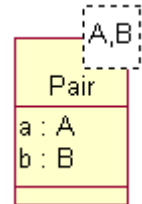
## Generic Types

---

- As of Java 5 the language supports **generic types**.
  - A generic is what UML calls a **parameterized type**: a type defined partially in terms of another type or types.
  - C++ popularized the use of parameterized types, and it is the **C++ template syntax** that generics borrow.
  - C++ coders, beware! These are not templates, but a more limited feature. More on this at the end of the chapter.
- Generics address a certain pattern of design challenges, in which the responsibilities of a class or method are necessarily parameterized.
  - Does this sound familiar? Dynamic collections in Java underwent a revolution between Java 1.4 and 5.0, and all are now generic types.
  - That is, we speak not just of a “list,” but of “a list of strings;” not just a “map,” but “a map from integers to instances of **MyClass**.”

```
List<String>
Map<Integer, MyClass>
```

- For collections, this removes one of the major disadvantages as compared to using arrays.
  - The other, loss of efficiency, is a completely natural tradeoff; ultimately both arrays and dynamic collections have their places.



## Declaring Generic Classes

---

- It will be beyond our scope in this course to get into creating our own generic types.
- Still, it's worth a little while to understand the basic syntax on the declaring side, and then we'll focus on how to use generics as defined in the Core API.
- The compiler recognizes a class as a generic type by the presence of an angle-bracketed list of predicate types following the class name.
  - There must be at least one such type; if more, separate them with commas.

```
public class Things<T>
public class OtherClass<A,B,X>
```

- The predicate types are understood to be placeholders for other classes – not primitives or arrays of things.
  - The class code then uses the defined placeholder (such as **T**) in its code, and the compiler treats references to **T** simply as references to some object.
  - Only when the generic class is used in client code is **T** replaced with a “real” type such as **String** or **Car**.
- It is also possible to define a generic that can only work on subtypes of a certain other class.
  - The predicate is declared to **extend** that other class:

```
public class PostalDelivery<T extends USAddress>
```

## Using Generics

---

- When a **generic** is put to a **specific** use, the compiler can replace the predicate types throughout, and apply its usual strict type checking to everything.

```
Things<String> stringThings = new Things<String>();
```

- Say **Things<T>** defines a method **addThing** that takes a parameter of type **T**; the compiler can assure that it is called safely, and so would allow this:

```
stringThings.addThing ("Hello");
```

- ... but flunk this:

```
stringThings.addThing (new YourClass ());
```

- It works the same way with return types, but now the advantage is less in avoiding silly mistakes and more in the convenience of not having to downcast a return type.

- If **Things<T>** also offers a method **getBestThing** that returns an instance of **T**, there is no need to downcast it to **String** when working with **Things<String>**.

```
String best = stringThings.getBestThing ();
```

- Nonetheless, **Things<T>** could be used in different code to operate on instances of **Car**, **Ellipsoid**, or any other class.

## Implicit Type Arguments

---

- As of Java 7, the compiler will try to deduce your type arguments when constructing new objects.
- This makes it possible to leave out the arguments in certain common constructs, where they're obvious:

```
List<String> myList = new ArrayList<> ();  
Pair<Integer,String> myPair = new Pair<> (5, "Hi");
```

- Note that it is not the constructor arguments that determine the type arguments – though the compiler could probably be taught to use those, too!
- Rather it is the context in which the expression is being evaluated.
  - That is, the compiler is working “**outside-in**” in a way that it did not do in the earliest versions of Java.
  - Another example of outside-in evaluation is the auto-boxing feature we'll discuss later in this chapter.

## A Generic Pair of Objects

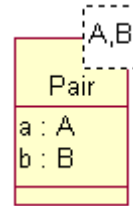
**EXAMPLE**

- Though the Collections API provides much of the motivation for generic types, we'll consider some abstract generic examples on their own, before concentrating on collections for most of the chapter.
- A very simple example – in fact a primitive sort of collection class itself – is in **Examples/Generics**.
  - See the source file `src/cc/generics/Pair.java`:

```
public class Pair<A,B>
{
    public A a;
    public B b;

    protected Pair () {}

    public Pair (A a, B b)
    {
        this.a = a;
        this.b = b;
    }
}
```



- This is just a class that holds a reference to two other objects, **a** and **b**.
- But the types **A** and **B** are not known when compiling this class – there is no implication that actual classes **A** and **B** exist when `Pair<A,B>` is compiled.
- These placeholders are only replaced with specific classes when other code uses the `Pair<A,B>` class.

