Collections

Madhavan Mukund

`https://www.cmi.ac.in/~madhavan`

Programming Concepts using Java

Week 6

Built-in data types

- Most programming languages provide built-in collective data types
  - Arrays, lists, dictionaries, ...

Built-in data types

- Most programming languages provide built-in collective data types
  - Arrays, lists, dictionaries, . . .
- Java originally had many such pre-defined classes
  - `Vector`, `Stack`, `Hashtable`, `Bitset`, . . .

Built-in data types

- Most programming languages provide built-in collective data types
  - Arrays, lists, dictionaries, . . .
- Java originally had many such pre-defined classes
  - `Vector`, `Stack`, `Hashtable`, `Bitset`, . . .
- Choose the one you need

Built-in data types

- Most programming languages provide built-in collective data types
  - Arrays, lists, dictionaries, ...
- Java originally had many such pre-defined classes
  - `Vector`, `Stack`, `Hashtable`, `Bitset`, ...
- Choose the one you need
- ...but changing a choice requires multiple updates

Built-in data types

- Most programming languages provide built-in collective data types
  - Arrays, lists, dictionaries, . . .
- Java originally had many such pre-defined classes
  - `Vector`, `Stack`, `Hashtable`, `Bitset`, . . .
- Choose the one you need
- . . . but changing a choice requires multiple updates
- Instead, organize these data structures by functionality

Built-in data types

- Most programming languages provide built-in collective data types
  - Arrays, lists, dictionaries, . . .
- Java originally had many such pre-defined classes
  - `Vector`, `Stack`, `Hashtable`, `Bitset`, . . .
- Choose the one you need
- . . . but changing a choice requires multiple updates
- Instead, organize these data structures by functionality
- Create a hierarchy of abstract interfaces and concrete implementations
  - Provide a level of indirection

The Collection interface

- The Collection interface abstracts properties of grouped data
  - Arrays, lists, sets, . . .
  - But not key-value structures like dictionaries

```
public interface Collection<E>{
  boolean add(E element);
  Iterator<E> iterator();
  ...
}
```

The Collection interface

- The Collection interface abstracts properties of grouped data
  - Arrays, lists, sets, . . .
  - But not key-value structures like dictionaries
- add() — add to the collection

```
public interface Collection<E>{
  boolean add(E element);
  Iterator<E> iterator();
  ...
}
```

The Collection interface

- The Collection interface abstracts properties of grouped data
  - Arrays, lists, sets, . . .
  - But not key-value structures like dictionaries
- add() — add to the collection
- iterator() — get an object that implements Iterator interface

```
public interface Collection<E>{
  boolean add(E element);
  Iterator<E> iterator();
  ...
}

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

The Collection interface

- The Collection interface abstracts properties of grouped data
  - Arrays, lists, sets, . . .
  - But not key-value structures like dictionaries
- add() — add to the collection
- iterator() — get an object that implements Iterator interface
- Use iterator to loop through the elements

```
public interface Collection<E>{
  boolean add(E element);
  Iterator<E> iterator();
  ...
}

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

Collection<String> cstr = new ...;
Iterator<String> iter = cstr.iterator();
while (iter.hasNext()) {
  String element = iter.next();
  // do something with element
}
```

Using iterators

- Use iterator to loop through the elements

```
Collection<String> cstr = new ...;
Iterator<String> iter = cstr.iterator();
while (iter.hasNext()) {
  String element = iter.next();
  // do something with element
}
```

Using iterators

- Use iterator to loop through the elements
- Java later added "for each" loop
  - Implicitly creates an iterator and runs through it

```
Collection<String> cstr = new ...;
Iterator<String> iter = cstr.iterator();
while (iter.hasNext()) {
  String element = iter.next();
  // do something with element
}
```

```
Collection<String> cstr = new ...;
for (String element : cstr){
  // do something with element
}
```

Using iterators

- Use iterator to loop through the elements
- Java later added "for each" loop
  - Implicitly creates an iterator and runs through it
- Generic functions to operate on collections

```
Collection<String> cstr = new ...;
Iterator<String> iter = cstr.iterator();
while (iter.hasNext()) {
  String element = iter.next();
  // do something with element
}
```

```
Collection<String> cstr = new ...;
for (String element : cstr){
  // do something with element
}
```

```
public static <E> boolean
       contains(Collection<E> c, Object obj) {
  for (E element : c)
    if (element.equals(obj))
      return true;
  return false;
}
```

Using iterators

- Use iterator to loop through the elements
- Java later added “for each” loop
  - Implicitly creates an iterator and runs through it
- Generic functions to operate on collections
- How does this line work?

