

CS61B: 2021

Lecture 11: The End of Java

- Lists and Sets
- Exceptions
- Iteration
- toString and Equals



Lists and Sets in Java

Lists

In lecture, we've build two types of lists: ALists and SLLists.

- Similar to Python lists.

```
List61B<Integer> L = new AList<>();  
L.addLast(5);  
L.addLast(10);  
L.addLast(15);  
L.print();
```

```
L = []  
L.append(3)  
L.append(4)  
L.append(5)  
print(L)
```

```
$ java ListExample  
3 4 5  
$ python list_example.py  
[3 4 5]
```

Note: We saw this in an earlier lecture, but repeated here for narrative flow.

Lists in Real Java Code

We built a list from scratch, but Java provides a built-in `List` interface and several implementations, e.g. `ArrayList`.

```
List61B<Integer> L = new AList<>();  
L.addLast(5);  
L.addLast(10);  
L.addLast(15);  
L.print();
```

```
java.util.List<Integer> L = new java.util.ArrayList<>();  
L.add(5);  
L.add(10);  
L.add(15);  
System.out.println(L);
```

Lists in Real Java Code

By including “import java.util.List” and “import java.util.ArrayList” at the top of the file, we can make our code more compact.

```
import java.util.List;
import java.util.ArrayList;

public class SimpleBuiltInListExample {
    public static void main(String[] args) {
        List<Integer> L = new ArrayList<>();
        L.add(5);
        L.add(10);
        L.add(15);
        System.out.println(L);
    }
}
```

If we import, we can use the “simple name” (ArrayList) as opposed to the longer “canonical name” (java.util.ArrayList).

Sets in Java and Python

Another handy data structure is the set.

- Stores a set of values with no duplicates. Has no sense of order.

```
Set<String> S = new HashSet<>();  
S.add("Tokyo");  
S.add("Beijing");  
S.add("Lagos");  
S.add("São Paulo");  
System.out.println(S.contains("Tokyo"));
```

```
s = set()  
s.add("Tokyo")  
s.add("Beijing")  
s.add("Lagos")  
s.add("São Paulo")  
print("Tokyo" in s)
```

```
$ java SetExample  
true  
$ python set_example.py  
True
```

ArraySet

Today we're going to write our own Set called ArraySet.

- Won't be implementing any specific interface (for now).

```
ArraySet<String> S = new ArraySet<>();  
S.add("Tokyo");  
S.add("Beijing");  
S.add("Lagos");  
S.add("São Paulo");  
System.out.println(S.contains("Tokyo"));  
System.out.println(S.size());
```

Goals

Goal 1: Create a class `ArraySet` with the following methods:

- `add(value)`: Add the value to the `ArraySet` if it is not already present.
- `contains(value)`: Checks to see if `ArraySet` contains the key.
- `size()`: Returns number of values.

Ok to ignore resizing for this exercise.

- In lecture, I'll just give away the answer, but you might find implementing it useful. See `DIY` folder in the `lectureCode` repo for starter code.

ArraySet (Basic Implementation)

```
public class ArraySet<T> {  
    private T[] items;  
    private int size;  
  
    public ArraySet() {  
        items = (T[]) new Object[100];  
        size = 0;  
    }  
    ...  
}
```

Array implementation of a Set:

- Use an array as the core data structure.
- `contains(x)`: Checks to see if `x` is in the underlying array.
- `add(x)`: Checks to see if `x` is in the underlying array, and if not, adds it.

ArraySet (Basic Implementation)

```
public boolean contains(T x) {  
    for (int i = 0; i < size; i += 1) {  
        if (items[i].equals(x)) {  
            return true;  
        }  
    }  
    return false;  
}
```

```
public void add(T x) {  
    if (!contains(x)) {  
        items[size] = x;  
        size += 1;  
    }  
}
```

Using An ArraySet

Generic type variables

```
public class ArraySet<T> {  
    private T[] items;  
    private int size;  
  
    public ArraySet() {  
        items = (T[]) new Object[100];  
        size = 0;  
    }  
    ...  
}
```

