

Lecture 20: Case Study: The Java Collections API

You can't be a competent Java programmer without understanding the crucial parts of the Java library. The basic types are all in `java.lang`, and are part of the language proper. The package `java.util` provides collections – sets, lists and maps – and you should know it well. The package `java.io` is also important, but you can get away with only a rough familiarity with what's in it, delving in as needed.

In this lecture, we'll look at the design of `java.util`, often called the Java 'collections API'. It's worth understanding not only because the collection classes are extremely useful, but also because the API is a nice example of well-engineered code. It's fairly easy to understand, and is very well documented. It was designed and written by Joshua Bloch, author of your recommended text *Effective Java*.

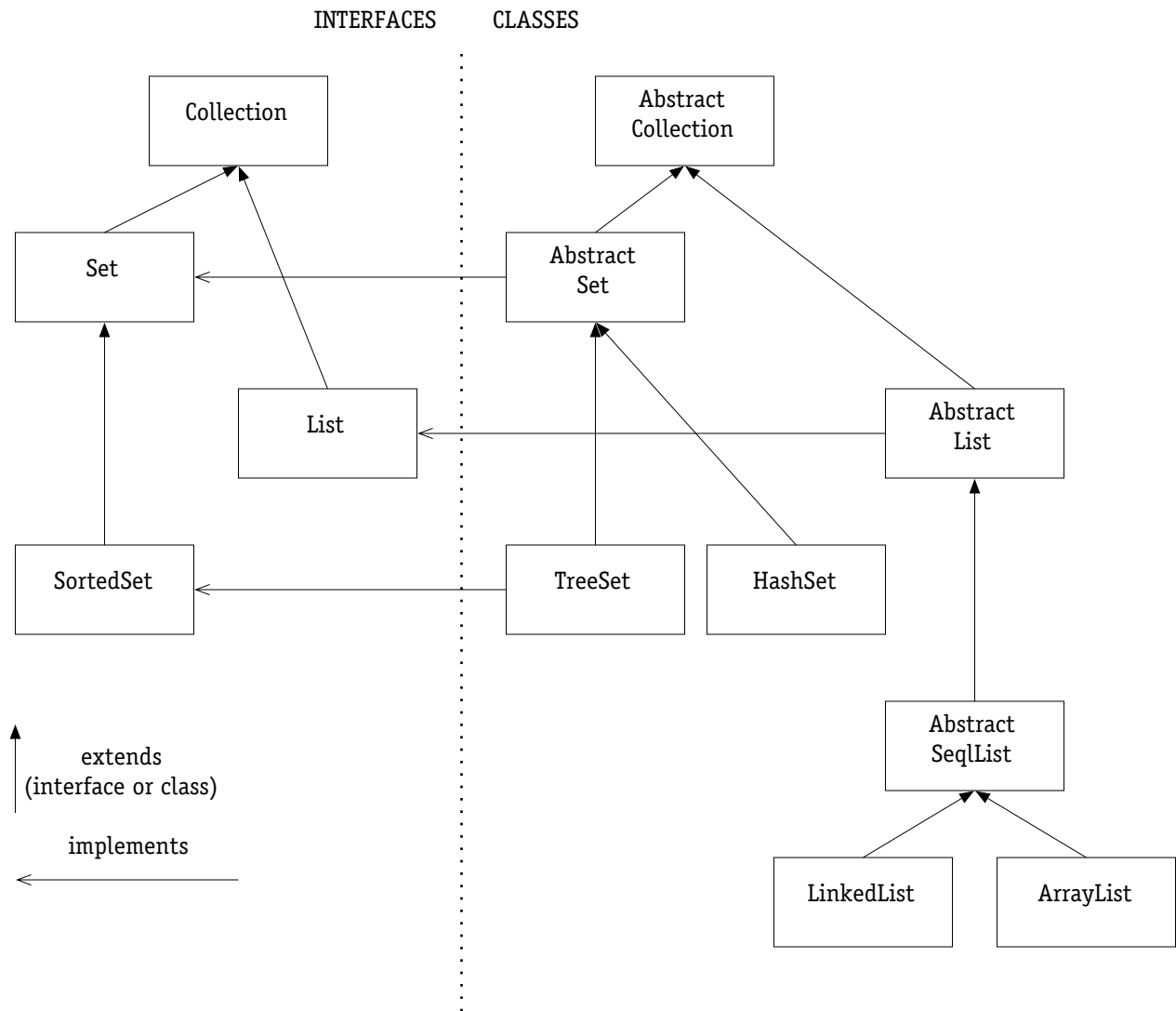
At the same time, though, almost all the complexities of object-oriented programming appear somewhere in it, so if you study the API carefully, you'll get a broad understanding of programming issues that you probably haven't yet considered in your own code. In fact, it wouldn't be an exaggeration to say that if you figure out how just one of the classes (eg, `ArrayList`) works in its entirety, then you will have mastered all the concepts of Java. We won't have time to look at all the issues today, but we'll touch on many of them. Some of them, such as serialization and synchronization, are beyond the scope of the course.

