PIC 20A
Collections and Data Structures

David Hyde
UCLA Mathematics

Last edited: May 20, 2020

Introductory example

How do you write a phone book program?

Some programmers may yell “hash table!” and write the pseudo-code:

```
HashTable phoneBook = new HashTable()
for each (name, phoneNumber) {
  phoneBook.put(name, phoneNumber)
}
print( "Sam's phone number is" )
print( phoneBook.get("Sam") )
```

This is correct, but the reasoning the skips an important step.

Introductory example

The hash table is good for this problem for 2 separate reasons.

- The interface of the hash table (it's `put` and `get` functions) are convenient for this application. I.e., the code is easy to write.
- The data structure that implements the interface is efficient. I.e., the code is fast.

Java separates these considerations via `interfaces` and their implementations.

Outline

interfaces to data structures

The `interface` specifies how to use the `Collection` and `Map`.

If the `interface` is convenient to use for a problem,
than its implementations are probably fast/efficient for the problem.

So it is often, but not always, enough to understand only the `interface`
and not the specific implementation.

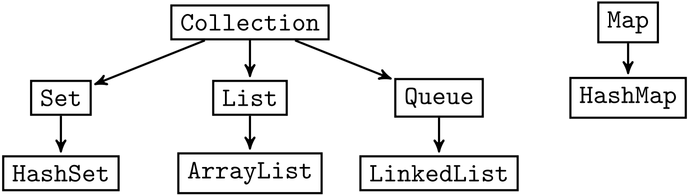
Interface Collection

The `interface` `Collection<E>` represents a group of `Object`s/elements of type `E`.

`Collection<E>` supports operations like `add`, `contains`, `remove`, `size`, and `toArray`.

`Collection<E>` is `abstract` (since it's an `interface`) and is inherited by more specific `interface`s.

Collection interface hierarchy



`Map` is not a `Collection`, but we'll talk about it anyways.

Interface Collection

```
Collection<String> c = new ArrayList<String>();

c.add("Alice");
c.add("Bob");
c.add("Carol");

System.out.println(c.contains("Eric")); //false

c.remove("Carol");
c.remove("Dave"); //does nothing

System.out.println(c.size()); //2

String[] s_arr = c.toArray();

for (String s : c) //c is Iterable<String>
  System.out.println(s);
```

Interface Set

`interface Set<E>` is a `Collection<E>` that doesn't contain duplicate elements. It's like the mathematical set.

```
public class CouponCollector {
  public static void main(String[] args) {
    Set<Integer> s = new HashSet<>(); //diamond
    Random random = new Random();
    int count = 0;
    while (s.size()<100) {
      s.add(random.nextInt(100)); //autoboxing
      count++;
    }
    System.out.println("Bought " + count
               + " coupons to collect all 100");
  }
}
```

Interface List

`interface List<E>` is a `Collection<E>` that

- allows duplicates,
- orders the elements,
- supports positional access via methods like `get(int index)` and `set(int index, E)`, and
- supports resizing (as most all `Collections` do) via methods like `add` and `remove`.

`ArrayList<E>`, a variable-length array, is the most used `List<E>`.

Interface Queue

`interface Queue<E>` is a `Collection<E>` that

- allows duplicates and
- typically (but not always) supports first-in, first-out (FIFO) access with methods like `add(E e)` and `remove()`.

FIFO means you retrieve elements in the order you add them.

Queue

A printer prints `Documents` in the order they arrive at.

```java
public class PrinterManager {
  //LinkedList supports FIFO access
  private Queue<Document> q = new LinkedList<>();
  public void addJob(Document d) {
    q.add(d);
  }
  public Document nextJob() {
    return q.remove();
  }
  public boolean jobDone() {
    return q.isEmpty();
  }
}
```

Map

`Map<K,V>` maps `Objects` of type `K` (key) to `Objects` of type `V` (value).
`Map<K,V>` is not a `Collection<E>`.

```
public class PhoneBook {
  private Map<String,Integer> m
                                = new HashMap<>();
  public void addNum(String name, Integer num) {
    m.put(name, num);
  }
  public Integer getNum(String name) {
    return m.get(name);
  }
  public boolean haveNumber(String name) {
    return m.containsKey(name);
  }
}
```

Outline

Why use Collections and Maps?

Whatever you do with `Collections` and `Maps` can be done with basic arrays.

