University of Southampton

Collections and Iterators

(slides adapted from D. Millard)

Thai Son Hoang

ECS, University of Southampton, U.K.

COMP1202
23th October 2023

Recap

University of Southampton

- Looping
  - **while**
  - **do** ... **while**
  - **for** loop
  - **for** `each` loop
- Arrays
  - Iterating through arrays using **for** `each` loop

Objectives

University of Southampton

- Arrays vs. ArrayLists
  - Declaration
  - Insertion
  - Access
  - Removal
- A Brief Introduction to Generics
  - Autoboxing and unboxing
- Iterator objects

Readings

- Chapter 4.10 of Barnes and Kölling [2016]

Outline

Problems with Arrays

- They don't change size
- It's a pain adding new elements
  if you don't know how many are there already
- You have to use indexes
- `ArrayIndexOutOfBoundsException`

ArrayList to the rescue!

- Arrays are built into the Java language (a bit like primitives)
- But Java also have a library of helpful classes
  that you can use for for free
- These are not part of the language,
  but are included with every JVM
- `ArrayList` is one of these library classes

Arrays vs. ArrayLists (1/2)

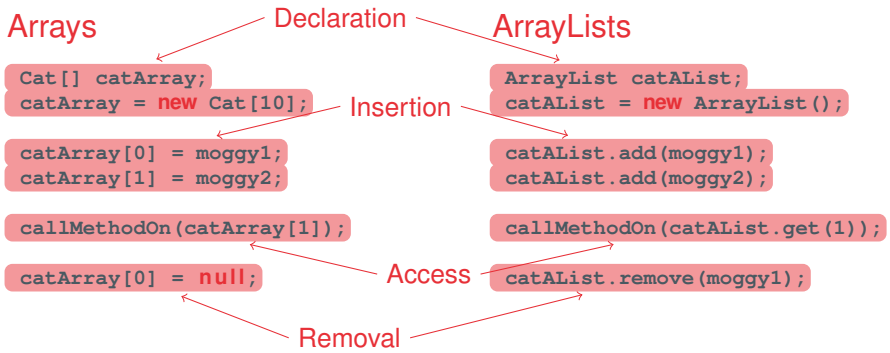
Arrays

- They don't change size
- It's a pain adding new elements if you don't know how many are there already
- You have to use indexes
- `ArrayIndexOutOfBoundsException`

ArrayLists

- Changes size as you add elements
- `ArrayList` has an `add()` method and takes care of its size itself
- You can use indexes if you want (but don't have to)
- Still thrown by `ArrayList`. Hey, it's a fact of life, okay?

Arrays vs. ArrayLists (2/2)



Advantages of ArrayLists

- Arrays are useful for simple small tasks
- ArrayLists are better for more complex tasks
  - They grow and shrink when you add and remove things (arrays are fixed size)
  - They have many useful methods ...
- Check out the API:
  - Application Programming Interface
  - https://docs.oracle.com/en/java/javase/17/docs/api/
  - type “java api” into google

ArrayList - Method Summary

Modifier and Type	Method	Description
void	add(int index, E element)	Inserts the specified element at the specified position in this list.
boolean	add(E e)	Appends the specified element to the end of this list.
boolean	addAll(int index, Collection<? extends E> c)	Inserts all of the elements in the specified collection into this list, starting at the specified position.
boolean	addAll(Collection<? extends E> c)	Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's Iterator.
void	clear()	Removes all of the elements from this list.
Object	clone()	Returns a shallow copy of this ArrayList instance.
boolean	contains(Object o)	Returns true if this list contains the specified element.
void	ensureCapacity(int minCapacity)	Increases the capacity of this ArrayList instance, if necessary, to ensure that it can hold at least the number of elements specified by the minimum capacity argument.
void	forEach(Consumer<? super E> action)	Performs the given action for each element of the Iterable until all elements have been processed or the action throws an exception.
E	get(int index)	Returns the element at the specified position in this list.
int	indexOf(Object o)	Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
boolean	isEmpty()	Returns true if this list contains no elements.
Iterator<E>	iterator()	Returns an iterator over the elements in this list in proper sequence.
int	lastIndexOf(Object o)	Returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.
ListIterator<E>	listIterator()	Returns a list iterator over the elements in this list (in proper sequence).
ListIterator<E>	listIterator(int index)	Returns a list iterator over the elements in this list (in proper sequence), starting at the specified position in the list.
E	remove(int index)	Removes the element at the specified position in this list.
boolean	remove(Object o)	Removes the first occurrence of the specified element from this list, if it is present.
boolean	removeAll(Collection<?> c)	Removes from this list all of its elements that are contained in the specified collection.
boolean	removeIf(Predicate<? super E> filter)	Removes all of the elements of this collection that satisfy the given predicate.
protected void	removeRange(int fromIndex, int toIndex)	Removes from this list all of the elements whose index is between fromIndex, inclusive, and toIndex, exclusive.
boolean	retainAll(Collection<?> c)	Retains only the elements in this list that are contained in the specified collection.
E	set(int index, E element)	Replaces the element at the specified position in this list with the specified element.
int	size()	Returns the number of elements in this list.
Spliterator<E>	spliterator()	Creates a *late-binding* and *fail-fast* Spliterator over the elements in this list.
List<E>	subList(int fromIndex, int toIndex)	Returns a view of the portion of this list between the specified fromIndex, inclusive, and toIndex, exclusive.
Object[]	toArray()	Returns an array containing all of the elements in this list in proper sequence (from first to last element).
<T> T[]	toArray(T[] a)	Returns an array containing all of the elements in this list in proper sequence (from first to last element); the runtime type of the returned array is that of the specified array.
void	trimToSize()	Trims the capacity of this ArrayList instance to be the list's current size.

