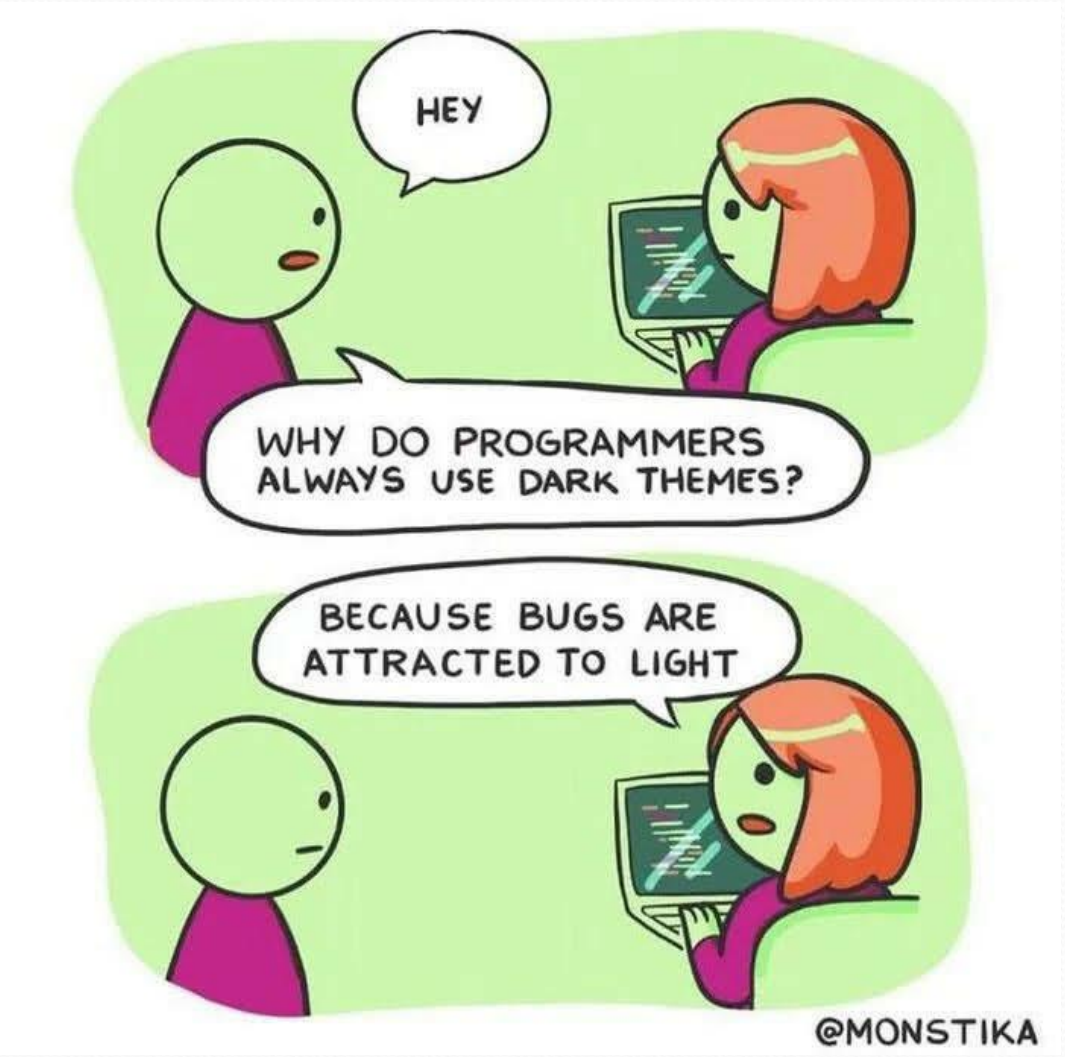
CS 142

Lecture 5:

using `ArrayList`s;

binary search;



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

Finishing `ArrayIntList`

- Our `ArrayIntList` class is pretty similar to Java's built in `ArrayList`. However, we still have some big issues:
  - Our `ArrayIntList` doesn't print itself out nicely
  - If we try to add more than 10 elements we will crash and not grow.
- Today in class:
  1. Fix our `ArrayIntList` printing.
  2. Make our `ArrayIntList` grow when necessary.
  3. Find out how to find the index of data more efficiently.

Printing an `ArrayIntList`

- How can we make our `ArrayIntList` print out nicely?
  - We can write a `toString` method!
    - Tells Java how to convert an object into a `String`

```
ArrayIntList list = new ArrayIntList();
System.out.println("list is " + list);
                // ("list is " + list.toString());
```

    - Syntax:

```
public String toString() {
    code that returns a suitable String;
}
```

Expanding `ArrayIntList`

- When are we in danger of running out of space?
- How can we increase the space available in `elementData`?
  - How much should we increase the space by each time?