Actual type arguments

```
ArraySet<String> S = new ArraySet<>();  
S.add("horse");  
S.add("fish");
```

Exceptions

Exceptions

Basic idea:

- When something goes really wrong, break the normal flow of control.
- So far, we've only seen implicit exceptions, like the one below.

```
public static void main(String[] args) {  
    ArraySet<String> s = new ArraySet<>();  
    s.add(null);  
    s.add("horse");  
}
```

```
$ java ExceptionDemo  
Exception in thread "main"  
java.lang.NullPointerException  
    at ArraySet.contains(ArraySet.java:16)  
    at ArraySet.add(ArraySet.java:26)  
    at ArraySet.main(ArraySet.java:40)
```

Explicit Exceptions

We can also throw our own exceptions using the **throw** keyword.

- Can provide more informative message to a user.
- Can provide more information to code that “catches” the exception.

More on
“catching” at
end of the
course.

```
public void add(T x) {  
    if (x == null) {  
        throw new IllegalArgumentException("Cannot add null!");  
    }  
    ...  
}
```

```
$ java ExceptionDemo  
Exception in thread "main"  
java.lang.IllegalArgumentException: Cannot add null!  
    at ArraySet.add(ArraySet.java:27)  
    at ArraySet.main(ArraySet.java:42)
```

Explicit Exceptions

Arguably this is a bad exception.

- Our code now crashes when someone tries to add a null.
- Other fixes:
 - Ignore nulls.
 - Fix contains so that it doesn't crash if items[i] is null.

```
public void add(T x) {  
    if (x == null) {  
        throw new IllegalArgumentException("Cannot add null!");  
    }  
    ...  
}
```

Iteration

The Enhanced For Loop

Java allows us to iterate through Lists and Sets using a convenient shorthand syntax sometimes called the “foreach” or “enhanced for” loop.

```
Set<Integer> javaset = new HashSet<>();  
javaset.add(5);  
javaset.add(23);  
javaset.add(42);  
for (int i : javaset) {  
    System.out.println(i);  
}
```

The Enhanced For Loop

Java allows us to iterate through Lists and Sets using a convenient shorthand syntax sometimes called the “foreach” or “enhanced for” loop.

- This doesn't work with our ArraySet.
- Let's strip away the magic so we can build our own classes that support this.

```
ArraySet<Integer> aset = new ArraySet<>();  
aset.add(5);  
aset.add(23);  
aset.add(42);  
for (int i : aset) {  
    System.out.println(i);  
}
```

```
$ javac IterationDemo  
error: for-each not applicable to expression type  
      for (int i : S) {  
                ^  
required: array or java.lang.Iterable  
found:    ArraySet<Integer>
```

How Iteration Really Works

An alternate, uglier way to iterate through a List is to use the `iterator()` method.

Set.java:

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

```
Set<Integer> javaset =  
    new HashSet<Integer>();  
...  
for (int x : javaset) {  
    System.out.println(x);  
}
```

“Nice” iteration.

```
Set<Integer> javaset =  
    new HashSet<Integer>();  
...  
Iterator<Integer> seer  
    = javaset.iterator();  
  
while (seer.hasNext()) {  
    System.out.println(seer.next());  
}
```

“Ugly” iteration.

How Iterators Work

An alternate, uglier way to iterate through a List is to use the `iterator()` method.

javaset:

5	23	42
---	----	----

```
Iterator<Integer> seer
    = javaset.iterator();
while (seer.hasNext()) {
    System.out.println(seer.next());
}
```

How Iterators Work

An alternate, uglier way to iterate through a List is to use the `iterator()` method.



javaset:

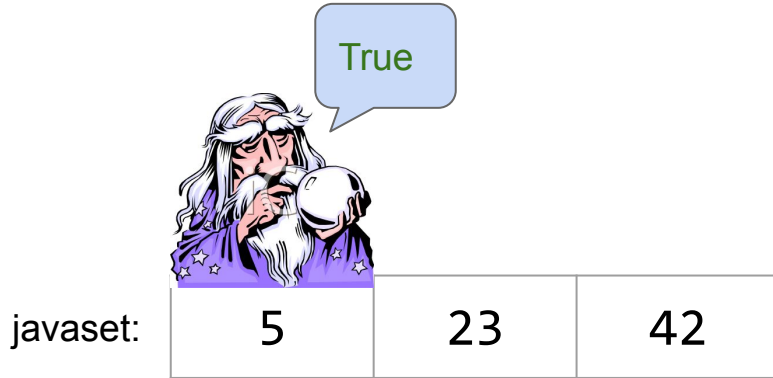
5	23	42
---	----	----

```
$ java IteratorDemo.java
```

```
→ Iterator<Integer> seer  
    = javaset.iterator();  
while (seer.hasNext()) {  
    System.out.println(seer.next());  
}
```

How Iterators Work

An alternate, uglier way to iterate through a List is to use the `iterator()` method.



```
$ java IteratorDemo.java
```

```
Iterator<Integer> seer  
    = javaset.iterator();  
→ while (seer.hasNext()) {  
    System.out.println(seer.next());  
}
```

How Iterators Work

An alternate, uglier way to iterate through a List is to use the `iterator()` method.



```
$ java IteratorDemo.java  
5
```

```
Iterator<Integer> seer  
    = javaset.iterator();  
while (seer.hasNext()) {  
    System.out.println(seer.next());  
}
```

How Iterators Work

An alternate, uglier way to iterate through a List is to use the `iterator()` method.



```
$ java IteratorDemo.java  
5
```

```
Iterator<Integer> seer  
    = javaset.iterator();  
→ while (seer.hasNext()) {  
    System.out.println(seer.next());  
}
```


How Iterators Work

An alternate, uglier way to iterate through a List is to use the `iterator()` method.

javaset:

5	23	42
---	----	----

23



```
$ java IteratorDemo.java  
5  
23
```

```
Iterator<Integer> seer  
    = javaset.iterator();  
while (seer.hasNext()) {  
    System.out.println(seer.next());  
}
```



How Iterators Work

An alternate, uglier way to iterate through a List is to use the `iterator()` method.

javaset:

5	23	42
---	----	----

True



```
$ java IteratorDemo.java  
5  
23
```

```
Iterator<Integer> seer  
    = javaset.iterator();  
→ while (seer.hasNext()) {  
    System.out.println(seer.next());  
}
```

How Iterators Work

An alternate, uglier way to iterate through a List is to use the `iterator()` method.

javaset:

5	23	42
---	----	----

42



```
$ java IteratorDemo.java  
5  
23  
42
```

```
Iterator<Integer> seer  
    = javaset.iterator();  
while (seer.hasNext()) {  
→   System.out.println(seer.next());  
}
```

How Iterators Work

An alternate, uglier way to iterate through a List is to use the `iterator()` method.

javaset:

5	23	42
---	----	----

False



```
$ java IteratorDemo.java  
5  
23  
42
```

```
Iterator<Integer> seer  
    = javaset.iterator();  
→ while (seer.hasNext()) {  
    System.out.println(seer.next());  
}
```

The Secret of the Enhanced For Loop yellkey.com/art

The secret: The code on the left is just shorthand for the code on the right. For code on right to compile, which checks does the compiler need to do?

- A. Does the Set interface have an iterator() method?
- B. Does the Set interface have next/hasNext() methods?
- C. Does the Iterator interface have an iterator method?
- D. Does the Iterator interface have next/hasNext() methods?

```
Set<Integer> javaset = new HashSet<Integer>();
```

```
for (int x : javaset) {  
    System.out.println(x);  
}
```

```
Iterator<Integer> seer  
    = javaset.iterator();  
  
while (seer.hasNext()) {  
    System.out.println(seer.next());  
}
```

The Secret of the Enhanced For Loop

The secret: The code on the left is just shorthand for the code on the right. For code on right to compile, which checks does the compiler need to do?