## A Generic Pair of Objects

**EXAMPLE**

- The class **PairOfPairs** exercises this generic type:

```
public class PairOfPairs
{
    public static void main (String[] args)
    {
        Pair<String,Integer> score1 =
            new Pair<String,Integer>
                ("Will", new Integer (60));

        // This would not compile:
        //score1.a = new Integer (60);
        //score1.b = "Will";

        Pair<String,String> sPair =
            new Pair<String,String> ("This", "That");
        sPair.b = "Will";
    }
}
```

- **score1** is a pair of string and integer.
  - The compiler will enforce this fact, replacing **A** in the generic definition of the class with **String**, and **B** with **Integer**.
  - Thus the user of the class is protected from the understandable mistake of swapping A and B when reading or writing values – an **Object**-based **Pair** would not do that.
- **sPair** is a pair of strings.
  - Where it's concerned, **b** is a **String**!
  - The two invocations of **Pair<A,B>** are as different to the compiler as **Car** and **Part**.

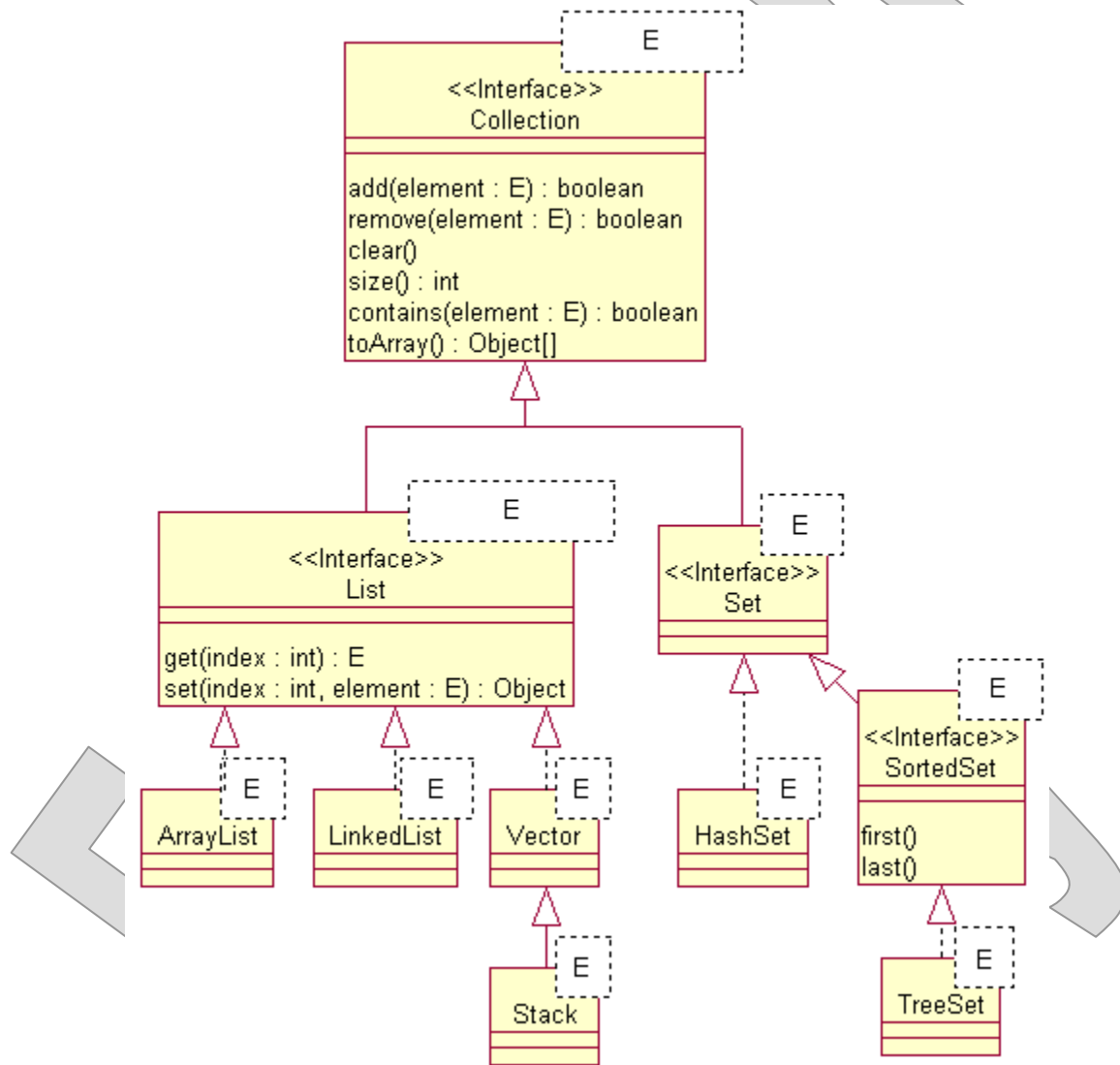
## The Collections API

---

- 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:
  - **Ordered collections** such as **ArrayList**, **LinkedList**, and **Stack**. Different implementations offer different interfaces and 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 **sorted** 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.
  - Several other specializations: weak-reference collections, types meant for working with enumerated values, and so on.
- There is also a class utility called **Collections**, which offers a very useful set of algorithms.