Example 1 (1/2)

```
ArrayList numArrayList = new ArrayList();

for (int i = 0; i < 9; i++){
    numArrayList.add(i);
}
System.out.println("Value at 4 is " +
    numArrayList.get(4));
```

Vevox (140-996-816)?

What number will be printed by the final statement?

Example 1 (2/2)

```
ArrayList numArrayList = new ArrayList();

for (int i = 0; i < 9; i++){
    numArrayList.add(i);
}
System.out.println("Value at 4 is " +
   numArrayList.get(4));
```

Answer

- The loop creates an ArrayList of numbers 0, 1, ..., 8.
- After the loop, the value at index 4 is 4.

Example 2 (1/2)

```
ArrayList numArrayList = new ArrayList();

for (int i = 0; i < 9; i++){
    numArrayList.add(10 - i);
}
System.out.println("Value at 4 is " +
   numArrayList.get(4));
```

Vevox (140-996-816)?

What number will be printed by the final statement?

Example 2 (2/2)

```
ArrayList numArrayList = new ArrayList();

for (int i = 0; i < 9; i++){
    numArrayList.add(10 - i);
}
System.out.println("Value at 4 is " +
   numArrayList.get(4));
```

Answer

- The loop creates an ArrayList of numbers 10, 9, ..., 2.
- After the loop, the value at index 4 is 6.

Example 3 (1/2)

```
ArrayList numArrayList = new ArrayList();

for (int i = 0; i < 9; i++){
    numArrayList.add(10 - i);
}

for (int i = 8; i >= 0; i--){
    System.out.println("Value is " +
       numArrayList.get(i));
}
```

Vevox (140-996-816)?

What number will be printed by the second loop?

Example 3 (2/2)

```
ArrayList numArrayList = new ArrayList();

for (int i = 0; i < 9; i++){
    numArrayList.add(10 - i);
}

for (int i = 8; i >= 0; i--){
    System.out.println("Value is " +
       numArrayList.get(i));
}
```

Answer

- The loop creates an ArrayList of numbers 10, 9, ..., 2.
- The second loop traverses in the reverse order.
- Hence it produces the sequence 2, 3, ..., 10.

Self-Checked

ArrayList

Blackboard Tests for ArrayList

Outline

Spot the Problem . . .

```
ArrayList kennel = new ArrayList();

kennel.add(new Dog("Rover"));
kennel.add(new Dog("Fido"));
kennel.add(new Dog("Patch"));
kennel.add(new Cat("Mr Tiddles"));


for(int i = 0; i < kennel.size(); i++)
    kennel.get(i).bark();
}
```

ArrayLists store objects of any type

Which means we can mix up the types of objects in the ArrayList

Which may cause problems later if we make assumptions about what is in there!

In fact this code will not compile, because Java does not know what is in the ArrayList, and therefore will not let you call **bark** on it

Solving the Problem . . . (1/2)

University of Southampton

```
ArrayList kennel = new ArrayList();

kennel.add(new Dog("Rover"));
kennel.add(new Dog("Fido"));
kennel.add(new Dog("Patch"));
kennel.add(new Cat("Mr Tiddles"));


for(int i = 0; i < kennel.size(); i++) {
    if (kennel.get(i) instanceof Dog) {
        Dog d = (Dog) kennel.get(i);
        d.bark();
    }
}
```

One option is to test what is in there using **instanceof**, and if it's a Dog we can tell the compiler.
This is called **typecasting**

Makes my inner software engineer cringe!
instanceof is a tool of last resort. If you've had to use it it probably means you're program is not designed particularly well.

Solving the Problem . . . (2/2)

```
ArrayList<Dog> kennel = new ArrayList<Dog>();

kennel.add(new Dog("Rover"));
kennel.add(new Dog("Fido"));
kennel.add(new Dog("Patch"));
kennel.add(new Cat("Mr Tiddles"));


for(int i = 0; i < kennel.size(); i++) {
    kennel.get(i).bark();
}
```

This is easily done because ArrayList uses a mechanism called **generics**.
We can specify the type allowed when we create the ArrayList.