- A. Does the Set interface have an iterator() method?
- B. Does the Set interface have next/hasNext() methods?
- C. Does the Iterator interface have an iterator method?
- D. Does the Iterator interface have next/hasNext() methods?

```
Set<Integer> javaset = new HashSet<Integer>();
```

```
for (int x : javaset) {  
    System.out.println(x);  
}
```

```
Iterator<Integer> seer  
    = javaset.iterator();  
  
while (seer.hasNext()) {  
    System.out.println(seer.next());  
}
```

Supporting Ugly Iteration in ArraySets

To support ugly iteration:

- Add an `iterator()` method to `ArraySet` that returns an `Iterator<T>`.
- The `Iterator<T>` that we return should have a useful `hasNext()` and `next()` method.

`Iterator<T>`

```
public interface Iterator<T> {  
    boolean hasNext();  
    T next();  
}
```

```
Iterator<Integer> aseer  
    = aset.iterator();  
  
while (aseer.hasNext()) {  
    System.out.println(aseer.next());  
}
```

Completed ArraySet iterator Method

To support ugly iteration:

- Add an `iterator()` method to `ArraySet` that returns an `Iterator<T>`.
- The `Iterator<T>` that we return should have a useful `hasNext()` and `next()` method.

```
private class ArraySetIterator implements Iterator<T> {  
    private int wizPos;  
    public ArraySetIterator() { wizPos = 0; }  
    public boolean hasNext() { return wizPos < size; }  
    public T next() {  
        T returnItem = items[wizPos];  
        wizPos += 1;  
        return returnItem;  
    }  
}
```

```
public Iterator<T> iterator() {  
    return new ArraySetIterator();  
}
```


The Enhanced For Loop

Our code now supports “ugly” iteration, but enhanced for loop still doesn’t work.

The problem: Java isn’t smart enough to realize that our `ArraySet` has an `iterator()` method.

- Luckily there’s an interface for that.

```
ArraySet<Integer> aset = new ArraySet<>();  
aset.add(5);  
aset.add(23);  
aset.add(42);  
for (int i : aset) {  
    ...  
}
```

```
$ javac IterationDemo  
error: for-each not applicable to expression type  
      for (int i : aset) {  
                ^  
required: array or java.lang.Iterable  
found:    ArraySet<Integer>
```

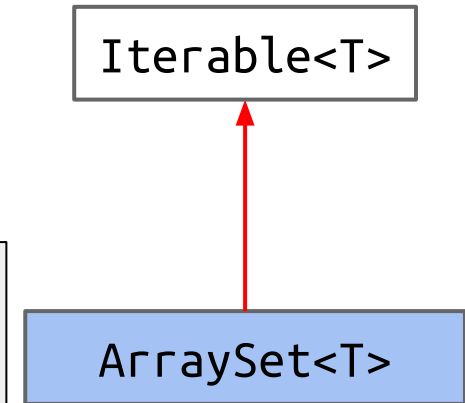
For-each Iteration And ArraySets

To support the enhanced for loop, we need to make `ArraySet` implement the `Iterable` interface.

- There are also some default methods in `Iterable`, not shown.

```
public interface Iterable<T> {  
    Iterator<T> iterator();  
}
```

```
public class ArraySet<T> implements Iterable<T> {  
    ...  
    public Iterator<T> iterator() { ... }  
}
```



The Iterable Interface

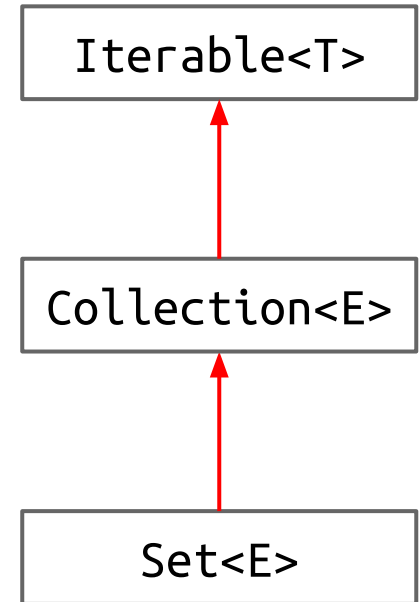
By the way, this is how Set works as well.

