Advanced Programming Techniques in Java

COSI 12B

ArrayList

Lecture 17

Class Objectives

- `ArrayList` (section 10.1)
- Errors (Section 1.3)
- Complexity & Running time (section 13.2)

Review: ArrayList of any type of objects

- When constructing an `ArrayList`, you must specify the type of elements it will contain between `< >`
  - By making the `ArrayList` class a Generic class, the same `ArrayList` class can store lists of different types

- **Syntax**: `ArrayList<Type> name = new ArrayList<Type>();`

```
ArrayList<String> names = new ArrayList<String>();
```

- Java 7's shorter "diamond operator" syntax

```
ArrayList<String> names = new ArrayList<>();
```

Review: `ArrayList` of any type of objects (cont.)

- You can store any type of object in an `ArrayList` object
- `ArrayList<Point> points = new ArrayList<Point>();`
  - The `points` list will manipulate and return `Points`
- `ArrayList<Color> points = new ArrayList<Color>();`
  - The `points` list will manipulate and return `Colors`

Review: Issues with dynamic addition

- Assume you have an `ArrayList words`

```
words = [four, score, and, seven, years, ago]
```

- You want to add '~' before each word
- Solution 1:

```
for (int i=0; i < words.size(); i++) {
      words.add(i,'~');
}
```

- Does this work? - NO!

Review: Issues with dynamic removal

- We now want to redo this operation (remove '~')
- Write code that removes every other element starting from the first one

```
words = [~, four, ~, score, ~, and, ~,seven, ~, years, ~, ago]
```

- Does this work? Why?

```
for (int i=0; i < words.size(); i+=2) {
      words.remove(i);
}
```

- Output

```
words = [four, ~, score, ~, and, ~,seven,~, years, ~, ago]
words = [four, ~, ~, and, ~,seven, ~, years, ~, ago]
words = [four, ~, ~, and, seven, ~, years, ~, ago]
...
```

Review: Solution 1

- Again, dynamic shifting causes the problem
  - Once you remove an element, all the rest are shifted to the left
- Correct solution:

```
for (int i=0; i < words.size(); i++) {
      words.remove(i);
}
```

- Output

```
words = [four, score, and, seven, years, ago]
```

“Backwards” solution 1

```
for (int i=words.size()-2; i>=0; i-=2) {
        words.remove(i);
}
```

- Output

```
words = [~, four, ~, score, ~, and, ~,seven, ~, years, ago]
words = [~, four, ~, score, ~, and, ~,seven, years, ago]
...
words = [four, score, and, seven, years, ago]
```

Accessing items

- To get an element at a specific index you can use the **`get(int index)`** method

```
int sum=0;
for (int i=0; i < list.size(); i++) {
        String s = list.get(i);
        sum += s.length();
}
```

Avoid expensive shifts

- Every time you call the `add` and `remove` method you cause elements to be shifted in the array
  - That's expensive!

- You can use **`set(int index, E value)`**
  - It replaces the element at the `index` position with the `value` parameter (with no shifting)

- **Example:** `list.set(0, "Harvard");`
  - This replaces the first element in the list

Problem

- Write a program that reads a file and displays the words of that file as a list
  - **First display all words**
  - Then display them with all plurals capitalized
  - Then display them in reverse order
  - **Then display them with all plural words removed**

Solution v. 1

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

// 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--;
    }
}
```

Searching for elements

- You can search the list for an element:

```
if (list.contains("Chemistry")) {
    System.out.println("Chemistry is in the list");
} else {
    System.out.println("Chemistry is not found.");
}
```

- Output: `Chemistry is not found.`
- **`contains(Object o)`** Returns true if the list contains the specified element

Where is my element?

- Sometimes you may need to know where a value is
- Example: you want to replace the first occurrence of a word with another word
  - You must tell the `set` method where the word to be replaced is
  - Use **`indexOf(Object o)`**

The content of the list is changed
no need to create a new list and return it

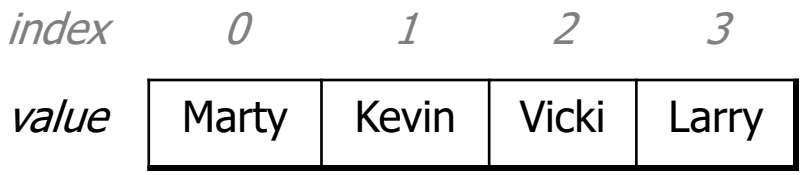
```
public static void replace (ArrayList<String> list, String target, String replacement)
          int index = list.indexOf(target);
          if (index>=0) {
                    list.set(index, replacement);
          }
}
```

How `contains` and `indexOf` work?

- How do search methods `contains`, `indexOf` understand that the object has the value you are looking for?
  - Remember: objects are just references to a memory address they don't hold any object state themselves!
  - They are using the `equals` method

Out-of-bounds

- Legal indexes are between **0** and the **list's size() - 1**
  - Reading or writing any index outside this range will cause an `IndexOutOfBoundsException`




```
ArrayList<String> names = new ArrayList<String>();
names.add("Marty");    names.add("Kevin");
names.add("Vicki");    names.add("Larry");
System.out.println(names.get(0));        // okay
System.out.println(names.get(3));        // okay
System.out.println(names.get(-1));       // exception
names.add(9, "Aimee");                   // exception
```

`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]"`

`ArrayList` methods (cont.)

`addAll(`**list**`)` `addAll(`**index**`, `**list**`)`	adds all elements from the given list to this list (at the end of the list, or inserts them at the given index)
`contains(`**value**`)`	returns true if given value is found somewhere in this list
`containsAll(`**list**`)`	returns true if this list contains every element from given list
`equals(`**list**`)`	returns true if given other list contains the same elements
`iterator()` `listIterator()`	returns an object used to examine the contents of the list (seen later)
`lastIndexOf(`**value**`)`	returns last index value is found in list (-1 if not found)
`remove(`**value**`)`	finds and removes the given value from this list
`removeAll(`**list**`)`	removes any elements found in the given list from this list
`retainAll(`**list**`)`	removes any elements *not* found in given list from this list
`subList(`**from**`, `**to**`)`	returns the sub-portion of the list between indexes **from** (inclusive) and **to** (exclusive)
`toArray()`	returns the elements in this list as an array

Arrays vs. ArrayList

- Construction

```
String[] names = new String[5];
ArrayList<String> list = new ArrayList<String>();
```

- Storing a value

```
names[0] = "Jessica";
list.set(0, "Jessica");
list.add("Jessica");
```

- Retrieving a value

```
String s = names[0];
String s = list.get(0);
```

Arrays vs. ArrayList (cont.)

- Doing something to each value that starts with "B"

```
for (int i = 0; i < names.length; i++) {
    if (names[i].startsWith("B")) { ... }
}
```

```
for (int i = 0; i < list.size(); i++) {
    if (list.get(i).startsWith("B")) { ... }
}
```

- Seeing whether the value "Benson" is found

```
if (names[i].equals("Benson")) { ... }
```

```
if (list.contains("Benson")) { ... }
```

Use `contains` to eliminate duplicates

- Assume you have the following file, and you want to get rid of duplicate names
  - *Maria Derek Erica Livia Jack Anita Kendall Maria Livia Derek Jamie Jack Erica*

- Output

```
list = [Maria, Derek, Erica, Livia, Jack, Anita, Kendal, Jamie]
```

Use `contains` to eliminate duplicates

- Assume you have the following file, and you want to get rid of duplicate names
  - *Maria Derek Erica Livia Jack Anita Kendall Maria Livia Derek Jamie Jack Erica*

```
Scanner input = new Scanner(new File("names.txt"));
ArrayList<String> list = new ArrayList<String>();
while (input.hasNext()){
          String name = input.next();
          if (!list.contains(name)) {
                    list.add(name);
          }
}
System.out.println("list = " + list);
```

- Output