## Collection Types

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



- Ordered collections implement **List<E>**.
- Collections that assure uniqueness implement **Set<E>**.
- Sorted collections implement **SortedSet<E>**.

## The Collection<E> Interface

---

- All collections implement the interface **Collection<E>**:

```
public interface Collection<E>
{
// Partial listing of methods:
    public int size ();
    public void clear ();
    public Object[] toArray ();
    public boolean add (E element);
    public boolean remove (E element);
    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<E>**.

## The List<E> Interface

---

- A specialization of **Collection<E>**, **List<E>** promises that the collection is ordered, and offers additional methods based on an ordinal index:

```
public interface List<E>
    extends Collection<E>
{
    public void add (int index, E element);
    public E remove (int index);
    public E get (int index);
    public void set (int index, E element);
    public ListIterator listIterator ();
}
```

- **Lists are the most commonly used collection types.**
  - They are also the most like arrays, which are also ordered and can be indexed, but do not offer any of the other, extended features we've discussed, such as uniqueness or sorting.
- **Though the interface offers essentially random access to data, not every implementation will be efficient in that usage pattern.**
  - Traditionally, though, it's the job of the implementer to anticipate what client code will do, and to provide the appropriate collection type for the anticipated usage.
  - Failing that, it is use of interfaces such as **List<E>** that allow the implementer the flexibility to swap in different collection types as actual usage patterns and performance are observed.

## The ArrayList<E> Class

---

- Maybe the most popular concrete type in the API is the **ArrayList<E>**, which, as the name suggests, implements the **List<E>** over an array of **E**.
- **ArrayLists** perform best in “random access” to their elements, because the underlying arrays are good at that, too.
- Their weakness, as with arrays, is insertion and deletion.
  - However they are smart about allocating in blocks, so that only occasional resizing of the backing array is required.
- Create an **ArrayList** with no arguments and begin adding elements to it.
- You can also specify the initial capacity as an integer.

```
List<String> myList = new ArrayList<String> ();  
List<Car> cars = new ArrayList<Car> (6);
```

- **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 typically larger than one – ten is the default.

## The **LinkedList<E>** Class

---

- A radically different means of achieving a scalar collection is the **LinkedList<E>**.
  - 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 a little slower, however, and random access to an arbitrary index is **LinkedList**'s real weak spot.**
  - Indexing right into the backing array used by **ArrayList** 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 an **ArrayList** or **LinkedList** and just assume that the object is there and ready for action.
  - This doesn't work with any other classes, either! but for some reason with collections it's a much more common mistake.
- **Once a collection object is created, simply add elements to it.**
  - Use **add** to append the new element to the end.
  - On a list, use the **add** overload that takes an index to insert at a certain point in the collection.
  - **remove** an element by identifying it, or on a list by its index.
- **Any Java object of the specific type that replaces E – or of a subtype of E – can be placed in any collection.**
  - Most collections will hold objects of the same type – these are **homogenous** collections.
  - **Heterogeneous** 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** and passing the object, as in:

```
List<MyClass> myList = new LinkedList<MyClass> ();  
myList.add (new MyClass (5));
```

```
List<String> stringList = new ArrayList<> (); //!!!  
stringList.add ("literal");  
stringList.add ("figurative");
```

```
List<Car> cars = new ArrayList<> (6); //!!! again  
cars.add (someCar);  
cars.add (someOtherCar);  
cars.add (1, someUsedCar);
```

- Notice the Java-7 usages above; the compiler can figure the expected type argument from the expression context.

- Read objects from a list using **get**.

```
String whichMeaning = stringList.get (1);
```

- **get** will return an object of type **E** – or a subtype.
- You may choose to downcast this reference to a subtype of **E** – with the same caveats as with any object reference.

