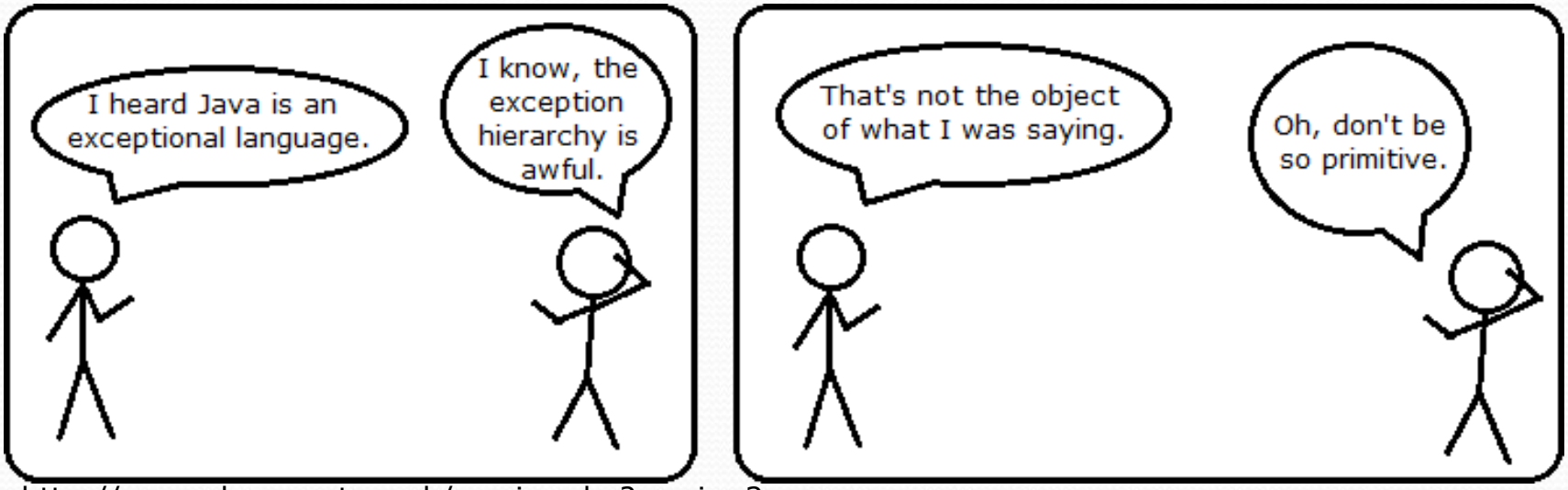
CS 142

Lecture 3: Class and object review; `ArrayLists`



http://www.alexsweet.co.uk/comics.php?comic=2

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

Recall: `Point` Class

```
public class Point {
    int x;
    int y;
    public void translate(int dx, int dy) {
        x = x + dx;
        y = y + dy;
    }
    public double distanceFromOrigin() {
        return Math.sqrt(x * x + y * y);
    }
}
public class PointMain {
    public static void main(String[] args) {
        Point p = new Point();
        p.x = 4;
        p.x = 5;
        System.out.println("(" + p.x + ", " + p.y + ")");
    }
}
```

Wouldn't it be nice if we could create a Point with x and y set from the beginning?

Constructors

- **constructor**: Initializes the state of new objects.

```
public type(parameters) {
    statements;
}
```

  - runs when the client uses the `new` keyword
  - no return type is specified;
    it implicitly "returns" the new object being created

  - If a class has no constructor, Java gives it a *default constructor* with no parameters that sets all fields to 0.

Multiple constructors

- A class can have multiple constructors.
  - Each one must accept a unique set of parameters.

