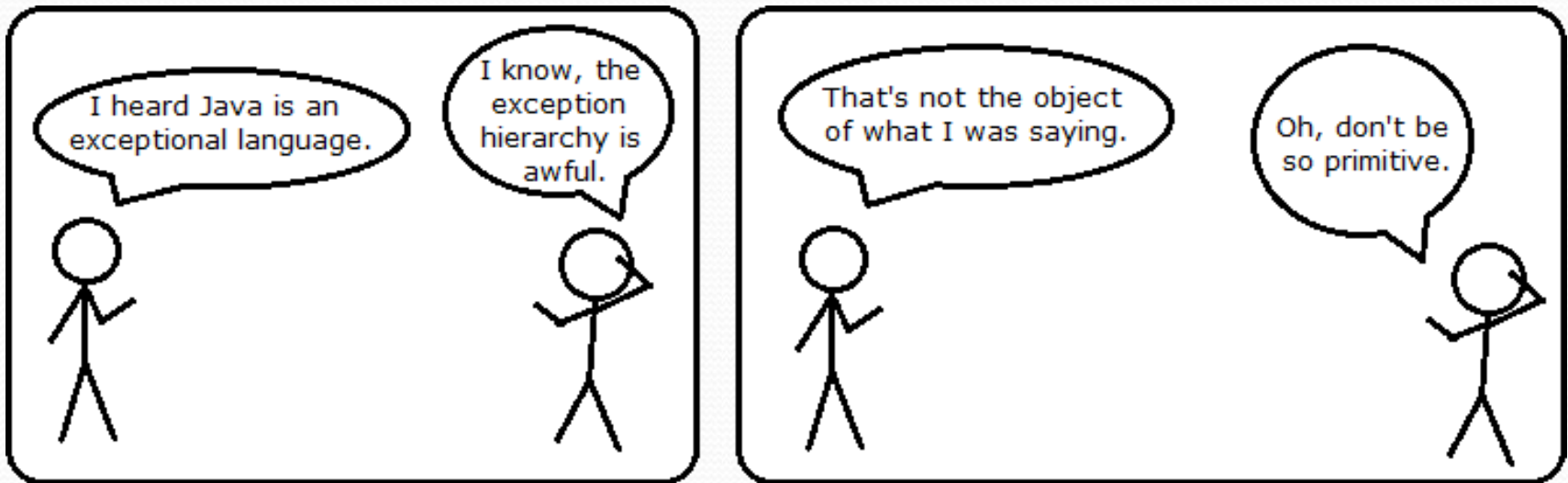


CS 142

Lecture 6: searching; using `ArrayLists`

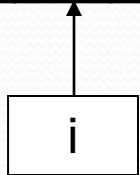


Thanks to Marty Stepp and Stuart Reges for parts of these slides

Sequential search

- **sequential search:** Locates a target value in an array / list by examining each element from start to finish. Use to write `indexOf`.
 - How many elements will it need to examine?
 - Example: Searching the array below for the value **42**:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	7	10	15	20	22	25	30	36	42	50	56	68	85	92	103



- The array is sorted. How can we take advantage of this?

Binary search

- **binary search:** Locates a target value in a *sorted* array or list by successively eliminating half of the array from consideration.
 - How many elements will it need to examine?
 - Example: Searching the array below for the value **42**:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	7	10	15	20	22	25	30	36	42	50	56	68	85	92	103

Diagram illustrating a binary search on a sorted array. The array is shown with indices 0 to 16 and corresponding values. The value 42 is highlighted in yellow at index 10. Below the array, three boxes labeled 'min', 'mid', and 'max' are shown with arrows pointing to the corresponding indices: 'min' points to index 0, 'mid' points to index 8, and 'max' points to index 16.

