The basic list datatype...

```
interface IList<X> {
  boolean isEmpty();
  X getFirst();
  IList<X> getRest();
  ....
}

class Empty<X> implements IList<X> {
  Empty() { }
  ....
}

class Cons<X> implements IList<X> {
  X first;
  IList<X> rest;
  Cons(X first, IList<X> rest) {
    this.first = first;
    this.rest = rest;
  }
  ....
}
```

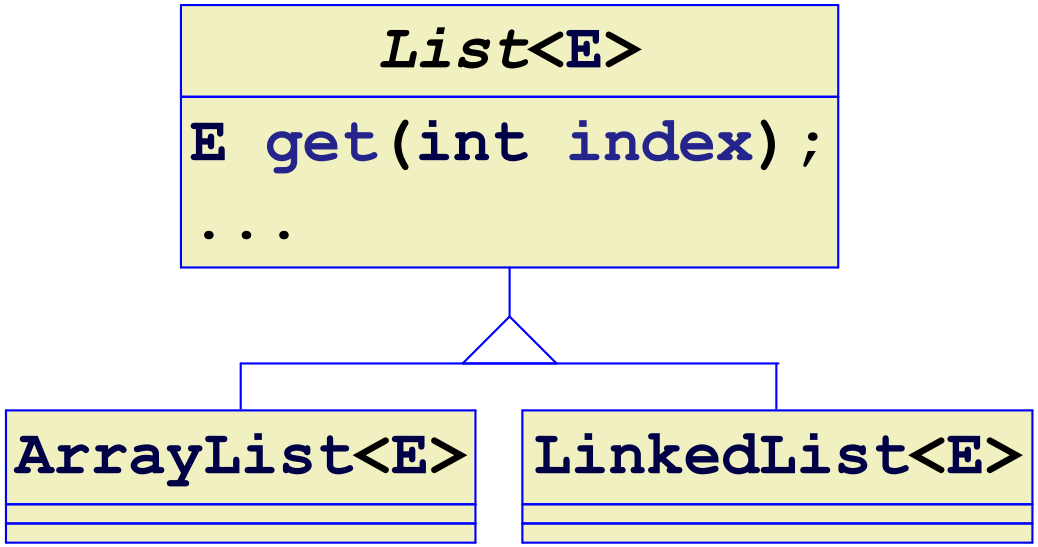
... is not in any standard Java library

List Containers

```
interface List<E> {
  int size();
  boolean isEmpty();
  boolean add(E element);
  E get(int pos);

  boolean contains(E element);
  boolean remove(E element);
  ....
}
```

List Container Classes



List Container in Java

```
import java.util.*;

List<String> ls = new LinkedList<String>();
ls.add("apple");
ls.add("banana");
ls.add("coconut");
t.checkExpect(ls.get(0), "apple");
t.checkExpect(ls.get(1), "banana");
```

List Container in Racket

```
(define-struct linked-list (content))

(define (create-linked-list)
  (make-linked-list empty))

(define (add! ll v)
  (set-linked-list-content!
   ll
   (append (linked-list-content ll) (list v))))

(define (get ll i)
  (list-ref (linked-list-content ll) i))

(define a-ll (create-linked-list))
(add! a-ll "apple")
(add! a-ll "banana")
(add! a-ll "coconut")
(check-expect (get a-ll 0) "apple")
(check-expect (get a-ll 1) "banana")
```

Concatenating Strings

```
t.checkExpect(concatAll(ls), "applebananacoconut");

static String concat(List<String> ls) {
  String accum = "";
  for (String s : ls) {
    accum = accum + s;
  }
  return accum;
}
```

Copy

Concatenating Strings

```
t.checkExpect(concatAll(ls), "applebananacoconut");

static String concat(List<String> ls) {
  String accum = "";
  for (String s : ls) {
    ac
  }
  retu
}
```

for is roughly like **map** fused with **lambda**

Copy

Concatenating Strings

```
t.checkExpect(concatAll(ls), "applebananacoconut");

static String concat(List<String> ls) {
  String accum = "";
  for (String s : ls) {
    accu = accum + s;
  }
  return accum;
}
```

always a (next

Copy

Concatenating Strings

```
t.checkExpect(concatAll(ls), "applebananacoconut");

static String concat(List<String> ls) {
  String accum = "";
  for (String s : ls) {
    accum = ac      m + s;
  }
  return accum
}
```

