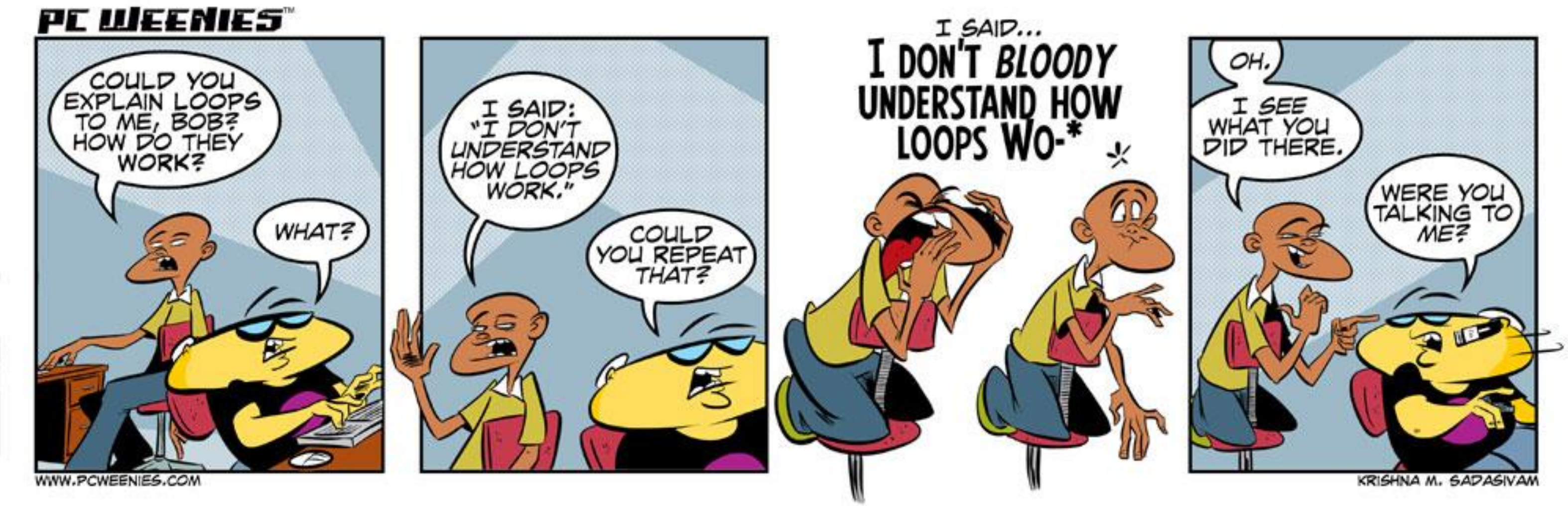
CSE 143

Lecture 2: Implementing `ArrayIntList`

reading: 15.1 - 15.3


PC WEENIES™
COULD YOU EXPLAIN LOOPS TO ME, BOB? HOW DO THEY WORK?
WHAT?
I SAID: "I DON'T UNDERSTAND HOW LOOPS WORK."
COULD YOU REPEAT THAT?
I SAID... I DON'T BLOODY UNDERSTAND HOW LOOPS WO-*
OH.
I SEE WHAT YOU DID THERE.
WERE YOU TALKING TO ME?
WWW.PCWEENIES.COM
KRISHNA M. SADASIVAM

Wrapper classes

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

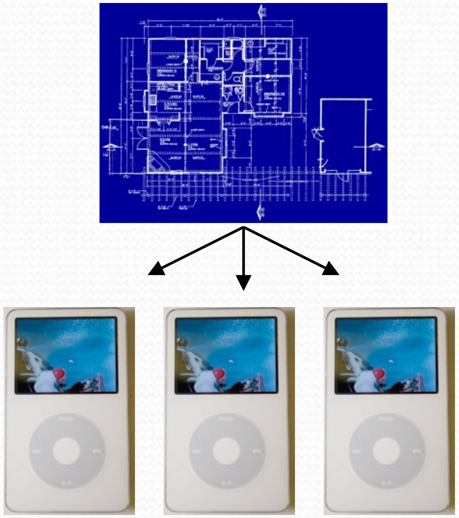
- A **wrapper** is an object whose sole purpose is to hold a primitive value.

- Once you construct the list, use it with primitives as normal:

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

Recall: classes and objects

- **class**: A program entity that represents:
  - A complete program or module, or
  - A template for a type of objects.
  - (`ArrayList` is a class that defines a type.)



- **object**: An entity that combines **state** and **behavior**.
  - **object-oriented programming (OOP)**: Programs that perform their behavior as interactions between objects.
  - **abstraction**: Separation between concepts and details. Objects provide abstraction in programming.

Elements of a class

```
public class BankAccount {
    private String name;     // fields:
    private int id;          //  data encapsulated
    private double balance;  //  inside each object

    public BankAccount(String name, int id) {
        this.name = name;     // constructor:
        this.id = id;         //  initializes
        this.balance = 0.0;   //  new objects
    }

    public void deposit(double amount) {
        this.balance += amount; // instance method:
    }                           //  each object's
    ...                         //  behavior
}
```

"**implicit parameter**": object on which a method was called

`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

- Simpler than `ArrayList<E>`
  - No generics (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

Printing an `ArrayIntList`

- Let's add a method that allows clients to print a list's elements.

  - You may be tempted to write a `print` method:

    ```
    // client code
    ArrayIntList list = new ArrayIntList();
    ...
    list.print();
    ```

    - Why is this a bad idea? What would be better?

The 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;
}
```

- Every class has a toString, even if it isn't in your code.
  - The default is the class's name and a hex (base-16) number:

```
ArrayIntList@9e8c34
```

toString solution

```
// Returns a String representation of the list.
public String toString() {
    if (size == 0) {
        return "[]";
    } else {
        String result = "[" + elementData[0];
        for (int i = 1; i < size; i++) {
            result += ", " + elementData[i];
        }
        result += "]";
        return result;
    }
}
```

Implementing add #2

- How do we add to the middle or end of the list?
  - must *shift* elements to make room for the value *(see book 7.4)*

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(3, 42);      // insert 42 at index 3`

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

  - Note: The order in which you traverse the array matters!

add #2 code

```
public void add(int index, int value) {
    for (int i = size; i > index; i--) {
        list[i] = list[i - 1];
    }
    list[index] = value;
    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(`**`3`**`, `**`42`**`);`

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

Other methods

- Let's implement the following methods in our list:
  - `get(`**index**`)`
    Returns the element value at a given index.
  - `set(`**index**`, `**value**`)`
    Sets the list to store the given value at the given index.
  - `size()`
    Returns the number of elements in the list.
  - `isEmpty()`
    Returns `true` if the list contains no elements; else `false`.
    (Why write this if we already have the `size` method?)

Implementing `remove`

- Again, we need to shift elements in the array
  - this time, it's a left-shift
  - in what order should we process the elements?
  - what indexes should we process?

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.remove(2);     // delete 9 from index 2`

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value	3	8	**7**	**5**	**12**	0	0	0	0	0
size	**5**				←					

Implementing `remove` code

```
public void remove(int index) {
    for (int i = index; i < size; i++) {
        list[i] = list[i + 1];
    }
    size--;
    list[size] = 0;        // optional (why?)
}
```

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.remove(`**`2`**`);    // delete 9 from index 2`

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value	3	8	**7**	**5**	**12**	0	0	0	0	0
size	**5**				←					