However, the `interfaces` provide a convenient interface, and their implementations are usually faster than using basic arrays.

The `Collection` and `Map` `interfaces` are implemented with *data structures*, which are often quite sophisticated and complicated.

Different implementations use different data structures and provide related but different features.

Set implementations

`HashSet` is most common and usually the fastest.
`LinkedHashSet` preserves *insertion-order*.

```
Set<Integer> s1 = new HashSet<>();
Set<Integer> s2 = new LinkedHashSet<>();
for (int i=10; i>0; i--){
  s1.add(i); s2.add(i);
}
for (Integer i : s1)
  System.out.println(i); //order unpredictable
for (Integer i : s2)
  System.out.println(i); //ordered as added
```

List implementations

`ArrayList` is most common and usually the fastest.
`LinkedList` can be faster depending on where you insert the elements.

```
List<Character> l1 = new ArrayList<>();
List<Character> l2 = new LinkedList<>();
long start,end;
start = System.currentTimeMillis();
for (int i=0; i<100000; i++)
  l1.add(0,'a');
end = System.currentTimeMillis();
System.out.println(end-start);
start = System.currentTimeMillis();
for (int i=0; i<100000; i++)
  l2.add(0,'a');
end = System.currentTimeMillis();
System.out.println(end-start);
```

Queue implementations

`LinkedList` is most commonly used. It supports FIFO access and is usually the fastest. (Note that `LinkedList` is both a `Queue` and a `List`.)

`PriorityQueue` uses an ordering that is not FIFO. We'll talk about `PriorityQueue` later.

Map implementations

`HashMap` is most common and usually the fastest.

`LinkedHashMap` preserves insertion-order.

Outline

Motivating example

Should this be allowed?

```
ArrayList<Integer> l1 = new ArrayList<Integer>();
ArrayList<Object> l2 = l1;
```

In other words, should `ArrayList<Object>` be a superclass of `ArrayList<Integer>`?

Motivating example

Should this be allowed?

```
ArrayList<Integer> l1 = new ArrayList<Integer>();
ArrayList<Object> l2 = l1;
```

In other words, should `ArrayList<Object>` be a superclass of `ArrayList<Integer>`?

No, and it isn't. Otherewise we could do

```
ArrayList<Integer> l1 = new ArrayList<Integer>();
ArrayList<Object> l2 = l1;
l2.add("Hello"); //terrible!
```

Motivating example

`Object` is a superclass of `Integer`, but `ArrayList<Object>` is not a superclass of `ArrayList<Integer>`.

Instead, we can use `?`, the *wildcard*.

```
ArrayList<Integer> l1 = new ArrayList<Integer>();
ArrayList<String> l2 = new ArrayList<String>();
ArrayList<?> l3;
l3 = l1;
l3 = l2;
```

`ArrayList<?>` is a superclass of `ArrayList<Integer>` and `ArrayList<String>`.

Methods with wildcards

You can use wildcards to define methods.

```java
public static void printList(List<?> list) {
  for (Object elem: list)
    System.out.print(elem + " ");
}
```

You can do the same with an explicit type parameter.

```java
public static <T> void printList(List<T> list) {
  for (Object elem: list)
    System.out.print(elem + " ");
}
```

Upper bounded wildcards

You can *upper bound* a wildcard to require it be a subclass or implementation. This is the same idea as bounding explicit type parameters.

```
static void sum(List<? extends Number> list) {
  double sum = 0;
  for (Number n: list)
    sum += n.doubleValue();
  System.out.println(sum);
}
```

Upper bounded wildcards

The upper bound doesn't allow us to add `Object`s to the `List`.

```
static void fn(List<? extends Number> list) {
  Double d = new Double(3.3);
  Number n = d;
  list.add(d); //error
  list.add(n); //error
}
```

Maybe `list` is a `List<Integer>` or a `List<Rational>`. The compiler doesn't know.

Lower bounded wildcards

You can *lower bound* a wildcard to require it be a superclass.

```
static void addNum(List<? super Integer> list) {
  for(int i=0; i<10; i++)
    list.add(new Integer(i));
}
```

`list` could be a `List<Integer>`, a `List<Object>`, or a `List<Number>`. In any case, the `add` is valid.

Lower bounded wildcards

The lower bound doesn't allow us to get from the `List` gracefully.

```
static void fn(List<? super Integer> list) {
  Integer i = list.get(0); //error
  Number n = list.get(0); //error
}
```

Maybe `list` is a `List<Obect>` that contains `Strings`. The compiler doesn't know.