the type of each element

Copy

Concatenating Strings

```
t.checkExpect(concatAll(ls), "applebananacoconut");

static String concat(List<String> ls) {
  String accum = "";
  for (String s : ls) {
    accum = accu    + s;
  }
  return accum;
}
```

a variable bound to each element of the lis in turn

Copy

Concatenating Strings

```
t.checkExpect(concatAll(ls), "applebananacoconut");

static String concat(List<String> ls) {
  String accum = "";
  for (String s : ls) {
    accum = accum + s;
  }
  return accum;
}
```

a : after the variable

Copy

Concatenating Strings

```
t.checkExpect(concatAll(ls), "applebananacoconut");

static String concat(List<String> ls) {
  String accum = "";
  for (String s : ls) {
    accum = accum +
  }
  return accum;
}
```

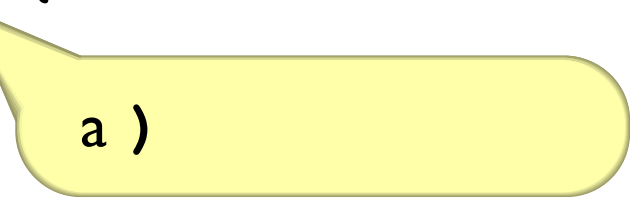
an expression of type **List<E>** for elements of type **E**

Copy

Concatenating Strings

```
t.checkExpect(concatAll(ls), "applebananacoconut");

static String concat(List<String> ls) {
  String accum = "";
  for (String s : ls) {
    accum = accum + s
  }
  return accum;
}
```



Copy

Concatenating Strings

```
t.checkExpect(concatAll(ls), "applebananacoconut");

static String concat(List<String> ls) {
  String accum = "";
  for (String s : ls) {
    accum = accum + s;
  }
  return acc
}
```

a `for` form doesn't return anything; to get a value out, you have to set a variable

Copy

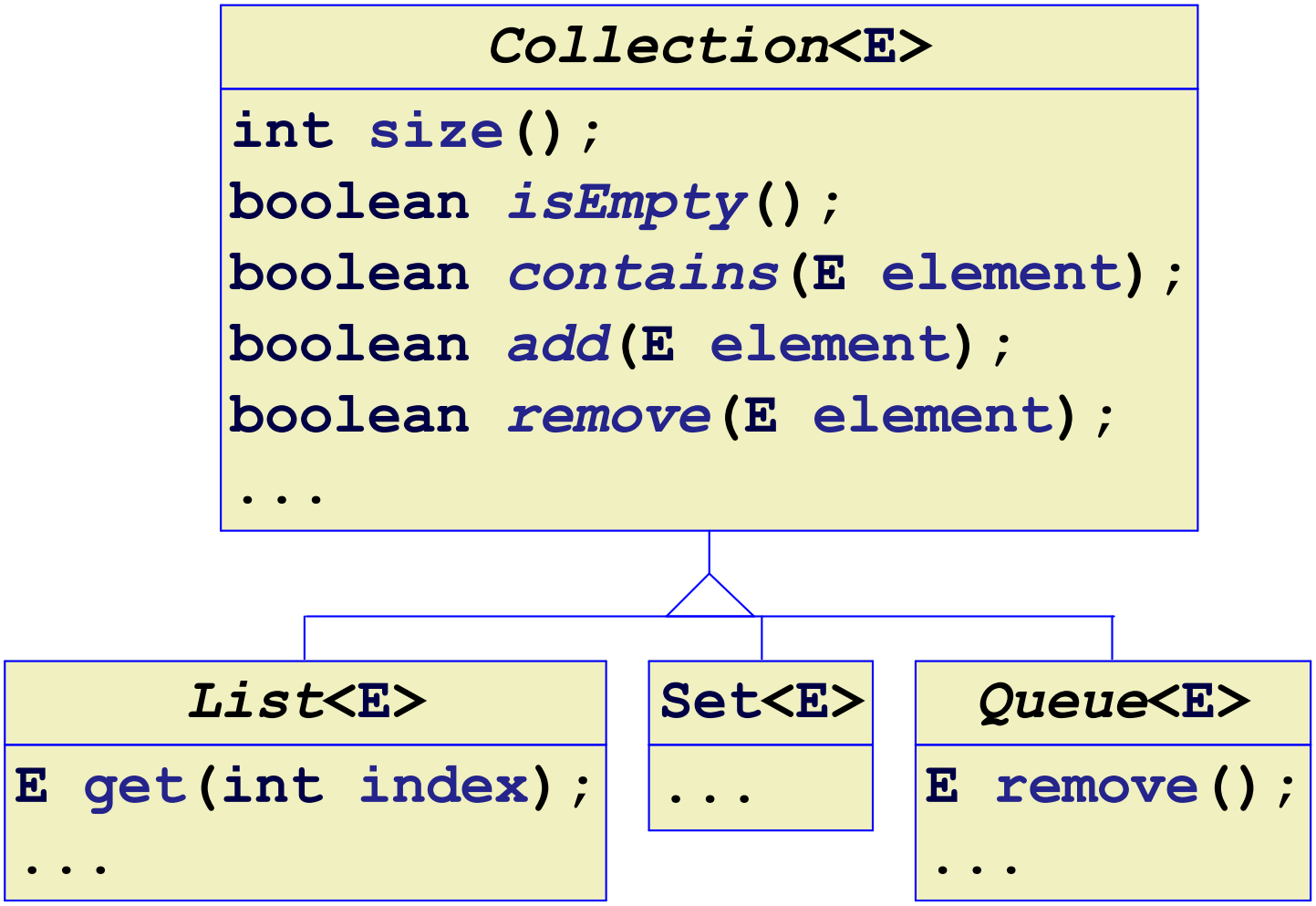
Upcasing Strings

```
t.checkExpect(concatAll(upcaseAll(ls)),
              "APPLEBANANACOCONUT");

static List<String> upcaseAll(List<String> ls) {
  List<String> uls = new LinkedList<String>();
  for (String s : ls) {
    uls.add(s.toUpperCase());
  }
  return uls;
}
```