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);
    }
}
```

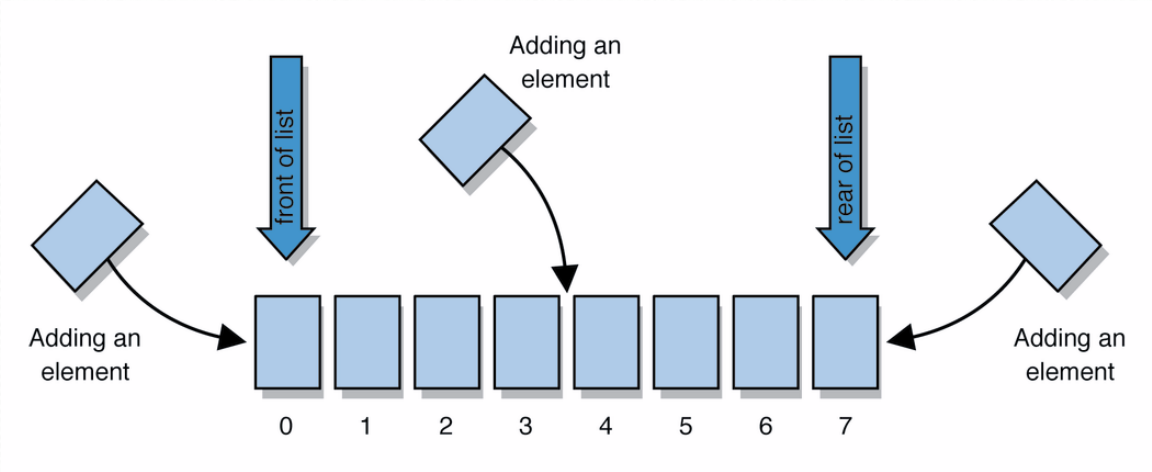
  - If your method states a postcondition, clients should be able to rely on that statement being true after they call the method.

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

`add(`**value**`)`	appends value at end of list
`add(`**index**`, `**value**`)`	inserts given value just before the given index, shifting subsequent values to the right
`clear()`	removes all elements of the list
`indexOf(`**value**`)`	returns first index where given value is found in list (-1 if not found)
`get(`**index**`)`	returns the value at given index
`remove(`**index**`)`	removes/returns value at given index, shifting subsequent values to the left
`set(`**index**`, `**value**`)`	replaces value at given index with given value
`size()`	returns the number of elements in list
`toString()`	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")) { ... }
}                                               // iterate
```

```
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")) { ... }
}                                               // iterate
```

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;
}
```

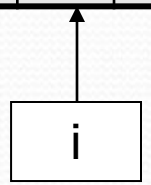
Searching methods

- From the Project 1 specification:
  - *In addition, the `indexOf` method should be rewritten to take advantage of the fact that the list is sorted. It should use the much faster **binary search** algorithm rather than the **sequential search** algorithm that is used in the original `ArrayIntList` class*

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**:

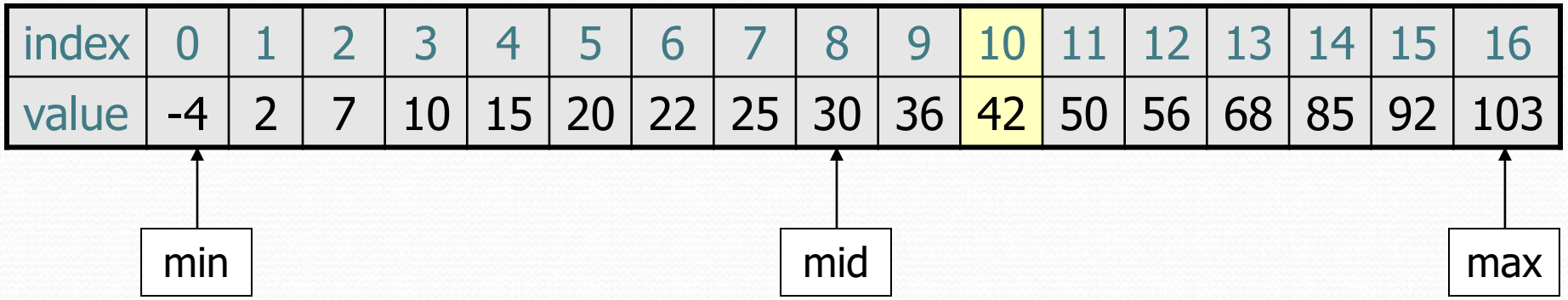
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	7	10	15	20	22	25	30	36	42	50	56	68	85	92	103



  - 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**:



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 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
```

How can we compare other types?

How can Java tell if one `Point` is greater or less than another `Point`?

```
Point[] places =
        {new Point(2, 3), new Point(1, 4), new Point(4, 3)};
```

Is this sorted?

The `compareTo` method

- The standard way for a Java class to define a comparison function for its objects is to define a `compareTo` method.
  - Example: in the `String` class, there is a method:

```
public int compareTo(String other)
```

- A call of **A**`.compareTo(`**B**`)` will return:

a value <	0	if **A** comes "before" **B** in the ordering,
a value >	0	if **A** comes "after" **B** in the ordering,
or	0	if **A** and **B** are considered "equal" in the ordering.

Using `compareTo`

- `compareTo` can be used as a test in an `if` statement.

```
String a = "alice";
String b = "bob";
if (a.compareTo(b) < 0) {   // true
    ...
}
```

Primitives	Objects
`if (a < b) { ...`	`if (a.compareTo(b) < 0) { ...`
`if (a <= b) { ...`	`if (a.compareTo(b) <= 0) { ...`
`if (a == b) { ...`	`if (a.compareTo(b) == 0) { ...`
`if (a != b) { ...`	`if (a.compareTo(b) != 0) { ...`
`if (a >= b) { ...`	`if (a.compareTo(b) >= 0) { ...`
`if (a > b) { ...`	`if (a.compareTo(b) > 0) { ...`