```
Car someCar = cars.get (1);  
if (someCar instanceof UsedCar)  
{  
    UsedCar used = (UsedCar) someCar;  
    ...  
}
```

- Through the **Collection<E>** interface, there is no such simple means of getting individual elements.

## Looping Over Collections

---

- A more common problem is to process every element in a collection, rather than drawing one arbitrary element.
- For a list, anyway, you could do it like this:

```
for (int i = 0; i < myList.size (); ++i)
    doSomethingWith (myList.get (i));
```

- Does this look familiar?
- This is very much like the “poor man’s” loop over arrays.

- For arrays there’s a simplified **for loop**; would the same technique work for collections?

- Oh, yes!

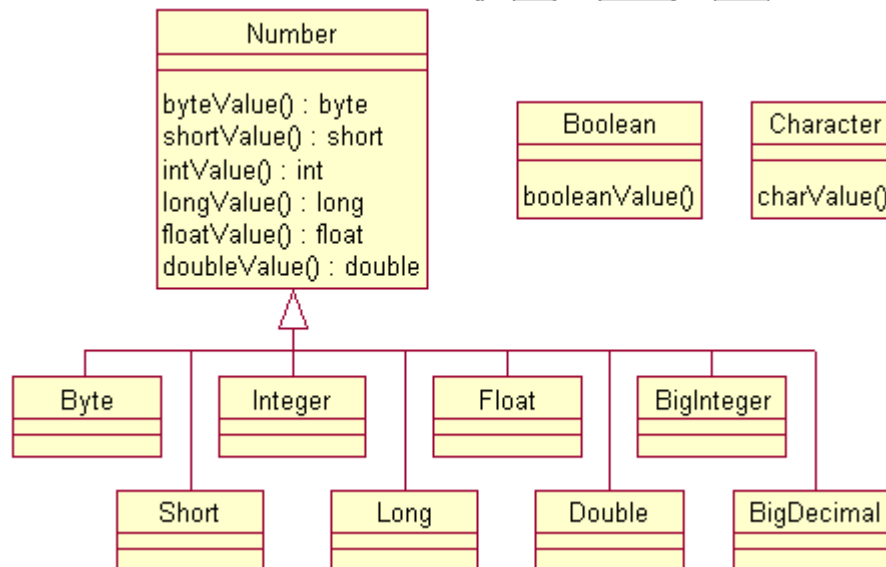
```
for (String whichMeaning : stringList)
    interpretAs (whichMeaning);
```

```
for (Car car : cars)
    if (car.getVIN ().equals (VIN))
        return car;
```

- The exact requirement is that the collection implement the interface **Iterable<E>**, which most collection types in the API do.

## Collecting Primitive Values

- Primitive values cannot be placed directly in a collection.
  - Generics only work for classes; `List<int>` is impossible.
- 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.

```
List<Integer> intList = new ArrayList<Integer> ();
```

```
intList.add (new Integer (5));
intList.add (new Integer (4));
```

```
int y = intList.get (0).intValue ()
        + intList.get (1).intValue ();
```

## Auto-Boxing

---

- Also as of Java 5.0, most boxing and un-boxing conversions will occur implicitly:

- Promotion of a primitive to the corresponding wrapper object
- Demotion of a wrapper back to the corresponding primitive

- For example, the code on the previous page can be simplified:

```
List<Integer> intList = new ArrayList<Integer> ();  
  
intList.add (5);  
intList.add (4);  
  
int y = intList.get (0) + intList.get (1);
```

- Note that you still need to cast generic **Number** objects to the specific types, such as **Integer** and **Double**, before auto-boxing will “kick in.”

- Working with a heterogeneous collection such as **ArrayList<Number>**, explicit casts are necessary:

```
List<Number> numList = new ArrayList<Number> ();  
  
numList.add (5); // fine, clearly an int literal  
numList.add (4.5); // also fine, a double  
  
int y = ((Integer) numList).get (0);  
    // cast is necessary so runtime can choose  
    // the correct primitive type for conversion
```

## xxxAll Methods

---

- There are a number of convenience methods in various collections that will, say, add or remove all elements in another collection.

- For these to work on a generic type such as `List<E>`, they need to take advantage of another, stranger generics syntax:

```
public void addAll (Collection<? extends E>);  
public void removeAll (Collection<? extends E>);  
public boolean containsAll  
    (Collection<? extends E>);
```

- The question mark is called a **wildcard**.
  - Wildcards allow limited type conversion, not between the types of elements in collections, but between the collections themselves.
  - A method like `addAll` wants to accept as an argument any collection of any type convertible to `E`.
  - Then why not a parameter type `Collection<T extends E>`?

## Convertibility of Generics

---

- This gets us into a fairly mind-bending exercise in how generics work (and don't work).
- The root issue can be expressed in a trick question: for some generic type **X** and two types **Base** and **Derived** that share an inheritance relationship: is **X<Derived>** convertible to **X<Base>**?

- Instinct immediately says, “yes, of course it is!”
- But in fact it is not – not safely.
- Consider what might happen:

```
List<Derived> dList = new LinkedList<Derived> ();  
List<Base> bList = dList; // seems safe  
bList.add (new Base ()); // hmm ...  
dList.get (0).someDerivedMethod (); // OOPS!
```

- See the problem?
- Once the compiler allows the second line to pass, it can't prevent the fourth line from blowing up at runtime.
- So the compiler makes the second line impossible, because, at bottom, it is not safe to treat a collection of **Derived** as a collection of **Base**.

## Wildcards

---

- It would be safe, though, if we could guarantee that the base-type reference **bList** would never modify the collection.
- And we have designs that call for methods like **addAll** and **removeAll**, which want to accept collections of their own type **E** or some derived type.

- How can they be declared?