Wildcards vs. type parameter

You can use generics with a type parameter and/or with a wildcard.

- You can use wildcards for method inputs and return values.
- You cannot use wildcards to define generic `classes`.
  You must use explicit type parameters.
- You can upper and lower bound wildcards.
- You can only upper bound type parameters.
- Wildcards are arguably more confusing.

Wildcards vs. type parameter

- Usually, upper bounded wildcards are useful when you mostly read from the `Object`.
- Usually, lower bounded wildcards are useful when you mostly write to the `Object`.
- Usually, unbounded wildcards are useful when you can accomplish everything by treating the `Objects` as just `Objects`.
- Mostly, type parameters can do what wildcards can do, but not the other way around. There are a few things (like lower bounding) that only wildcards can do.
- A function signature is more concise when written with a wildcard.

Wildcards examples

`Collection<E>` has the method `removeAll`. Here's how you might implement it.

```
boolean removeAll(Collection<?> c) {
  for (Object o : c)
    remove(o);
  //return something
  ...
}
```

We use the fact that `Collection` contains `Object`s.
We know `?` inherits `Object`, and that's all we need.

Wildcards examples

`Collection<E>` has the method `addAll`. Here's how you might implement it.

```
boolean addAll(Collection<? extends E> c) {
  for (E e : c)
    add(e);
  //return something
  ...
}
```

What we add to a `Collection<E>`, must be an `E`.
The upper bound guarantees `?` is an `E`.

Outline

Collections

The utility class `Collections` (not to be confused with `Collection<E>`) contains polymorphic algorithms for `Collection<E>`s.

The algorithms only use methods specified by the `interfaces` (i.e., they are polymorphic) and they therefore work with a wide range of `Collection<E>`s.

Collections examples

`Collections` has the method `frequency`. Here's how you might implement it.

```
public static int
          frequency(Collection<?> c, Object o) {
  int count = 0;
  for (Object o2 : c)
    if (o2.equals(o))
      count++;
  return count;
}
```

Again, we use the fact that `Collection` contains `Object`s.

Collections examples

`Collections` has the method `shuffle`. Here's how you might implement it.

```
public static void shuffle(List<?> list) {
  Random r = new Random();
  for (int i=0; i<list.size(); i++) {
    j = r.nextInt(i);
    Collections.swap(list,i,j);
  }
}
```

Collections examples

`Collections` has the method `copy`. Here's how you might implement it.

```
public static <T> void
copy(List<? super T> dest, List<? extends T> src)
{
  for (T t : src)
    dest.add(t);
}
```

The lower bound guarantees you can add `T` to `dest`.
The upper bound guarantees `src` contains `T`.

Outline

You need an ordering for max or sort

Imagine you have

```
public class Complex {
  public final double real, imag;
  ...
}
```

and you try to sort a `List` of them

```
List<Complex> list = new ArrayList<>();
...
Collections.sort(list);
```

What could the sort possibly mean? `Collections.sort` sorts from "smallest" to "biggest", but what is "small" and what is "big"?

interface Comparator

The `interface` `Comparator<T>` compares a `T` against a `T`.

`Comparator<T>` demands

```
public int compare(T t1, T t2)
```

be implemented.

interface Comparator

For `class Complex`, we could use the *lexicographical* ordering

```
public class Lex implements Comparator<Complex> {
  public int compare(Complex c1, Complex c2) {
    if (c1.real<c2.real) return -1;
    else if (c1.real>c2.real) return 1;
    else if (c1.imag<c2.imag) return -1;
    else if (c1.imag>c2.imag) return 1;
    else return 0;
  }
}
```

interface Comparator

`Collections` has the method
`max(Collection<? extends T>, Comparator<? super T>)`.

If the first input is a `List`, here's how you might implement it.

```
public static <T> T max(List<? extends T> list,
                        Comparator<? super T> comp) {
  T currMax = list.get(0);
  for (int i=1; i<list.size(); i++)
    if (comp.compare(currMax, list.get(i))<0)
      currMax = list.get(i);
  return currMax;
}
```

If the input is a general `Collection`, you must use its `Iterator`.

interface Comparator

Now you can do things like

```
Collection<Complex> coll = new LinkedList<>();
List<Complex> list = new ArrayList<>();
...
Complex maxC = Collections.max(coll, new Lex());
Collections.sort(list, new Lex());
```

If you want to sort in another way you can write a different `Comparator<T>`.

reversed Comparator

The `default` method `reversed()` returns a `Comparator<T>` with reversed ordering.

```
Collection<Complex> coll = new LinkedList<>();
...
Complex maxC =
  Collections.max(coll, new Lex());
Complex minC =
  Collections.max(coll, (new Lex()).reversed());
```

We don't have to write `reversed()` since it's `default`.

interface Comparable

When a `class T` has a *natural ordering*, you can implement the `interface Comparable<T>`.