Now Java will only allow us to add things of type **Dog**. So this line will force a **compile time** error

A Note About Primitives

University of Southampton

```
ArrayList<Integer> numStore;
numStore = new ArrayList<Integer>();

numStore.add(new Integer(3));
numStore.add(new Integer(5));
numStore.add(new Integer(2));

int total = 0;

for(int i = 0; i < numStore.size(); i++) {
    total = total + numStore.get(i).intValue();
}

System.out.println("Total is " + total);
```

ArrayLists (and other collections in the API) can only store objects.
This means that when you want to store primitives you need to use wrapper objects.
This is a pain :-(

A Note About Primitives

University of Southampton

```
ArrayList<Integer> numStore;
numStore = new ArrayList<Integer>();

numStore.add(3);
numStore.add(5);
numStore.add(2);

int total = 0;

for(int i = 0; i < numStore.size(); i++) {
    total = total + numStore.get(i);
}

System.out.println("Total is " + total);
```

ArrayLists (and other collections in the API) can only store objects.
This means that when you want to store primitives you need to use wrapper objects.
This is a pain :-(

Java 5 introduced **autoboxing**, a process where primitives are automatically cast to a wrapper where necessary.

And **unboxing**, where they can be cast back again too

Self-Checked

Genericity

Blackboard Tests for Genericity

Outline

Design Patterns

- In Programming a neat and elegant way of solving a problem is sometimes called a design pattern
- The Java API uses a number of well-known design patterns
- Including the use of iterators to help you iterate over a collection

Back at the Kennel . . .

```
ArrayList<Dog> kennel = new ArrayList<Dog

kennel.add(new Dog("Rover"));
kennel.add(new Dog("Fido"));
kennel.add(new Dog("Patch"));

for(int i = 0; i < kennel.size(); i++) {
    kennel.get(i).bark();
}

Iterator<Dog> it = kennel.iterator();

while(it.hasNext()) {
    it.next().bark();
}
```

In our kennel example we used a for loop to iterate over the array

We could instead use an iterator object.
Iterators are generic classes (like the ArrayList) and track our progress through a collection.
We can use **hasNext()** to see if there are more elements
And **next()** to get the next element (the iterator will automatically move to the next element).

Why are Iterators a useful pattern?

University of Southampton

```
ArrayList<Dog> kennel = new ArrayList<Dog>();

kennel.add(new Dog("Rover"));
kennel.add(new Dog("Fido"));
kennel.add(new Dog("Patch"));


for(int i = 0; i < kennel.size(); i++) {
    kennel.get(i).bark();
}

Iterator<Dog> it = kennel.iterator();

while(it.hasNext()) {
    it.next().bark();
}
```

1) They are neater, and neat code is easier to read and understand

2) They decouple the loop from the collection (notice that in the loop we do not reference the ArrayList at all)
This means we could pass the iterator to a method – and that method does not even need to know what the collection is!

Why are Iterators a useful pattern?

University of Southampton

```
public void makeThemBark(Iterator<Dog> it) {

    while (it.hasNext()) {
        it.next().bark();
    }

}
```

1) They are neater, and neat code is easier to read and understand

2) They decouple the loop from the collection (notice that in the loop we do not reference the Arraylist at all)
This means we could pass the iterator to a method – and that method does not even need to know what the collection is!

Example 4 (1/2)

University of Southampton

```
ArrayList numArrayList = new ArrayList();

for (int i = 0; i < 9; i++){
    numArrayList.add(10 - i);
}

Iterator it = numArrayList.iterator();
while (it.hasNext()) {
    System.out.println("Value is " + it.next());
}
```

Vevox (140-996-816)?

What number will be printed by the second loop?

Example 4 (2/2)

University of Southampton

```
ArrayList numArrayList = new ArrayList();

for (int i = 0; i < 9; i++){
    numArrayList.add(10 - i);
}

Iterator it = numArrayList.iterator();
while (it.hasNext()) {
    System.out.println("Value is " + it.next());
}
```

Answer

- The loop creates an ArrayList of numbers 10, 9, ..., 2.
- The second loop traverses in the array from the beginning.
- Hence it produces the sequence 10, 9, ..., 2.

Self-Checked

Iterators

University of Southampton

Blackboard Tests for Iterators

Outline

University of Southampton

Summary

University of Southampton

- Arrays vs. ArrayLists
  - Declaration
  - Insertion
  - Access
  - Removal
- A Brief Introduction to Generics
  - Autoboxing and unboxing
- Iterator objects

YOUR QUESTIONS

References I

- David J. Barnes and Michael Kölling. *Objects First with Java: A Practical Introduction using BlueJ*.
  Pearson, sixth edition edition, 2016 (Chapter 4.10)