- *Exercise:* Write a `Point` constructor with no parameters that initializes the point to (0, 0).

```
// Constructs a new point at (0, 0).
public Point() {
    x = 0;
    y = 0;
}
```

Private fields

- A field can be declared *private*.
  - No code outside the class can access or change it.

```
private type name;
```

  - Examples:

```
private int id;
private String name;
```

- Client code sees an error when accessing private fields:

```
PointMain.java:11: x has private access in Point
System.out.println("p1 is (" + p1.x + ", " + p1.y + ")");
                                   ^
```

Accessing private state

- We can provide methods to get and/or set a field's value:

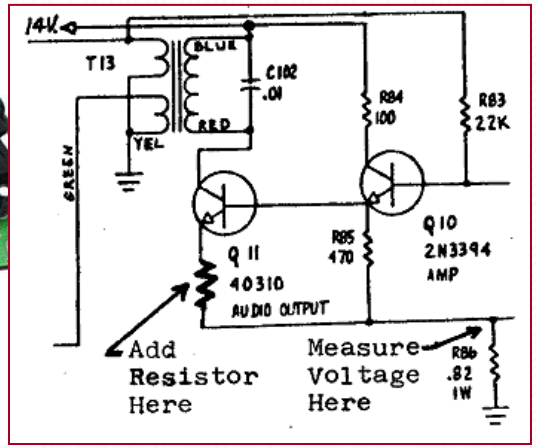
```
// A "read-only" access to the x field ("accessor")
public int getX() {
    return x;
}

// Allows clients to change the x field ("mutator")
public void setX(int newX) {
    x = newX;
}
```

  - Client code will look more like this:

```
System.out.println("p1: (" + p1.getX() + ", " + p1.getY() + ")");
p1.setX(14);
```

Encapsulation

- **encapsulation**: Hiding implementation details of an object from its clients.
  - Encapsulation provides *abstraction*.
    - separates external view (behavior) from internal view (state)
  - Encapsulation protects the integrity of an object's data.



From Last Time: Array Limitations

- 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.

```
[]
```

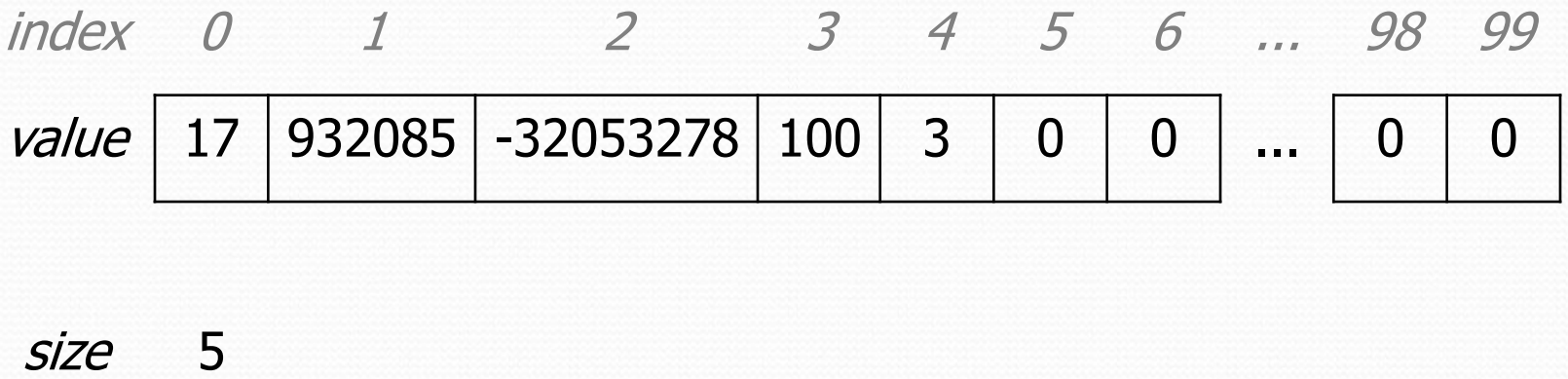
- Use `add` methods to add to different locations in list

```
[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



`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];
}
```