- Source code for Iterable: [Link](#), Set: [Link](#), Collection: [Link](#).

```
public interface Iterable<T> {  
    Iterator<T> iterator(); ...  
}
```

```
public interface Collection<E> extends Iterable<E> {  
    public Iterator<E> iterator();  
}
```

```
public interface Set<E> extends Collection<E> {  
    public Iterator<E> iterator();  
}
```



Iteration Summary

To support the enhanced for loop:

- Add an `iterator()` method to your class that returns an `Iterator<T>`.
- The `Iterator<T>` returned should have a useful `hasNext()` and `next()` method.
- Add `implements Iterable<T>` to the line defining your class.

Part 5 of HW1 gives you a chance to try this out yourself.

Object Methods: Equals and toString()

Objects

All classes are hyponyms of Object.

- **String toString()**
- **boolean equals(Object obj)**
- **Class<?> getClass()**
- **int hashCode()**
- **protected Object clone()**
- **protected void finalize()**
- **void notify()**
- **void notifyAll()**
- **void wait()**
- **void wait(long timeout)**
- **void wait(long timeout, int nanos)**

Very important, but won't discuss for a few weeks.

Won't discuss or use in 61B.

toString()

The `toString()` method provides a string representation of an object.

- `System.out.println(Object x)` calls `x.toString()`
 - If you're curious: [println](#) calls [String.valueOf](#) which calls `toString`

```
Set<Integer> javaset = new HashSet<>();  
javaset.add(5);  
javaset.add(23);  
javaset.add(42);  
  
System.out.println(javaset);
```

```
$ java JavaSetPrintDemo  
[5, 23, 42]
```

toString()

The `toString()` method provides a string representation of an object.

- `System.out.println(Object x)` calls `x.toString()`
- The [implementation of toString\(\) in Object](#) is the the name of the class, then an `@` sign, then the memory location of the object.
 - See 61C for what the “memory location” really means.

```
ArraySet<Integer> aset = new ArraySet<>();  
aset.add(5);  
aset.add(23);  
aset.add(42);  
  
System.out.println(aset);
```

```
$ java ArraySetPrintDemo  
ArraySet@75412c2f
```


ArraySet toString

Let's try implementing toString for ArraySet.

ArraySet toString

One approach is shown below.

- Warning: This code is slow. Intuition: Adding even a single character to a string creates an entirely new string. Will discuss why at end of course.

```
@Override
public String toString() {
    String returnString = "{";
    for (int i = 0; i < size; i += 1) {
        returnString += keys[i];
        returnString += ", ";
    }
    returnString += "}";
    return returnString;
}
```

Spoiler: It's
because
Strings are
"immutable".

ArraySet toString

Much faster approach is shown below.

- Intuition: Append operation for a StringBuilder is fast.

```
@Override
public String toString() {
    StringBuilder returnSB = new StringBuilder("{}");
    for (int i = 0; i < size; i += 1) {
        returnSB.append(items[i]);
        returnSB.append(", ");
    }
    returnSB.append("}");
    return returnSB.toString();
}
```

Objects

All classes are hyponyms of Object.

- `String toString()`
- **`boolean equals(Object obj)`**
- `Class<?> getClass()`
- `int hashCode()`
- `protected Object clone()`
- `protected void finalize()`
- `void notify()`
- `void notifyAll()`
- `void wait()`
- `void wait(long timeout)`
- `void wait(long timeout, int nanos)`

Very important, but won't discuss for a few weeks.

Won't discuss or use in 61B.

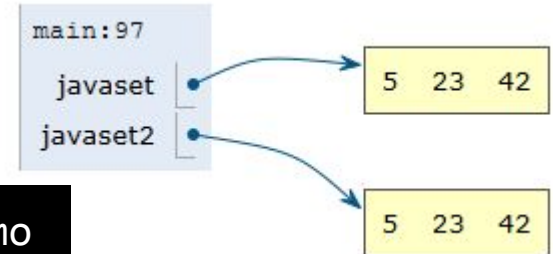
Equals vs. ==

As mentioned in an offhand manner previously, `==` and `.equals()` behave differently.