```
list = [Maria, Derek, Erica, Livia, Jack, Anita, Kendal, Jamie]
```

`ArrayList` and enhanced for loop

- New loop syntax:

```
for (<type> <name> : <collection>) {
    <statement(s)>;
}
```

- This syntax can be used to examine an `ArrayList`:

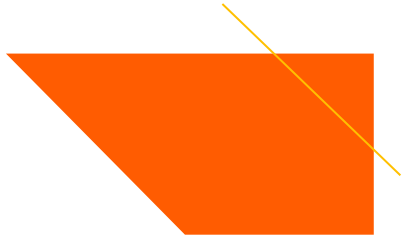
```
int sum = 0;
for (String s : list) {
    sum += s.length();
}
System.out.println("Total of lengths = " + sum);
```

Easier but trickier!

- You cannot skip elements with this syntax
- You cannot modify the lists while you are iterating over it

```
ArrayList<String> list = new ArrayList<String>();
for (String s : list) {
    System.out.println(s);
    list.remove(0);
}
```

- **`ConcurrentModificationException`** error



Errors

Three main categories of Errors

- Syntax Errors
- Run-time errors
- Logic errors

Syntax Errors

- You have a typo somewhere or wrote something the compiler didn't understand
- Easy to find because you just need to try to compile your code
- Syntax errors are UNACCEPTABLE, as it shows you never even had the chance to run your code
- The compiler actually tells you what's wrong with your code
- If you are overwhelmed by a multitude of errors, just look at them one at a time (top-most first), fix it, and compile again

Run-time Errors

- Run-time errors: Your program crashes during execution
- Reasonably easy to find with thorough testing, though **much** harder if code is multithreaded and the error is the result of a race condition
- Trace back where it happens to figure out what's wrong with the code; debuggers can be very helpful with this type of error

Logic Errors

- Hardest ones to fix
- Program doesn't crash but produces wrong output; it doesn’t do what you intended
- May result in code that leads to a race condition and manifests itself as a run-time error somewhere else