Arrays.binarySearch

```
// searches an entire sorted array for a given value  
// returns its index if found; a negative number if not found  
// Precondition: array is sorted  
Arrays.binarySearch(array, value)
```

```
// searches given portion of a sorted array for a given value  
// examines minIndex (inclusive) through maxIndex (exclusive)  
// returns its index if found; a negative number if not found  
// Precondition: array is sorted  
Arrays.binarySearch(array, minIndex, maxIndex, value)
```

- The `binarySearch` method in the `Arrays` class searches an array very efficiently if the array is sorted.
 - You can search the entire array, or just a range of indexes (useful for "unfilled" arrays such as the one in `ArrayIntList`)

Using `binarySearch`

```
// index    0  1  2  3  4  5  6  7  8  9  10  11  12  13  14  15
int[] a = {-4, 2, 7, 9, 15, 19, 25, 28, 30, 36, 42, 50, 56, 68, 85, 92};

int index = Arrays.binarySearch(a, 0, 16, 42); // index1 is 10
int index2 = Arrays.binarySearch(a, 0, 16, 21); // index2 is -7
```

- `binarySearch` returns the index where the value is found
- if the value is *not* found, `binarySearch` returns:
 - (`insertionPoint` + 1)
 - where `insertionPoint` is the index where the element *would* have been, if it had been in the array in sorted order.
 - To insert the value into the array, negate `insertionPoint` + 1

```
int indexToInsert21 = -(index2 + 1); // 6
```

Binary search

- Write a `binarySearch` method.
 - If the target value is not found, return its negative insertion point.

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	7	10	15	20	22	25	30	36	42	50	56	68	85	92	103

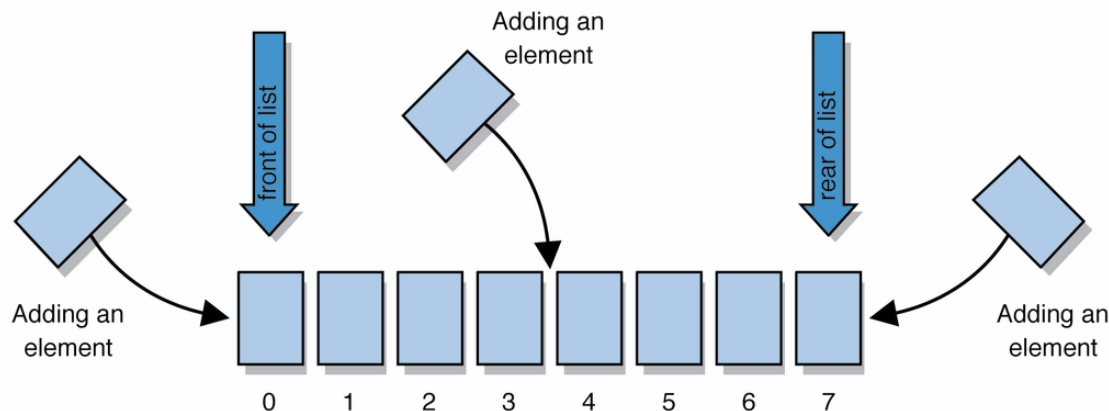
```
int index = binarySearch(data, 42); // 10
int index2 = binarySearch(data, 66); // -14
```

Java's ArrayList

- **collection:** an object that stores data ("elements")

```
import java.util.*; // to use Java's collections
```

- **list:** a collection of elements with 0-based **indexes**
 - elements can be added to the front, back, or elsewhere
 - a list has a **size** (number of elements that have been added)
 - in Java, a list can be represented as an **ArrayList** object



Type parameters (generics)

```
ArrayList<Type> name = new ArrayList<>();
```

- When constructing an `ArrayList`, you must specify the type of its elements in `< >`
 - This is called a *type parameter*; `ArrayList` is a *generic* class.
 - Allows the `ArrayList` class to store lists of different types.
 - Arrays use a similar idea with **Type** []

```
ArrayList<String> names = new ArrayList<>();  
names.add("Merlin");  
names.add("Percy");
```

Wrapper classes

Primitive Type	Wrapper Type
int	Integer
double	Double
char	Character
boolean	Boolean

- A **wrapper** is an object whose sole purpose is to hold a primitive value.
- Once you construct the list, use it with primitives as normal:

```
ArrayList<Double> grades = new ArrayList<>();  
grades.add(3.2);  
grades.add(2.7);  
...  
double myGrade = grades.get(0);
```