20.1 Type Hierarchy

Roughly, the API offers three kinds of collection: sets, lists and maps. A set is a collection of elements that maintains only whether an element is present, with no notion of order or repetition count – each element is either in the set or not. A list is a sequence of elements, and thus maintains both order and count. A map is an association between keys and values: it holds a set of keys, and maps each key to a single value.

The API organizes its classes with a hierarchy of interfaces – the specifications of the various types – and a separate hierarchy of implementation classes. The diagram shows some select classes and interfaces to illustrate this. The interface `Collection` captures the common properties of lists and sets, but not maps, but we'll use the informal term 'collections' to refer to maps anyway. `SortedMap` and `SortedSet` are used for maps and sets which provide additional operations to retrieve the elements in some order.

The concrete implementation classes, such as `LinkedList`, are built on top of skeletal implementations, such as `AbstractList`, from which they inherit. This parallel structure of interfaces and classes is an important idiom that is worth studying. Many novice programmers are tempted to use abstract classes when they should be using interfaces. But in general, you should prefer interfaces to abstract classes. You can't easily retrofit



an existing class to extend an abstract class (because a class can have at most one superclass), but it's usually easy to make it implement a new interface.

Bloch shows (in Item 16 of his book: 'Prefer interfaces to abstract classes') how to combine the advantages of both, using skeletal implementation classes, as he does here in the Collections API. You get the advantage of interfaces for specification-based decoupling, and the advantage of abstract classes to factor out shared code amongst related implementations.

Each Java interface comes with an informal specification in the Java API documentation. This is important because it tells a user of a class that implements an interface

what to expect. If you implement a class and claim that it meets the specification `List`, for example, you have to ensure that it meets the informal specification too, otherwise it will fail to behave according to programmers' expectations.

These specifications are intentionally incomplete (as many specifications often are). The concrete classes also have specifications, and these fill in some of the details of the interface specifications. The `List` interface, for example, doesn't say whether null elements can be stored, but `ArrayList` and `LinkedList` say explicitly that nulls are allowed. `HashMap` allows both null keys and null values, unlike `Hashtable`, which allows neither.

When you write code that uses collection classes, you should refer to an object by the most general interface or class possible. For example,

```
20.1.1 List p = new LinkedList ();
```

is better style than

```
20.1.2 LinkedList p = new LinkedList ();
```

If your code compiles with the former, then you can easily change to a different list implementation later:

```
20.1.3 List p = new ArrayList ();
```

since all the following code relied only on `p` being a `List`. With the latter, however, you may find that you can't make the change, because some other part of your program performs an operation on `x` that only `LinkedList` provides – an operation that in fact might not be needed. This is explained in more detail in Item 34 of Bloch's book ('Refer to objects by their interfaces').

20.2 Optional Methods

The collections API allows a class to claim to implement a collections interface without implementing all of its methods. For example, all the mutators of `List` are specified as optional. This means you can implement a class that satisfies the `List` specification, but which throws an `UnsupportedOperationException` whenever you call a mutator, such as `add`.

This intentional weakening of the `List` specification is problematic, because it means that if you're writing some code that receives a list, you don't know, in the absence of additional information about the list, whether it will support `add`.

But without this notion of optional operations, you'd have to declare a separate interface `ImmutableList`. These interfaces would proliferate. Sometimes, we want to require some mutators but not others. For example, the `keySet` method of `HashMap` returns a `Set` containing the keys of the map. This set is a view: deleting a key from the set causes a key

and its associated value to be deleted from the map. So remove is supported. But add is unsupported, since you can't add a key to a map without an associated value.

So the use of optional operations is a reasonable engineering judgment. It means less compile-time checking, but it reduces the number of interfaces. One way to view the idea of an optional operation is as a workaround to enable subtyping when it would not otherwise be present: a collection class whose add operation throws UnsupportedOperationException subtypes the Set interface because the spec of add in that interface explicitly permits the raising of this exception as a possible outcome. Of course, in practice the client of such a type will need to know that this does not happen, so there is a need for programmer-provided annotations and some discipline to overcome the lack of help from the compiler.

