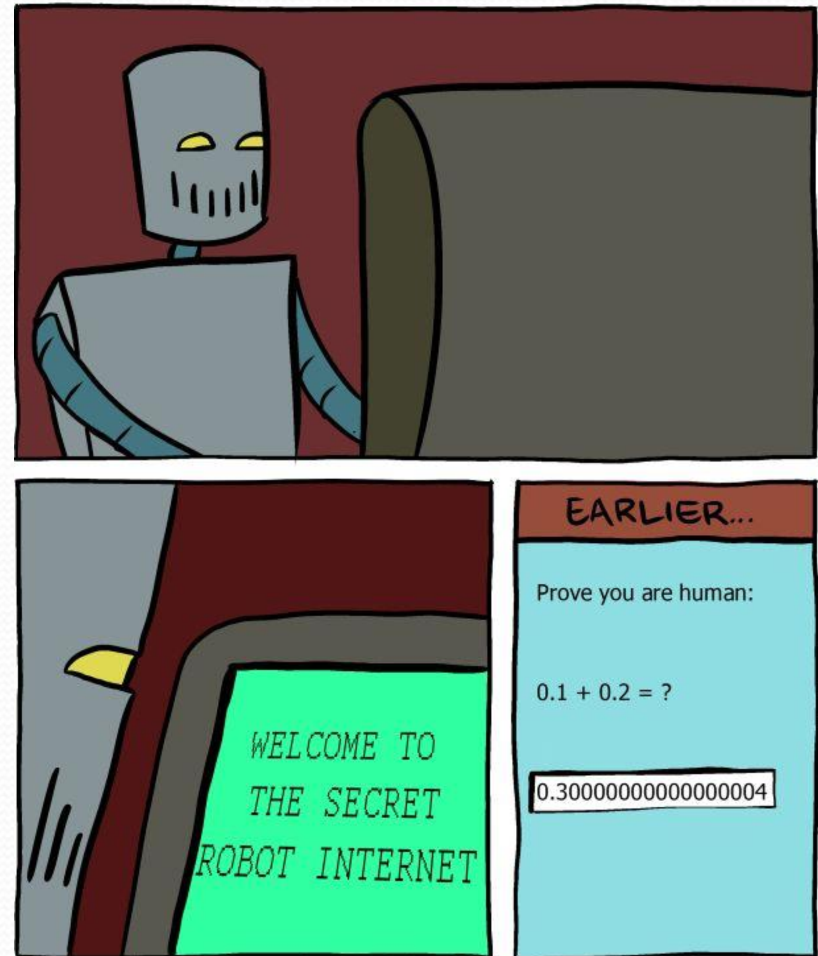
# CS 142

**Lecture 4:**

`ArrayList`;

binary search




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

# Array Limitations

What is frustrating about arrays?

- Fixed-size
- Adding or removing from middle is hard
- Not much built-in functionality (need `Arrays` class)

# Java's fix: `List` Abstraction

- Like an array that resizes to fit its contents.
- When a list is created, it is initially empty.

    ```
    []
    ```

- Use `add` methods to add to different locations in list

    ```
    [hello, ABC, goodbye, okay]
    ```

    - The list object keeps track of the element values that have been added to it, their order, indexes, and its total size.
    - You can add, remove, get, set, ... any index at any time.

# `ArrayList` implementation

- What is an `ArrayList`'s behavior?
  - add, remove, indexOf, etc
- What is an `ArrayList`'s state?
  - Many elements of the same type
  - For example, unfilled array

| *index* | *0* | *1* | *2* | *3* | *4* | *5* | *6* | *...* | *98* | *99* |
|---|---|---|---|---|---|---|---|---|---|---|
| *value* | 17 | 932085 | -32053278 | 100 | 3 | 0 | 0 | ... | 0 | 0 |

*size* 5

# `ArrayIntList` implementation

- Starting out simpler than Java's built-in `ArrayList`
  - Only stores `int`s
  - Fewer methods: `add(`**value**`)`, `add(`**index**`, `**value**`)`, `get(`**index**`)`, `set(`**index**`, `**value**`)`, `size()`, `isEmpty()`, `remove(`**index**`)`, `indexOf(`**value**`)`, `contains(`**value**`)`, `toString()`,

- Fields?
  - `int[]`
  - `int` to keep track of the number of elements added
  - The default capacity (array length) will be 10

# Implementing `add`

- How do we add to the end of a list?

```
public void add(int value) {   // just put the element
    list[size] = value;        // in the last slot,
    size++;                    // and increase the size
}
```

| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| value | 3 | 8 | 9 | 7 | 5 | 12 | 0 | 0 | 0 | 0 |
| size | 6 | | | | | | | | | |

  - `list.add(42);`

| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| value | 3 | 8 | 9 | 7 | 5 | 12 | **42** | 0 | 0 | 0 |
| size | **7** | | | | | | | | | |

# 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;
}
```

# 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];
}
```

# Class constants

`public static final` **type name** `=` **value**`;`

- **class constant**: a global, unchangeable value in a class
  - used to store and give names to important values used in code
  - documents an important value; easier to find and change later

- classes will often store constants related to that type
  - `Math.PI`
  - `Integer.MAX_VALUE`, `Integer.MIN_VALUE`
  - `Color.GREEN`

```
// default array length for new ArrayIntLists
public static final int DEFAULT_CAPACITY = 10;
```

# 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)

# 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:

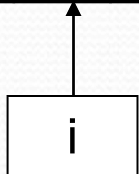
```
// Makes sure that this list's internal array is large
// enough to store the given number of elements.
// Postcondition: elementData.length >= capacity
public void ensureCapacity(int capacity) {
    // double in size until large enough
    while (capacity > elementData.length) {
        elementData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

# 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 |

min (index 0) mid (index 8) max (index 16)

# 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
```