- `==` compares the bits. For references, `==` means “referencing the same object.”

```
Set<Integer> javaset = Set.of(5, 23, 42);  
Set<Integer> javaset2 = Set.of(5, 23, 42);  
System.out.println(javaset == javaset2);
```

```
$ java EqualsDemo  
False
```



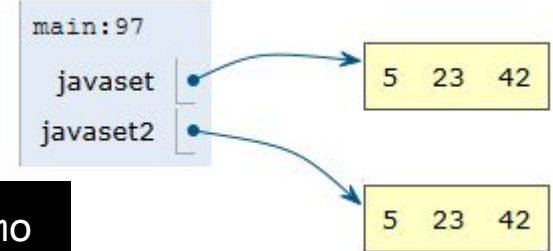
Equals vs. ==

As mentioned in an offhand manner previously, `==` and `.equals()` behave differently.

- `==` compares the bits. For references, `==` means “referencing the same object.”

```
Set<Integer> javaset = Set.of(5, 23, 42);  
Set<Integer> javaset2 = Set.of(5, 23, 42);  
System.out.println(javaset.equals(javaset2));
```

```
$ java EqualsDemo  
True
```



To test equality in the sense we usually mean it, use:

- `.equals` for classes. Requires writing a `.equals` method for your classes.
 - [Default implementation](#) of `.equals` uses `==` (probably not what you want).
- BTW: Use `Arrays.equals` or `Arrays.deepEquals` for arrays.

The Default Implementation of Equals

```
ArraySet<Integer> aset = new ArraySet<>();  
aset.add(5);  
aset.add(23);  
aset.add(42);
```

```
System.out.println(aset);
```

```
ArraySet<Integer> aset2 = new ArraySet<>();  
aset2.add(5);  
aset2.add(23);  
aset2.add(42);
```

```
System.out.println(aset.equals(aset2));
```

Returns false because the default implementation of equals just uses ==.

```
$ java EqualsDemo  
False
```

ArraySet equals

Let's try implementing equals for ArraySet.

ArraySet equals

The implementation below is a good start, but fails if o is null or another class.

```
@Override
public boolean equals(Object o) {
    ArraySet<T> other = (ArraySet<T>) o;
    if (this.size() != other.size()) { return false; }
    for (T item : this) {
        if (!other.contains(item)) {
            return false;
        }
    }
    return true;
}
```

ArraySet equals

The implementation below is much better, but we can speed things up.

```
@Override
public boolean equals(Object o) {
    if (o == null) { return false; }
    if (this.getClass() != o.getClass()) { return false; }
    ArraySet<T> other = (ArraySet<T>) o;
    if (this.size() != other.size()) { return false; }
    for (T item : this) {
        if (!other.contains(item)) {
            return false;
        }
    }
    return true;
}
```

ArraySet equals

The code below is pretty close to what a standard equals method looks like.

```
@Override
public boolean equals(Object o) {
    if (o == null) { return false; }
    if (this == o) { return true; } // optimization
    if (this.getClass() != o.getClass()) { return false; }
    ArraySet<T> other = (ArraySet<T>) o;
    if (this.size() != other.size()) { return false; }
    for (T item : this) {
        if (!other.contains(item)) {
            return false;
        }
    }
    return true;
}
```

Summary

We built our own Array based Set implementation.

To make it more industrial strength we:

- Added an exception if a user tried to add null to the set.
 - There are other ways to deal with nulls. Our choice was arguably bad.
- Added support for “ugly” then “nice” iteration.
 - Ugly iteration: Creating a subclass with next and hasNext methods.
 - Nice iteration: Declaring that ArraySet implements Iterable.
- Added a toString() method.
 - Beware of String concatenation.
- Added an equals(Object) method.
 - Make sure to deal with null and non-ArraySet arguments!
 - Used getClass to check the class of the passed object. Use sparingly.

Even Better toString and ArraySet.of (Extra)

Citations

Seer:

http://www.clipartoday.com/_thumbs/022/Fantasy/astrology_crystal_190660_tnb.png

Edge of the world:

<https://www.flickr.com/photos/tochis/2947187311>

.....