`Comparable<T>` requires

```
public int compareTo(T o)
```

to be implemented.

interface Comparable

```
public class Rational extends Number
                 implements Comparable<Number> {
  ...
  public int compareTo(Number rhs) {
    if (doubleValue()<rhs.doubleValue())
      return -1;
    else if (doubleValue()>rhs.doubleValue())
      return 1;
    else
      return 0;
  }
}
```

(This implementation can run into round-off errors though.)

interface Comparable

When you have a `Collection` or `List` of `Comparable`s, `max` and `sort` can use the elements' natural ordering.

The first `Collections.max` has signature

```
public static <T extends Comparable<? super T>>
                          T max(Collection<T> coll)
```

This uses the natural ordering provided by the `interface Comparable`.

The second `Collections.max` has signature

```
public static <T> T max(
                     Collection<? extends T> coll,
                     Comparator<? super T> comp)
```

This uses the ordering provided by the `interface Comparator`.

What do the wildcards mean?

```
<T extends Comparable<? super T>>
```

T is comparable to T (and possibly other things).

```
Collection<? extends T> coll
```

coll contains Ts (or descendents of T).

```
Comparator<? super T> comp
```

comp can compare a T to a T (and a superclass of T to a superclass of T).

PriorityQueue

Not all `Queue<E>`s support FIFO access. `PriorityQueue<E>` places elements with priority at the head. "Small" means high priority.

The constructor

```
public PriorityQueue()
```

creates a `PriorityQueue<E>` that processes its elements according to their natural ordering.

The constructor

```
public PriorityQueue(Comparator<? super E> comp)
```

creates a `PriorityQueue<E>` that processes its elements according to the `Comparator<? super E>`.

Outline

interface Iterable

We can use the enhanced for loop with a `Collection<E>` since it extends `Iterable<E>`.

```
Collection<?> coll;
...
for (Object o : coll)
  System.out.println(o);
```

interface Iterable

`interface Iterable<E>` requires

```
public Iterator<E> iterator()
```

to be implemented.

Each call of `iterator()` returns a new instance of `Iterator<E>`.

interface Iterator

`interface Iterator<E>` requires

```
public boolean hasNext()
```

and

```
public E next()
```

to be implemented.

Unpacking the enhanced for loop

Using the `Iterator`, we can write a `while` loop equivalent to the enhanced `for` loop.

```
Collection<Number> coll;
...
Iterator<Number> itr = coll.iterator();
while ( itr.hasNext() ) {
  Number n = itr.next();
  System.out.println(n);
}
```

Unpacking the enhanced for loop

You can do the same with a normal `for` loop.

```
Collection<Number> coll;
...
for (Iterator<Number> itr = coll.iterator();
                                          itr.hasNext(); ) {
  Number n = itr.next();
  System.out.println(n);
}
```

(I show this only for educational purposes. Always use the enhanced for loop when you can.)

max with Iterator

Here's how you might implement `Collections.max`.

```
static <T> T max(Collection<? extends T> coll,
                 Comparator<? super T> comp) {
  Iterator<T> itr = coll.iterator();
  T currMax = itr.next();
  while (itr.hasNext()) {
    T currT = itr.next();
    if (comp.compare(currMax, currT) < 0)
      currMax = currT;
  }
  return currMax;
}
```

Writing an Iterable

```
public class <E> ArrayList<E>
    extends AbstractList<E>
    implements List<E>, Iterable<E>, ... {
  ...
  public Iterator<E> iterator() {
    return new Itr();
  }
  private class Itr implements Iterator<E> {
    private int currInd = 0;
    public boolean hasNext() {
      return currInd<size();
    }
    public E next() { return get(currInd++); }
  }
}
```

(Why must `Itr` be an inner `class`, and not a top-level or a `static` nested `class`?)