20.3 Polymorphism

All these containers – sets, lists and maps – take elements of type Object. They are said to be polymorphic, meaning ‘many shaped’, because they allow you to make different kinds of containers: lists of Integers, lists of URLs, lists of lists, and so on.

This kind of polymorphism is called *subtype polymorphism*, because it relies on the type hierarchy. A different form of polymorphism, called *parametric polymorphism*, allows you to define containers with type parameters, so that a client can indicate what type of element a particular containers will contain:

```
20.3.1 List[URL] bookmarks; // not legal Java
```

Java doesn't have this kind of polymorphism, although there have been many proposals to add it. Parametric polymorphism has the big advantage that the programmer can tell the compiler what type the elements have. The compiler can then catch errors in which an element of the wrong type is inserted, or an element that is extracted is treated as having a different type.

With subtype polymorphism, you have to explicitly cast the elements on extraction. Consider this code:

```
20.3.2 List bookmarks = new LinkedList ();
20.3.3 URL u = ...;
20.3.4 bookmarks.add (u);
20.3.5 ...
20.3.6 URL x = bookmarks.get (0); // compiler will reject this
```

The statement that adds u is fine, since the add method expects an Object, and URL is a subclass of Object. The statement that gets x, however, is broken, since the type of the expression on the RHS is Object, and you can't assign an Object to a variable of type URL,

since then you couldn't rely on that variable holding a URL. So a downcast is needed, and we have to write instead:

```
20.3.7    URL x = (URL) bookmarks.get (0);
```

The effect of the downcast is to perform a runtime check. If it succeeds, and the result of the method call is of type URL, execution continues normally. If it fails, because the result is not of the correct type, a `ClassCastException` is thrown, and the assignment is not performed. Make sure you understand this, and don't get confused into thinking (as students often do) that the cast somehow mutates the object returned by the method invocation. Objects carry their type at runtime, and if an object was created with a constructor from the class URL, it will have that type, and there is no need to somehow 'change it' to give it that type.

These downcasts can be a nuisance and occasionally it's worth writing a wrapper class just to factor them out. In a browser, you'd probably want a special abstract data type for a list of bookmarks anyway (to support other functionality). If you did this, you would perform the cast within the abstract type, and clients would see methods such as

```
20.3.8    URL getURL (int i);
```

which would not require the cast in their calling contexts, thus limiting the scope in which cast errors can occur.

Subtype polymorphism does give some flexibility that parametric polymorphism does not. You can form heterogeneous containers that contain different kinds of elements. And you can put containers inside themselves – try and figure out how to express this as a polymorphic type – although this is not usually a wise thing to do. In fact, as we mentioned in our earlier lecture on equality, the Java API classes will break if you do this.

Writing down what type of an element a container has is often the most important part of an abstract type's rep invariant. You should get into the habit of writing a comment whenever you declare a container, either using a pseudo-parametric type declaration:

```
20.3.9    List bookmarks; // List [URL]
```

or as part of the rep invariant proper:

```
20.3.10   RI: bookmarks.elems in URL
```

20.4 Skeletal Implementations

The concrete implementations of the collections build on skeletal implementations. These use the *Template Method* design pattern (see Gamma et al, pages 325–330). These abstract classes have no instance variables of their own, but define 'template methods' that call other 'hook methods' that are declared to be abstract and have no

code. In the inheriting subclass, the hook methods are overridden, and the template methods are inherited unchanged.

`AbstractList`, for example, makes `iterator` a template method that returns an iterator implemented using the `get` method as a hook. The `equals` method is implemented as another template in terms of `iterator`. A subclass, such as `ArrayList`, then provides a representation (such as an array of elements) and an implementation for `get` (such as getting the *i*th element of the array), and can inherit `iterator` and `equals`.

Some concrete classes replace the abstract implementations. `LinkedList`, for example, replaces the `iterator` functionality, since, with using the representation of list entries directly, it's possible to write a much more efficient traversal than using the hook method `get`, which does a sequential search for each call!

20.5 Capacity, Allocation & GC

An implementation that uses an array for its representation – such as `ArrayList` or `HashMap` – must select a size for the array when it is allocated. Choosing a good size can be important for performance. If it's too small, the array will have to be replaced by a new array, incurring the cost of allocating the new one and garbage collecting the old one. If it's too large, space will be wasted, which will be a problem especially when there are many instances of the collection type.