```
public void addAll (Collection<E>)
```

- ... won't accept a collection of E-derived type.

```
public void addAll (Collection<T extends E>)
```

- ... won't pass, for exactly the reason on the previous page.

- This is the motivation for the wildcard syntax.

```
public void addAll (Collection<? extends E>)
```

- This syntax strikes a deal with the compiler:
  - The compiler agrees to accept collections of **E** or collections of E-derived types – even though it makes the poor boy nervous.
  - In return, the programmer agrees not to modify the collection it finds through the object reference.
  - A method such as **addAll** only needs to read elements from the source collection, and there's no difficulty there.

## 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 **ArrayLists**.

```
List<Car> cars = new ArrayList<Car> ();  
List<UsedCar> usedCars = new ArrayList<UsedCar> ();  
List<Part> parts = new ArrayList<Part> ();
```

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

```
protected List<Car> getAllCars ()  
{  
    ArrayList<Car> result = new ArrayList<Car>  
        (cars.size () + usedCars.size ());  
    result.addAll (cars);  
    result.addAll (usedCars);  
    return Collections.unmodifiableList (result);  
}
```

- The last line converts the temporary collection into an unmodifiable collection, using a utility method we'll study a little later in the chapter.

## Car Dealership

**EXAMPLE**

- `findCar` runs one loop over this merged collection:

```
public Car findCar (String VIN)
{
    for (Car car : getAllCars ())
        if (car.getVIN ().equals (VIN))
            return car;

    return null;
}
```

- 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 7A**

In this lab you will overhaul the **Scores** application 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 7B.

Detailed instructions are contained in the Lab 7A write-up at the end of the chapter.

Suggested time: 30-45 minutes.

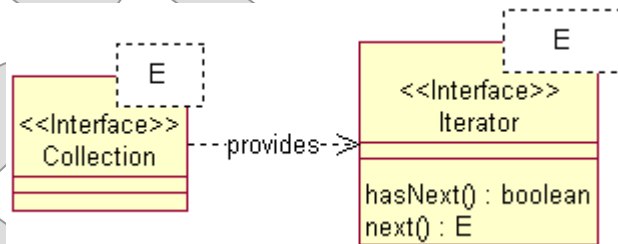
Evaluation Only

## The Iterator<E> Interface

- The **Iterator<E>** interface isolates iteration over a collection from the collection class itself.