A new Iterator starts anew

Each call of `iterator()` returns a new intance of `Iterator<E>` that starts anew.

```
Collection<Integer> coll = new ArrayList<>();
coll.add(1); coll.add(2); coll.add(3);

for (Iterator<?> itr = coll.iterator();
     itr.hasNext(); )
  System.out.print(itr.next() + " ");

for (Iterator<?> itr = coll.iterator();
     itr.hasNext(); )
  System.out.print(itr.next() + " ");
```

The output is `1 2 3 1 2 3`.

Iterating through a Map

A `Map<K,V>` and is not an `Iterable<E>`. (Remember, a `Map<K,V>` is not a `Collection<E>`.)

You can iterate through the keys of `Map<K,V>` with a `Set<K>`.

```
Map<String,Integer> m = new HashMap<>();
...
for (String key : m.keySet())
  System.out.println(key);
```

Iterating through a Map

You can iterate through the key-value pairs of `Map<K,V>` with a `Set<Map.Entry<K,V>>`.

```
Map<String,Integer> m = new HashMap<>();
...
for (Map.Entry<String, Integer> entry
                                : m.entrySet()) {
  String key = entry.getKey();
  Integer value = entry.getValue();
  ...
}
```

`Entry` is a `public static interface` of `class Map`.

Outline

Some features are optional

The documentation for `interface Collection<E>` says,

```
public boolean add(E e)
```

"Ensures that this collection contains the specified element (optional operation)."

All `Collection<E>`s must "implement" `add(E e)`, but some will throw the `Exception` `java.lang.UnsupportedOperationException`.

Some features are optional

```java
Set<Number> s = new HashSet<>();
...
Set<Number> const_s =
                 Collections.unmodifiableSet(s);
const_s.add(7); //UnsupportedOperationException
System.out.println(const_s.getClass());
```

`const_s.getClass()` is `java.util.Collections$UnmodifiableSet`.

Some features are optional

`UnmodifiableSet` is a `private static` nested `class` of `Collections`. You can't created references or instances of it from the outside

```
Collections.UnmodifiableSet<Number> const_s;
//error nested class is private
```

but `Collections` can return an instance of it to you.

Optional features are anti-polymorphic

To keep the library size manageable, Java doesn't provide separate interfaces for each variant of each collection.

Instead, some operations are designated optional and `throws` an `UnsupportedOperationException` when called.

This goes against the spirit of polymorphism, but the authors of the Java API had to make a trade-off.

Outline

method hashCode

`Collections` or `Maps` named `HashX` use the method `hashCode` of `class Object`.

General contract of `hashCode`: two equal `Objects` must have the same `hashCode`.

So when

```
o1.equals(o2)
```

is `true`, then

```
o1.hashCode()==o2.hashCode()
```

must be `true`, where `o1` and `o2` are `Objects`.

method hashCode

When you override `equals`, you should also override `hashCode`. Otherwise, you take a significant performance hit when using `HashX`.

What a “hash code” is is beyond the scope of this course.

For optimal performance, you want a “good” hash code. This discussion is also beyond the scope of this course.

If you don’t know what a hash code is, you can follow the simple instructions of *Effective Java* by Joshua Bloch Section 9. The instructions will give you a decent hash code.

Just remember: Always override `hashCode` when you override `equals`.

method hashCode

In this example, we override `equals`.

```
class Complex {
  public final int real, imag;
  public Complex(int r, int i) {real=r; imag=i;}
  @Override
  public boolean equals(Object o) {
    if (o instanceof Complex)
      return (real == ((Complex) o).real)
          && (imag == ((Complex) o).imag);
    else
      return false;
  }
  ...
```

method hashCode

We override `equals` according to *Effective Java* Section 9.

```
  ...
  @Override
  public int hashCode() {
    int result = 17;
    result = 31 * result + real;
    result = 31 * result + imag;
    return result;
  }
}
```

method hashCode

```
public static void main(String[] args) {
  Set<Complex> s = new HashSet<>();
  Random rand = new Random();
  long start = System.currentTimeMillis();
  for (int i=0; i<10_000_000; i++) {
    int real = rand.nextInt(10);
    int imag = rand.nextInt(10);
    s.add(new Complex(real,imag));
  }
  long end = System.currentTimeMillis();
  System.out.println(end-start);
}
```

Without overriding `hashCode()`, this code takes `2.65`s to run.
When we override `hashCode()`, this code takes `0.47`s.