Copy

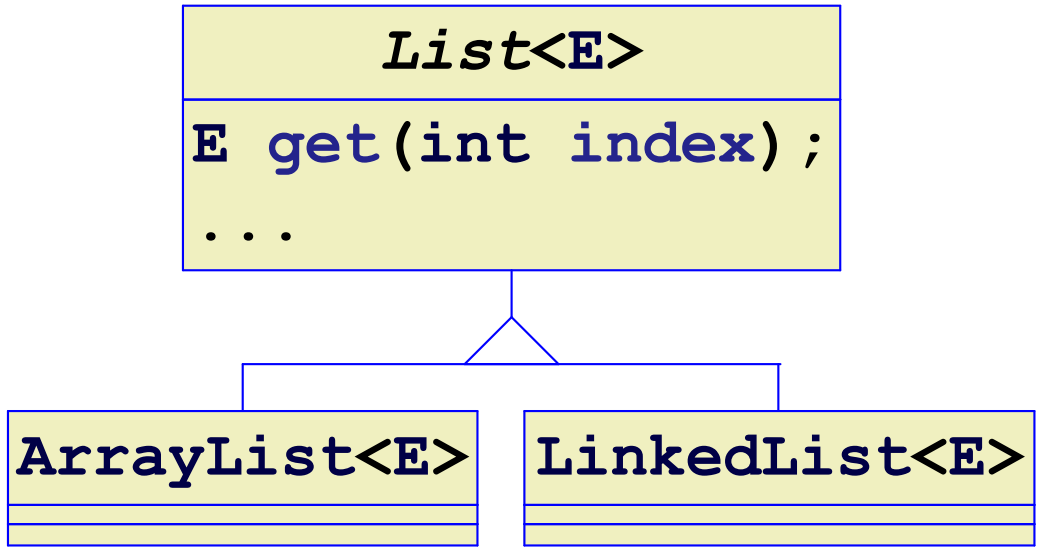
Collections



for actually works on any **Collection<E>**

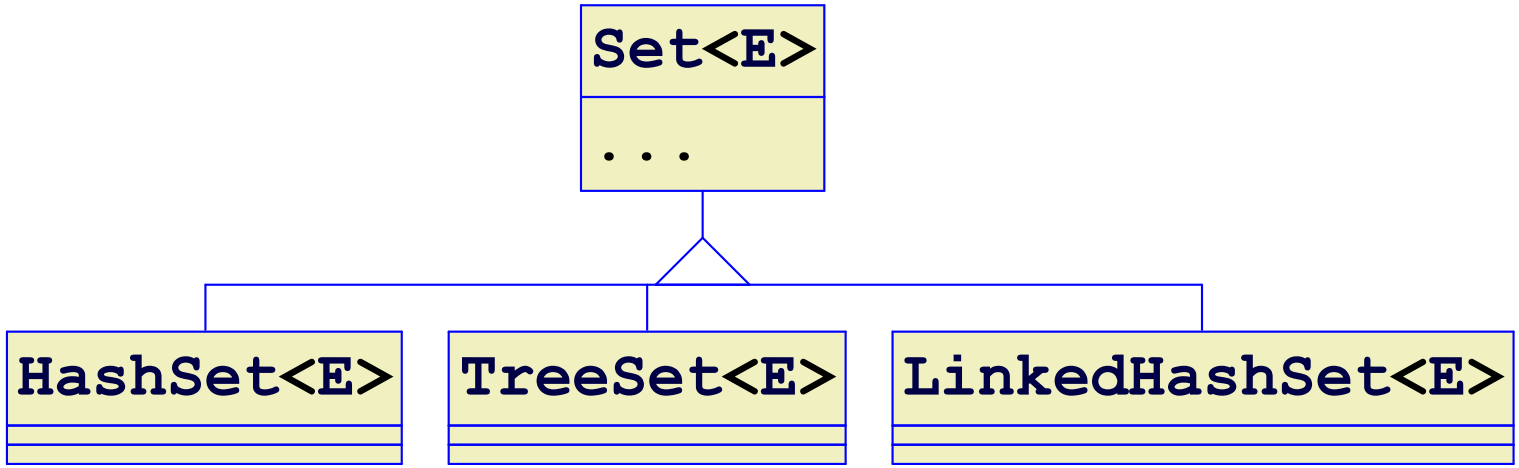
Lists

Lists: maybe duplicates, specific order



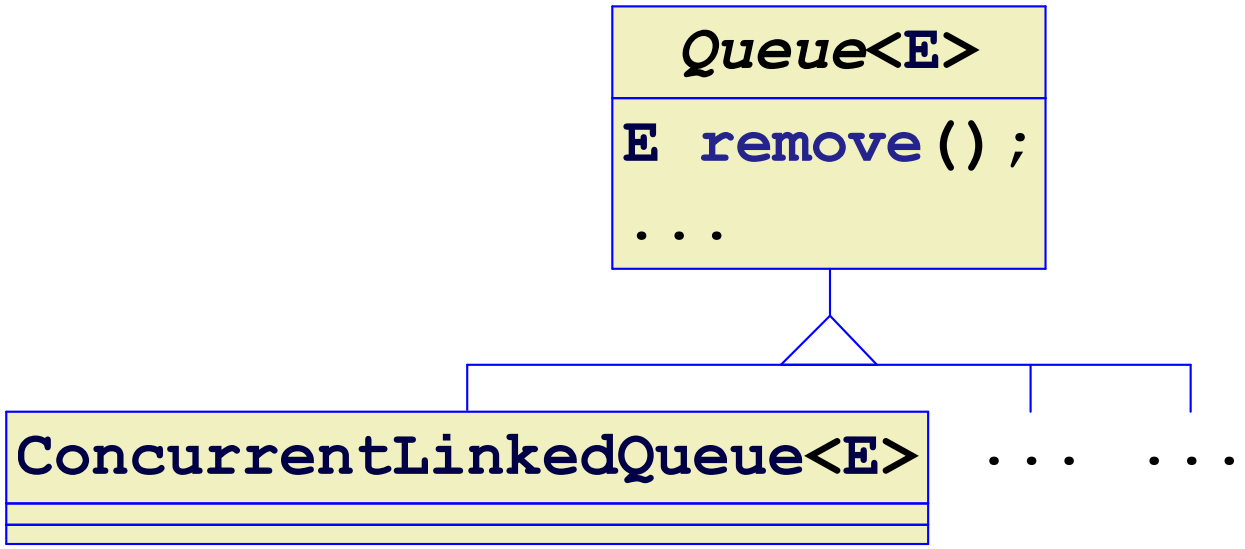
Sets

Sets: no duplicates, no order



Queues

Queues: maybe duplicates, specific order, remove from front and add to end



Sets

```
Set<String> strs = new HashSet<String>();

strs.add("apple");
strs.add("apple");
t.checkExpect(strs.size(), 1);
```