```
interface Iterator<E>
{
    public void remove ();
    public boolean hasNext ();
    public E next ();
}
```

- An iterator can be derived from a collection using the **iterator** method.



- Then, the common looping construct is:

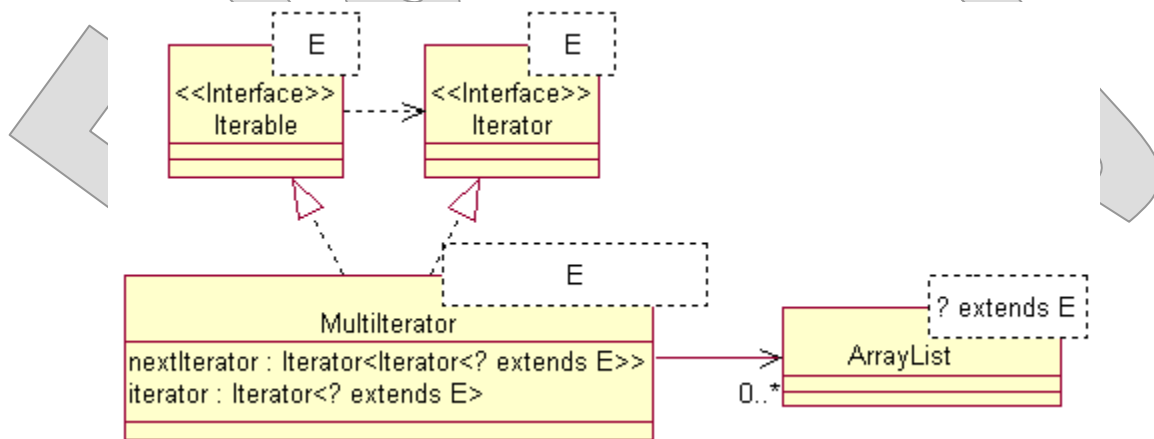
```
while (iterator.hasNext ())
    doSomethingWith (iterator.next ());
```

- Iterators offer a clean way to expose collections to the outside world without being tied to the collection type.
- An application is free to implement its own iterator class, which might do any sort of gymnastics to present a smooth stream of objects to the caller.
  - There might be multiple collections behind a single iterator.
  - Some of the data may not even be known when the iterator is requested – the iterator might use lazy evaluation, caching, etc.

## Custom Cars Iterator

**EXAMPLE**

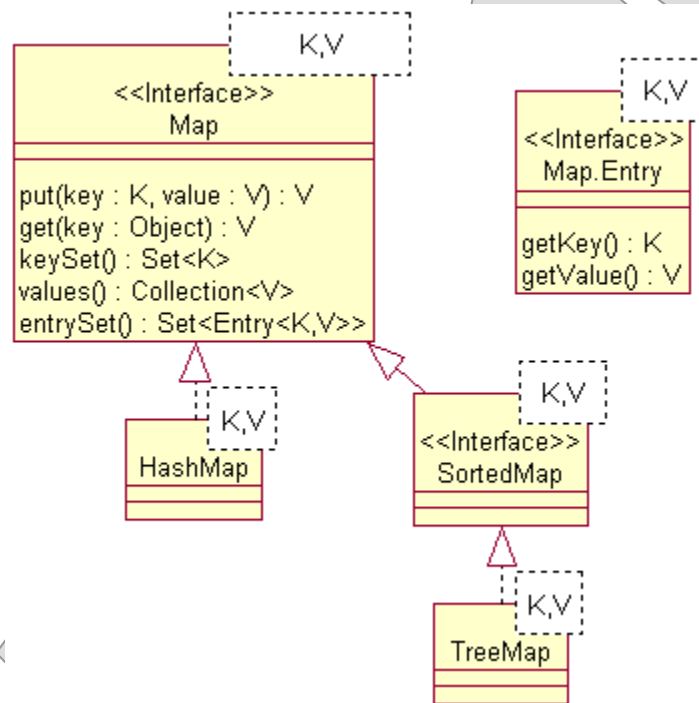
- Consider again the **getAllCars** method in **CarsII**.
  - It creates a temporary collection for the caller’s use.
  - This is not at all efficient: we’re making deep copies of the data, and the temporary collection is re-created for each call.
- If the collections are static, we could create the combined collection once and hold on to it.
  - But if that’s the case, why use dynamic collections at all?
- The “A” answer would not make copies of the data, but rather offer an **Iterator<Car>** on the total set.
  - This involves creating a custom iterator that knows how to traverse the two collections where they sit.
  - See **Examples/CarsII/CustomIterator/cc/cars/ Dealership.java** for a more thorough solution.



- Since **Iterator<E>** can’t be used in the simplified **for** loop, we also make our custom iterator implement **Iterable<E>**.

## Maps

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



- Keys must be unique.
- There can be only one value per key, and it may not be **null**.

## The Map<K,V> Interface

---

```
interface Map
{
    public int size ();
    public void clear ();
    public V get (Object key);
    public V put (K key,V value);
    public V remove (Object key);
    public boolean containsKey (Object key);
    public boolean containsValue (Object value);
    public Set<K> keySet ();
    public Collection<V> values ();
    public Set<Map.Entry<K,V>> entrySet ();
}
```

- Maps do provide **views** of their data as collections:
  - **keySet** provides just the keys.
  - **values** provides the values.
  - **entrySet** provides a **Set** of **Map.Entry** objects, each of which is a pair of key and value. (Map.Entry is an **inner class**, a feature we'll study in a later chapter.)

## 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<String,String> 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"));

// Either:
for (String ZIP : map.keySet ())
    System.out.println (ZIP + ": " + map.get (ZIP));

// Or:
for (Map.Entry<String,String> entry :
     map.entrySet ())
    System.out.println
        (entry.getKey () + ": " + entry.getValue ());
```

## Sorted Sets and Maps

---

- Collections that implement **SortedSet<E>** provide automatic insertion sorting whenever elements are added.
  - Thus the collection is always well ordered.
  - The only Core implementation is **TreeSet<E>**, which stores its elements in a binary tree.
- 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<K,V>** interface defines similar semantics for maps.
  - The only Core implementation is **TreeMap<K,V>**.