- Hardest part is that the bug usually comes from your thought processes, making you think a wrong line of code is producing good output when it isn't

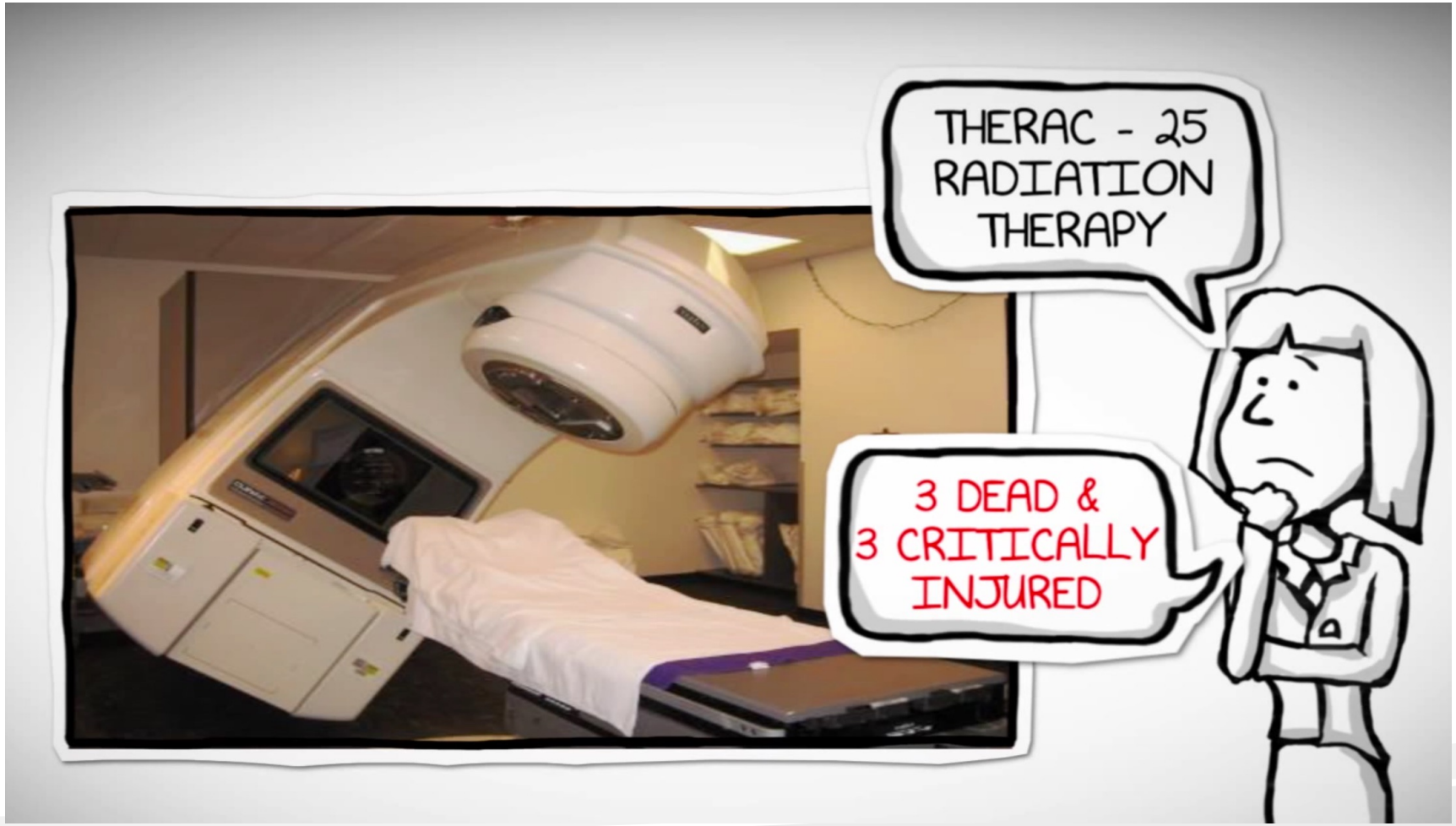
Purposes of Testing

- To make sure the software
  - meets the requirements that guided its design and development,
  - responds correctly to all kinds of inputs,
  - performs its functions within an acceptable time,
  - is sufficiently usable,
  - can be installed and run in its intended environments, and
  - achieves the general result its stakeholders desire

Why is Testing Important?

China Airlines Flight 140, April 26th, 1994

Why is Testing Important?



Why is Testing Important?

April, 1999

Why is Testing Important?

May, 1996

Paul Ehrlich:
"To err is human, but to really foul things up you need a computer."

Defect Clustering

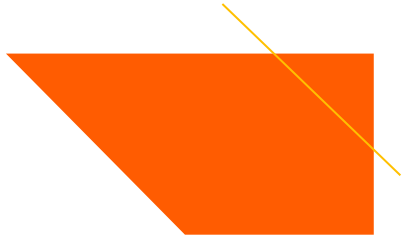
- A small number of modules contain most of the defects detected
- With experience, you can identify such risky modules
- However, this can lead to another problem..

Pesticide Paradox

- If the same tests are repeated over and over again, eventually the same test cases will no longer find new bugs
- This is called the Pesticide Paradox
- To overcome this, the test cases needed to be regularly reviewed and revised, adding new and different test cases to help find more defects
- You can never claim that your code is BUG-FREE
- Why?
- Absence of error is a fallacy

7 Testing Principles

- Testing shows presence of defects
- Exhaustive Testing is impossible
- Early Testing
- Defect Clustering
- Pesticide Paradox
- Testing is context dependent
- Absence of errors is a fallacy



Runtime Efficiency

Data Structure

Data structure

- a way to store and organize data in order to facilitate access and modification.
- no single data structure optimal for all purposes.
- usually optimized for a specific problem setting.