Such implementations therefore provide constructors in which the client can set an initial capacity, from which the allocation size can be determined. `ArrayList`, for example, has the constructor

```
20.5.1 public ArrayList(int initialCapacity)
20.5.2     Constructs an empty list with the specified initial capacity.
20.5.3 Parameters:
20.5.4     initialCapacity - the initial capacity of the list.
20.5.5 Throws:
20.5.6     IllegalArgumentException - if the specified initial capacity is negative
```

There are also methods that adjust the allocation: `trimToSize`, which sets the capacity so that the container is just large enough for the elements currently stored, and `ensureCapacity`, which increases capacity to some given amount. Note that these methods have no abstract effect at all; they are executed solely for their beneficent side effects. Slightly oddly, the decision to produce these side effects is made by the client of the type, not internally. There is therefore some violation of abstraction here, but it seems inevitable if the client is to be able to tweak the performance of the type.

Using the capacity features is tricky. If you don't have precise knowledge of how big your collections are for the particular application, you can run a profiler to find out.

Note that this notion of capacity translates a behavioural problem into a performance problem – a very desirable tradeoff. In many old programs, there are fixed resource limits, and when they are reached, the program just fails. With the capacity approach, the program just slows down. It's a good idea to design a program so that it works efficiently almost all the time, even if there's a performance hit occasionally.

If you study the implementation of the `remove` method in `ArrayList`, you'll see this code:

```
20.5.7   public Object remove(int index) {
20.5.8       ...
20.5.9       elementData[--size] = null; // Let gc do its work
20.5.10      ...
20.5.11      }
```

What's going on? Isn't garbage collection automatic? Herein lies a common mistake of many novice programmers. If you have an array in your representation, with a distinct instance variable holding an index to indicate which elements of the array are to be considered part of the abstract collection, it's tempting to think that to remove elements all you need to do is decrement this index. An analysis in terms of the abstraction function will support this confusion: elements that fall above the index are, after all, not considered part of the abstract collection, and their values are irrelevant.

There's a snag, however. If you fail to assign null to the unused slots, the elements whose references sit in those slots will not be garbage collected, even if there are no other references to those elements elsewhere in the program. The garbage collector can't read the abstraction function, so it doesn't know that those elements are not really reachable from the collection, even though they are reachable in the representation. If you forget to null out these slots, the performance of your program may suffer badly.

20.6 Copies, Conversions, Wrappers, etc

All the concrete collection classes provide constructors that take collections as arguments. These allow you to copy collections, and to convert one collection type to another. For example, `LinkedList` has

```
20.6.1   public LinkedList(Collection c)
20.6.2       Constructs a list containing the elements of the specified collection, in
           the order they are returned by the collection's iterator.
20.6.3   Parameters:
20.6.4       c – the collection whose elements are to be placed into this list.
```

which can be used for copying:

```
20.6.5   List p = new LinkedList ()
```

```
20.6.6    ...
20.6.7    List pCopy = new LinkedList (p)
```

or for creating a linked list from some other collection type:

```
20.6.8    Set s = new HashSet ()
20.6.9    ...
20.6.10   List p = new LinkedList (s)
```

Since constructors cannot be declared in interfaces, the specification `List` doesn't say that all of its implementations should have such constructors, although they do.

There is a special class *java.util.Collections* that contains a bunch of static methods that operate on, or return collections. Some of these are generic algorithms (eg, for sorting), and some are wrappers. For example, the method `unmodifiableList` takes a list and returns a list with the same elements, but which is immutable:

```
20.6.11   public static List unmodifiableList(List list)
20.6.12   Returns an unmodifiable view of the specified list. This method allows
modules to provide users with "read-only" access to internal lists. Query operations
on the returned list "read through" to the specified list, and attempts to modify
the returned list, whether direct or via its iterator, result in an UnsupportedOperation
Exception.
20.6.13   The returned list will be serializable if the specified list is serializable.
20.6.14   Parameters:
20.6.15   list - the list for which an unmodifiable view is to be returned.
20.6.16   Returns:
20.6.17   an unmodifiable view of the specified list.
```

The list returned isn't exactly immutable, since its value can change because of modifications to the underlying list (see Section 19.8 below), but it can't be modified directly. There are similar methods that take collections and return wrapped versions that are synchronized.