Sets

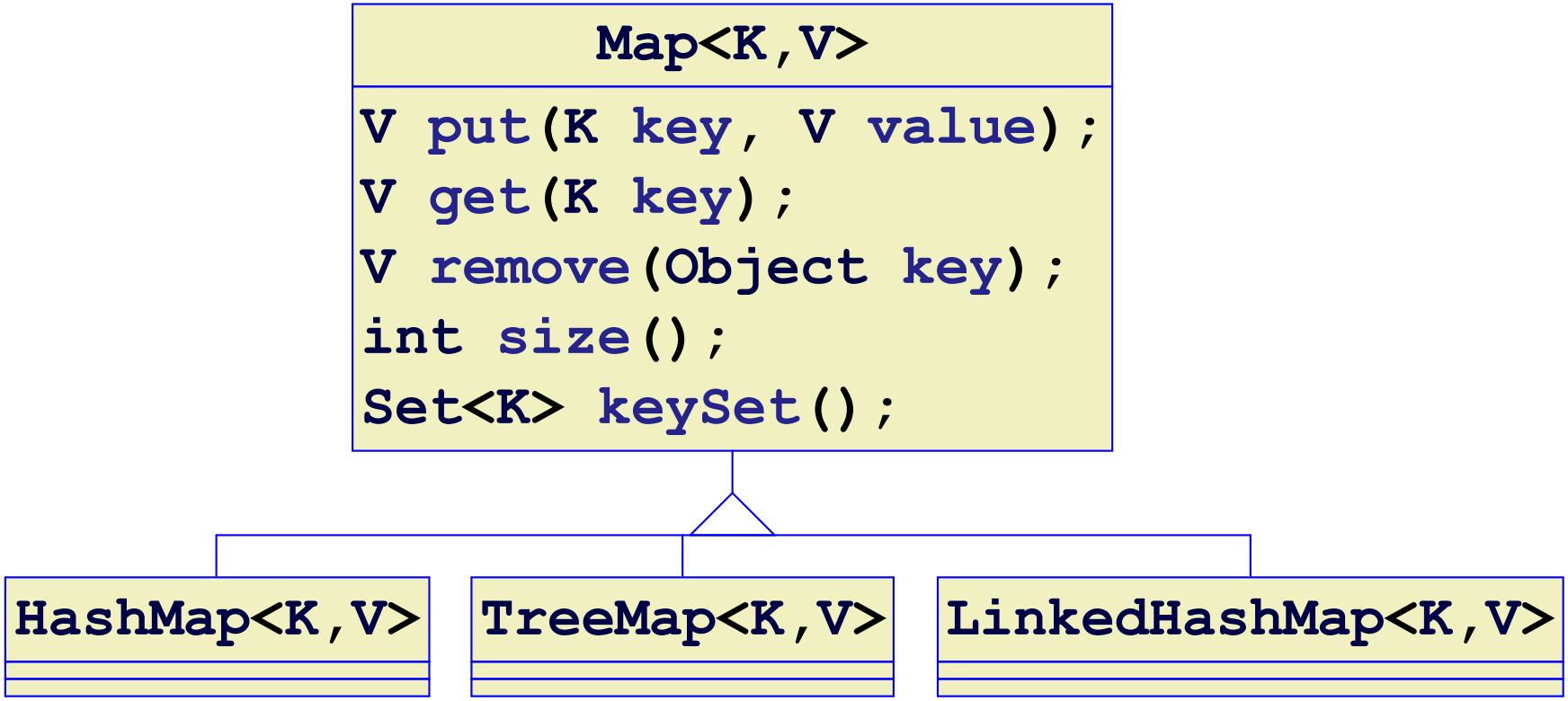
```
Set<Posn> posn = new HashSet<Posn>();

posns.add(new Posn(1, 2));
posns.add(new Posn(1, 2));
t.checkExpect(strs.size(), 2);
```

By default, objects are compared using ==

Maps

Maps: like a set, but with a ***value*** for each member ***key***



Maps

```
Map<String, Integer> phones = new HashMap<String, Integer>();

void tests(Tester t) {
  phones.put("Sam", 5551212);
  phones.put("Gil", 5557777);
  t.checkExpect(phones.get("Sam"), 5551212);
  t.checkExpect(phones.get("Gil"), 5557777);
  phones.put("Sam", 5553333);
  t.checkExpect(phones.get("Sam"), 5553333);
  t.checkExpect(addNums(phones), 5553333+5557777);
}

static int addNums(Map<String, Integer> phones) {
  int result = 0;
  for (String k : phones.keySet()) {
    result = result + phones.get(k);
  }
  return result;
}
```