## 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 an earlier challenge lab, 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<String> set = new TreeSet<> ();
for (String name : names)
    set.add (name);
```

- It prints the contents of the sorted set:

```
for (String name : set)
    System.out.print (name + " ");
System.out.println ();
```

- 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 7B

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 7B write-up at the end of the chapter.

Suggested time: 30 minutes.

Evaluation Only

## The Collections Class Utility

---

- Since dynamic collections are so neatly polymorphic, and generic-therefore-type-safe, it is possible for the Core API to include a set of high-quality **algorithms** that operate on the collection types.
  - An algorithm isolates a generally useful behavior, such as iterating, searching, or sorting, from the underlying collection type.
  - It can also vary its behavior according to parameters: sort in a particular order, or search based on a certain criterion.
- The Collections API includes a single class utility **Collections**, with several very handy algorithms:
  - **sort** will sort an existing **List<E>**, with an overload that takes a **Comparator** for the type **E**.
  - **reverseOrder** will “invert” an existing **Comparator** or provide one that inverts the default comparator for a type.

```
Collections.sort (myList,  
Collections.reverseOrder ());
```

- Several other methods will re-order an existing list: **reverse**, **shuffle**, **swap**, and **rotate**.
- **min** and **max** will get a single object from a collection, and **frequency** will tell how many times an object occurs.
- **binarySearch** will find a given object on a given list (which must already be sorted).

## Algorithms

**EXAMPLE**

- In **Examples/Algorithms**, there is an application that puts a list of integers through its paces:

```
List<Integer> numbers = new ArrayList<> ();
Collections.addAll (numbers, 1, 2, 3, 4, 5);
// Another nice one! Using the varargs feature.
printList ("Start", numbers);
Collections.reverse (numbers);
printList ("Reverse", numbers);
Collections.shuffle (numbers);
printList ("Shuffle", numbers);
Collections.swap (numbers, 0, 4);
printList ("Swap", numbers);
System.out.format ("% -16s %d%n", "Minimum:",
    Collections.min (numbers));
System.out.format ("% -16s %d%n", "Maximum:",
    Collections.max (numbers));
Collections.sort (numbers);
printList ("Sort", numbers);
...
numbers.addAll (Collections.nCopies (3, 3));
printList ("Expanded", numbers);
System.out.format ("% -16s %d%n", "Frequency of 3:",
    Collections.frequency (numbers, 3));
Collections.replaceAll (numbers, 3, 9);
printList ("Replaced", numbers);
System.out.format ("% -16s %d%n", "Frequency of 3:",
    Collections.frequency (numbers, 3));
Collections.sort (numbers,
    Collections.reverseOrder ());
printList ("Descending", numbers);
...
System.out.format ("% -16s %d%n", "Position of 4:",
    Collections.binarySearch (numbers, 4));
```

## Algorithms

**EXAMPLE**

- Build and run the application to watch the show:

**build****run**

```
Start:          1 2 3 4 5
Reverse:       5 4 3 2 1
Shuffle:       1 2 5 3 4
Swap:          4 2 5 3 1
Minimum:       1
Maximum:       5
Sort:          1 2 3 4 5
Expanded:      1 2 3 4 5 5 4 3 2 1 3 3 3
Frequency of 3: 5
Replaced:     1 2 9 4 5 5 4 9 2 1 9 9 9
Frequency of 3: 0
Descending:   9 9 9 9 9 5 5 4 4 2 2 1 1
Position of 4: 7
Position of 4: -1
```

## Sorting Scores

**EXAMPLE**

- Another, simple example is in **Examples/Scores/Step8**.
  - The code that sifted the original scores into a new **ArrayList** is replaced by a call to **sort** in reverse order:

```
ArrayList<Record> sorted = new ArrayList<> (scores);
Collections.sort
    (sorted, Collections.reverseOrder ());
```

```
System.out.println ("Student      Score      Grade");
System.out.println ("-----      -----      -----");
for (Record record : sorted)
    System.out.format ("% -24s   %3d           %1s%n",
        record.name, record.score, record.grade);
```

- For this to work there must be a way of comparing **Record** objects
  - Java doesn't know a **natural order** for these things.
  - So **Record** implements **Comparable<Record>**:

```
class Record
    implements Comparable<Record>
{
    ...
    public Record (String name, int score)
    {
        this.name = name;
        this.score = score;
    }

    public int compareTo (Record other)
    {
        return new Integer (score).compareTo
            (new Integer (other.score));
    }
}
```

## Conversion Utilities

---

- Also in **Collections** are several sets of methods that convert from one sort of collection to another.
- The most interesting are the **unmodifiableXXX** methods: **unmodifiableList**, **unmodifiableSet**, etc.
  - Each of these creates an immutable “view” of an existing, mutable collection.
  - This is a great utility, one that addresses a design problem that comes up almost constantly: how to expose a collection for reading without risking unwanted modification by the client code?
  - The view will throw an **UnsupportedOperationException** if any attempt is made to modify the source collection through it, or through an iterator that it hands out.
- Another set of methods **synchronizedXXX** offer views of source collections that are thread-safe.
  - It may be a rude surprise to discover that most of the basic collection types are not thread-safe already!
  - The trade-off is made for performance in single-threaded use.
  - Where safety is needed, these methods can provide it.
- Finally the **checkedXXX** methods provide views that are “dynamically type-safe.”
  - This is strictly for integrating generics with legacy code – see Appendix B on Compatibility and Migration.

## SUMMARY

- Dynamic collections can make many coding tasks much easier, and the resulting code more maintainable.
- Java Generics offer excellent type safety, and also make code more self-documenting.
- 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.
- The **Collections** class utility offers a very handy set of algorithms and conversion utilities.
- In all, the **java.util** package is well named!
  - Use generic collections, iterators and algorithms liberally in your code.
  - Use arrays where collections of values or object references can be expected to stay the same once created, or where the very best possible performance is required.

## Better Management of Test Scores

LAB 7A

### Introduction

In this lab you will overhaul the **Scores** application 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.

**Directories:**           **Labs/Lab07A**                           (do your work here)  
                          **Examples/Scores/Step5**               (backup copy of starter files)  
                          **Examples/Scores/Step6**               (contains lab solution)

**Files:**                   **src/Scores.java**

### Instructions

1. Open **Scores.java** and note that the hard-coded data is provided in a new form. At the bottom of the source file there is a second, package-visible class **Record**, which is a simple data structure holding **name**, **score** and **grade**. What used to be three separate arrays for these three sets of values is now a single **List<Record>** – initialized to use an **ArrayList** as the list implementation. The list is now initialized with a series of **add** calls that create new **Records** that wrap the data. The rest of the code has not been adjusted to use the new collection, however; it's the old code that used the arrays. Therefore the code will neither compile nor run correctly. Let's fix that!
2. Next the method prints a header and loops over the old array. Change this to a simplified **for** loop:

```
for (Record record : scores)
```

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

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

4. 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, using the simplified **for** loop syntax.
5. In testing for a matching grade inside the inner loop, change **grades[student]** to the grade in the current **record**:

```
if (record.grade.equals (possibleGrades[g]))
```

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

7. Build and test at this point. You should get the same behavior as before:

**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:

- 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 **ArrayList<Record>**. You'll do this over the next few steps; to start with, declare a **List<Record> sorted** and initialize to a new **ArrayList<>**.
- Now build a nested loop: the outer loop (label it **Source:**) will iterate over **scores**, as usual, and the inner loop will iterate over **sorted**. This inner loop will need to use the full syntax of the **for** loop, because we need an integer index to operate on specific elements of the **sorted** collection.

```

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

```

- At the top of the inner loop, declare a local **Record** called **test** and set it to **sorted.get(s)**.
- If the **record.score** is greater than the **test.score**, call **sorted.add**, passing the index **s** and **record** (this overload of **add** inserts at a given index), and then **continue Source**;

12. If the inner loop falls through, then **record** belongs at the end of the growing **sorted** vector; pass it in a call to **sorted.add** (the overload that appends to the end of the list).
13. 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. (Or, if you like, use **System.out.format** this time, instead of the filling loop. The answer code takes this approach.)
14. 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 7B

### 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.

**Directories:**           **Labs/Lab07B**                           (do your work here)  
                          **Examples/Scores/Step6**                   (backup copy of starter files)  
                          **Examples/Scores/Step7**                   (contains lab solution)

**Files:**                   **src/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, initialize a **Map<String,Integer> grades** to a new **HashMap<>**.
3. Rip out the inner loop labeled **Grade:**, but leave the last line that accumulates values in **total**.
4. 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 = grades.get (record.grade);
```

5. 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 the value 1. (Thanks to auto-boxing, you don't need to instantiate a new **Integer** here – the compiler will do it.)
6. Otherwise, call **grades.put** and pass **record.grade** and **frequency + 1**. (Lots of auto-conversion here! Down to **int**, then add one, then back up to **Integer**.) Note that this will overwrite the value you just looked up, which means you've incremented the existing count.

7. After the code that prints the “Statistics:” header to the console, replace the loop over **possibleGrades** with an iteration over **grades.entrySet**:

```
for (Map.Entry<String,Integer> entry : grades.entrySet ())
```

8. For each entry, print a line of text formatted to show the grade and the number of times it occurred, which will be **entry.getKey** and **entry.getValue**.
9. 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.
```

10. 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.
```