20.7 Sorted Collections

A sorted collection must have some way to compare elements to determine their order. The Collections API offers two approaches to this. You can use the 'natural ordering,' which is determined by using the `compareTo` method on the element type from the interface *java.lang.Comparable*:

```
20.7.1    public int compareTo(Object o)
```

which returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the given object `o`. When you add an element to a sorted col-

lection that is using the natural ordering, the element must have been constructed in a class that implements the `Comparable` interface. The `add` method downcasts the element to `Comparable` in order to compare it with the existing elements in the collection, so if this was not the case, it will throw a class cast exception.

Alternatively, you can use an ordering given independently of the elements, as an object that implements the interface `java.util.Comparator`, which has the method

```
20.7.2 public int compare(Object o1, Object o2)
```

which is just like `compareTo`, but takes both elements to be compared as arguments. This is an instance of the Strategy pattern, in which an algorithm is decoupled from the code that uses it (see Gamma, pp. 315–323).

Which approach is used depends on which constructor you use to create the collection object. If you use the constructor that takes a `Comparator` as an argument, that will be used to determine the ordering; if you use the no-argument constructor, the natural ordering will be used.

Comparison suffers from the same problems as equality, which we discussed in detail in a previous lecture on equality and copying. A sorted collection has a rep invariant that the elements of the representation are sorted. If the ordering of two elements can be changed by mutating one of them through a public method call, a rep exposure occurs.

20.8 Views

Views are a sophisticated mechanism, very useful now and then, but dangerous. They break many of our basic conceptions about what kinds of behaviour can occur in a well-formed object-oriented program.

Three kinds of views can be identified, according to their purpose:

- *Functionality extension.* Some views are provided to extend the functionality of an object without adding new methods to its class. Iterators fall in this category. One could instead put the methods `next` and `hasNext` in the collection class itself. But this would complicate the API of the class itself. It would also be hard to support multiple iterations over the same collection. We could add a `reset` method to the class which is called to restart an iteration, but this would only allow one iteration at a time. Such a method would also lead to errors in which the programmer forgets to reset.
- *Decoupling.* Some views provide a subset of the functionality of the underlying collection. The `keySet` method on `Map`, for example, returns a set that consists of the keys of the map. It therefore allows part of the code that is only concerned with the

keys, and not with the values, to be decoupled from the rest of the specification of `Map`.

- *Coordinate Transformation.* The view provided by the `subList` method of `List` gives a kind of coordinate transformation. Mutations on the view produce mutations on the underlying list, but allow access to the list by an indexing that is offset by the parameter passed to the `subList` method.

Views are dangerous for two reasons. First, things change underneath you: call `remove` on an iterator and its underlying collection changes; call `remove` on a map and its key set view changes (and vice versa). This is a form of abstract aliasing in which a mutation to one object causes another object, of a different type to change. The two objects need not even be within the same lexical scope. Note that the meaning of our `modifies` clause in specifications must be refined: if you say *modifies* `c` and `c` has a view `v`, does that mean that `v` can change also?

Second, the specification of a method that returns a view often limits the kinds of mutation that are allowed. To make sure that your code works, you'll need to understand this specification. And not surprisingly, these specifications are often obscure. The `post-requires` clause of the Liskov text is one way to extend our specification notion to handle some of the complications.

Some views allow only the underlying collection to be mutated. Others allow only the view to be mutated – iterators, for example. Some allow mutations to both the view and the underlying collection, but place complex stipulations on the mutations. The Collections API, for example, says that when a `subList` view has been taken on a list, this underlying list must not suffer any 'structural modifications'; it explains this term rather obliquely as follows:

20.8.1 Structural modifications are those that change the size of this list, or otherwise perturb it in such a fashion that iterations in progress may yield incorrect results.

It's not clear exactly what this means. My inclination would be to avoid any modifications of the underlying list.

The situation is complicated further by the possibility of multiple views on the same underlying collection. You can have multiple iterators on the same list, for example. In this case, you have to also consider interactions between views. If you modify the list through one of its iterators, the other iterators will be invalidated, and must not be used subsequently.

There are some useful strategies that mitigate the complexity of views. If you are using a view, you should think carefully about whether these will help:

- You can limit the scope in which the view is accessible. For example, by using a `for`-loop rather than a `while`-loop for an iterator, you can limit the scope of the iterator's

declaration to the loop itself. This makes it much easier to ensure that there aren't any unintended interactions during iteration.

- You can prevent mutation of a view or underlying object by wrapping it using a method of the Collections class. For example, if you take a `keySet` view on a map, and don't intend to modify it, you could make the set immutable:

```
20.8.2    Set s = map.keySet ();
```

```
20.8.3    Set safe_s = Collections.unmodifiableSet (s);
```