ArrayList methods

<code>add (value)</code>	appends value at end of list
<code>add (index, value)</code>	inserts given value just before the given index, shifting subsequent values to the right
<code>clear ()</code>	removes all elements of the list
<code>indexOf (value)</code>	returns first index where given value is found in list (-1 if not found)
<code>get (index)</code>	returns the value at given index
<code>remove (index)</code>	removes/returns value at given index, shifting subsequent values to the left
<code>set (index, value)</code>	replaces value at given index with given value
<code>size ()</code>	returns the number of elements in list
<code>toString ()</code>	returns a string representation of the list such as "[3, 42, -7, 15]"

* (a partial list; see the Java API for other methods)

ArrayList vs. array

```
String[] names = new String[5];           // construct
names[0] = "Jessica";                    // store
String s = names[0];                      // retrieve
for (int i = 0; i < names.length; i++) {
    if (names[i].startsWith("B")) { ... }
}
```

```
ArrayList<String> list = new ArrayList<>();
list.add("Jessica");                       // store
String s = list.get(0);                     // retrieve
for (int i = 0; i < list.size(); i++) {
    if (list.get(i).startsWith("B")) { ... }
}
```

Words exercise

- Write a program that reads a file and displays the words of that file as a list.
 - Then display the words in reverse order.
 - Then display them with all plural words (ending in "s") removed.

Exercise solution (partial)

```
ArrayList<String> allWords = new ArrayList<>();
Scanner input = new Scanner(new File("words.txt"));
while (input.hasNext()) {
    String word = input.next();
    allWords.add(word);
}

// display in reverse order
for (int i = allWords.size() - 1; i >= 0; i--) {
    System.out.println(allWords.get(i));
}

// remove all plural words
for (int i = 0; i < allWords.size(); i++) {
    String word = allWords.get(i);
    if (word.endsWith("s")) {
        allWords.remove(i);
        i--;
    }
}
```

ArrayList as param/return

```
public static void name(ArrayList<Type> name) {// param  
public static ArrayList<Type> name(params) // return
```

- Example:

```
// Returns count of plural words in the given list.  
public static int countPlural(ArrayList<String> list) {  
    int count = 0;  
    for (int i = 0; i < list.size(); i++) {  
        String str = list.get(i);  
        if (str.endsWith("s")) {  
            count++;  
        }  
    }  
    return count;  
}
```

Preconditions

- **precondition:** Something your method *assumes is true* at the start of its execution.

- Often documented as a comment on the method's header:

```
// Returns the element at the given index.  
// Precondition: 0 <= index < size  
public int get(int index) {  
    return elementData[index];  
}
```

- Stating a precondition doesn't really "solve" the problem, but it at least documents our decision and warns the client what not to do.
- What if we want to actually enforce the precondition?

Bad precondition test

- What is wrong with the following way to handle violations?

```
// Returns the element at the given index.  
// Precondition: 0 <= index < size  
public int get(int index) {  
    if (index < 0 || index >= size) {  
        System.out.println("Bad index! " + index);  
        return -1;  
    }  
    return elementData[index];  
}
```

- returning -1 no better than returning 0 (could be legal value)
- `println` is not a very strong deterrent to the client (esp. GUI)

Throwing exceptions

```
throw new ExceptionType ();
```

```
throw new ExceptionType ("message");
```

- Generates an exception that will crash the program, unless it has code to handle ("catch") the exception.
- Common exception types:
 - ArithmeticException, ArrayIndexOutOfBoundsException, FileNotFoundException, IllegalArgumentException, IllegalStateException, IOException, NoSuchElementException, NullPointerException, RuntimeException, UnsupportedOperationException
- Why would anyone ever *want* a program to crash?

Exception example

```
public int get(int index) {  
    if (index < 0 || index >= size) {  
        throw new ArrayIndexOutOfBoundsException(index);  
    }  
    return elementData[index];  
}
```

Postconditions

- **postcondition:** Something your method *promises will be true* at the end of its execution.
 - Often documented as a comment on the method's header:

```
// Makes sure that this list's internal array is large
// enough to store the given number of elements.
// Postcondition: elementData.length >= capacity
public void ensureCapacity(int capacity) {
    // double in size until large enough
    while (capacity > elementData.length) {
        elementData = Arrays.copyOf(elementData,
                                    2 * elementData.length);
    }
}
```

- If your method states a postcondition, clients should be able to rely on that statement being true after they call the method.