```
if (element.equals(obj))
```

```
Collection<String> cstr = new ...;
Iterator<String> iter = cstr.iterator();
while (iter.hasNext()) {
  String element = iter.next();
  // do something with element
}
```

```
Collection<String> cstr = new ...;
for (String element : cstr){
  // do something with element
}
```

```
public static <E> boolean
       contains(Collection<E> c, Object obj) {
  for (E element : c)
    if (element.equals(obj))
      return true;
  return false;
}
```

Using iterators

- Use iterator to loop through the elements
- Java later added "for each" loop
  - Implicitly creates an iterator and runs through it
- Generic functions to operate on collections
- How does this line work?

  ```
  if (element.equals(obj))
  ```
- Later!

```
Collection<String> cstr = new ...;
Iterator<String> iter = cstr.iterator();
while (iter.hasNext()) {
  String element = iter.next();
  // do something with element
}
```

```
Collection<String> cstr = new ...;
for (String element : cstr){
  // do something with element
}
```

```
public static <E> boolean
       contains(Collection<E> c, Object obj) {
  for (E element : c)
    if (element.equals(obj))
      return true;
  return false;
}
```

Removing elements

- Iterator also has a `remove()` method
  - Which element does it remove?

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

Removing elements

- Iterator also has a `remove()` method
    - Which element does it remove?
- The element that was last accessed using `next()`

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

Collection<String> cstr = new ...;
Iterator<String> iter = cstr.iterator();
while (iter.hasNext()) {
  String element = iter.next();
  // Delete element if it has some property
  if (property(element)) {
     iter.remove();
  }
}
```

Removing elements

- Iterator also has a `remove()` method
  - Which element does it remove?
- The element that was last accessed using `next()`
- To remove consecutive elements, must interleave a `next()`

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

```
Collection<String> cstr = new ...;
Iterator<String> iter = cstr.iterator();
...
iter.remove();
iter.remove(); // Error
```

Removing elements

- Iterator also has a `remove()` method
  - Which element does it remove?
- The element that was last accessed using `next()`
- To remove consecutive elements, must interleave a `next()`

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

```
Collection<String> cstr = new ...;
Iterator<String> iter = cstr.iterator();
...
iter.remove();
iter.next();
iter.remove();
```

Removing elements

- Iterator also has a `remove()` method
  - Which element does it remove?
- The element that was last accessed using `next()`
- To remove consecutive elements, must interleave a `next()`
- To remove the first element, need to access it first

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

```
Collection<String> cstr = new ...;
Iterator<String> iter = cstr.iterator();

// Remove first element in cstr
iter.next();
iter.remove();
```

The Collection interface — the full story

- How does this line work?

  ```
  if (element.equals(obj))
  ```

```
public static <E> boolean
       contains(Collection<E> c, Object obj) {
  for (E element : c)
    if (element.equals(obj))
      return true;
    return false;
}
```

The Collection interface — the full story

- How does this line work?

  ```
  if (element.equals(obj))
  ```

- Actually, Collection defines a much larger set of abstract methods
  - addAll(from) adds elements from a compatible collection
  - removeAll(c) removes elements present in c
  - A different remove() from the one in Iterator

```
public static <E> boolean
       contains(Collection<E> c, Object obj) {
  for (E element : c)
    if (element.equals(obj))
      return true;
    return false;
}
```

```
public interface Collection<E>{
  boolean add(E element);
  Iterator<E> iterator();
  int size() boolean isEmpty();
  boolean contains(Object obj);
  boolean containsAll(Collection<?> c);
  boolean equals(Object other);
  boolean addAll(Collection<? extends E> from);
  boolean remove(Object obj);
  boolean removeAll(Collection<?> c);
  ...
}
```