- important to know the strength and limitations of several of them.

Examples:

- Trees (binary search trees, red-black trees, b-trees, ...).
- Stacks (last in, first out), queues (first in, first out), priority queues.

Algorithms

Algorithms:

- well-defined computational procedure.
- takes value or set of values as input.
- produces value or set of values as output.
- tool for solving a **well-specified** computational problem.
- instance of a problem consists of the input needed to compute a solution to the problem.
- correct algorithm solves the given computational problem.

Sorting problem:

Input: A sequence of n numbers $a_1,...,a_n$.

Output: A permutation $a_1,...,a_n$ of the input sequence such that $a_1 \leq \cdots \leq a_n$.

Efficiency

Computing time and memory are bounded resources.

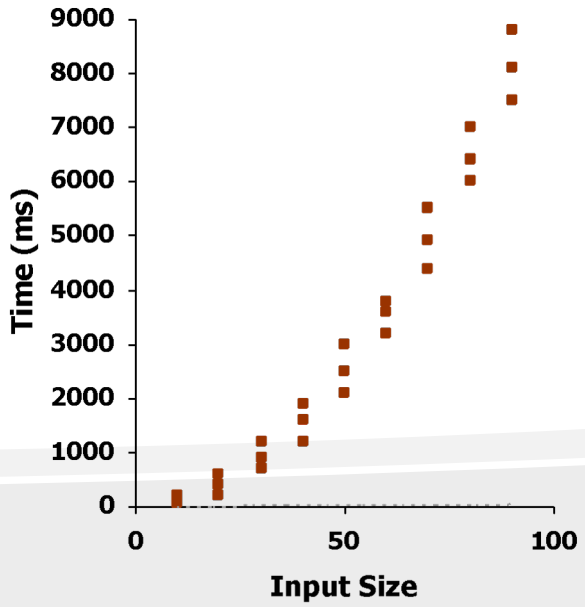
Efficiency:

- Different algorithms that solve the same problem often differ in their efficiency.
- More significant than differences due to hardware (CPU, memory, disks, ...) and software (OS, programming language, compiler, ...).

=> Running Time/Computational Complexity

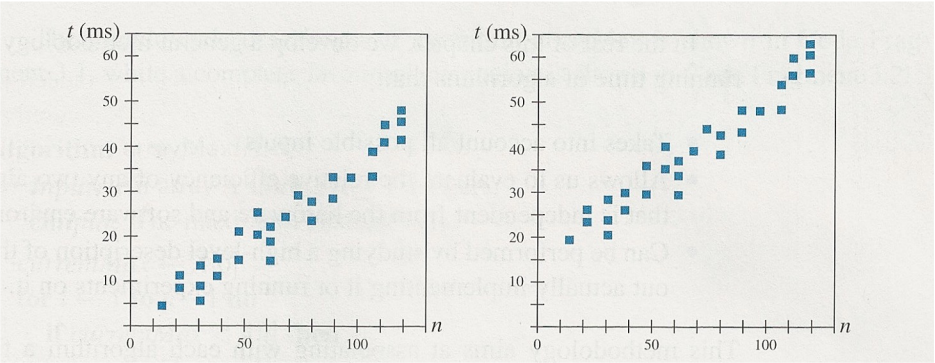
Empirical Analysis

- Run time can be studied experimentally
  - Write a program implementing the algorithm
  - Run the program with inputs of varying size
  - Get an accurate measure of the actual running time
  - Plot the results



Limitations of Empirical Analysis

- Experiment can be done only on a limited set of test inputs
- Difficult to compare the efficiency of two algorithms unless experiments have been performed on same environment
  - Hardware environment (processor, clock, rate, memory, etc.)
  - Software environment (OS, programming language, compiler, interpreter, etc.)



- Necessary to implement and execute an algorithm to study its run time