The `Collection` interface — the full story

- How does this line work?

  ```
  if (element.equals(obj))
  ```

- Actually, `Collection` defines a much larger set of abstract methods
  - `addAll(from)` adds elements from a compatible collection
  - `removeAll(c)` removes elements present in `c`
  - A different `remove()` from the one in `Iterator`
- To implement the `Collection` interface, need to implement all these methods!

```
public static <E> boolean
       contains(Collection<E> c, Object obj) {
  for (E element : c)
    if (element.equals(obj))
      return true;
    return false;
}
```

```
public interface Collection<E>{
  boolean add(E element);
  Iterator<E> iterator();
  int size() boolean isEmpty();
  boolean contains(Object obj);
  boolean containsAll(Collection<?> c);
  boolean equals(Object other);
  boolean addAll(Collection<? extends E> from);
  boolean remove(Object obj);
  boolean removeAll(Collection<?> c);
  ...
}
```

The AbsractCollection class

- To implement Collection, need to implement all these methods!

```
public interface Collection<E>{
  boolean add(E element);
  Iterator<E> iterator();
  int size() boolean isEmpty();
  boolean contains(Object obj);
  boolean containsAll(Collection<?> c);
  boolean equals(Object other);
  boolean addAll(Collection<? extends E> from);
  boolean remove(Object obj);
  boolean removeAll(Collection<?> c);
  ...
}
```

The AbsractCollection class

- To implement Collection, need to implement all these methods!
- “Correct” solution — provide default implementations in the interface

```
public interface Collection<E>{
  boolean add(E element);
  Iterator<E> iterator();
  int size() boolean isEmpty();
  boolean contains(Object obj);
  boolean containsAll(Collection<?> c);
  boolean equals(Object other);
  boolean addAll(Collection<? extends E> from);
  boolean remove(Object obj);
  boolean removeAll(Collection<?> c);
  ...
}
```

The AbsractCollection class

- To implement Collection, need to implement all these methods!
- “Correct” solution — provide default implementations in the interface
- Added to Java interfaces later!

```
public interface Collection<E>{
  boolean add(E element);
  Iterator<E> iterator();
  int size() boolean isEmpty();
  boolean contains(Object obj);
  boolean containsAll(Collection<?> c);
  boolean equals(Object other);
  boolean addAll(Collection<? extends E> from);
  boolean remove(Object obj);
  boolean removeAll(Collection<?> c);
  ...
}
```

The AbsractCollection class

- To implement Collection, need to implement all these methods!
- “Correct” solution — provide default implementations in the interface
- Added to Java interfaces later!
- Instead, AbstractCollection abstract class implements Collection

```
public abstract class AbstractCollection<E>
                   implements Collection<E> {
  ...
  public abstract Iterator<E> iterator();

  public boolean contains(Object obj) {
    for (E element : this)
      if (element.equals(obj))
        return true;
    return false;
  }
  ...
}
```

The AbsractCollection class

- To implement Collection, need to implement all these methods!
- "Correct" solution — provide default implementations in the interface
- Added to Java interfaces later!
- Instead, AbstractCollection abstract class implements Collection
- Concrete classes now extend AbstractCollection
  - Need to define iterator() based on internal representation
  - Can choose to override contains(), ...

```
public abstract class AbstractCollection<E>
                          implements Collection<E> {
  ...
  public abstract Iterator<E> iterator();

  public boolean contains(Object obj) {
    for (E element : this)
      if (element.equals(obj))
        return true;
    return false;
  }
  ...
}
```

Summary

- The `Collection` interface captures abstract properties of collections
  - Add an element, create an iterator, . . .
- Can use for each loop to avoid explicit iterator
- Write generic functions that operate on collections
- `Collection` defines many additional abstract functions, tedious if we have to implement each of them
- `AbstractCollection` provides default implementations to many functions required by `Collection`
- Concrete implementations of collections extend `AbstractCollection`