

CSE 143

Computer Programming II

Welcome to CSE 143!



CSE 142 vs. CSE 143: The Big Picture

In **CSE 142**, you learned how to use logic, control flow, and decomposition to write programs.

In **CSE 143**, you will learn to solve **more complex** and **larger** tasks **efficiently**.

Big Learning Goals

- Abstraction (implementation vs. client)
- Data Structures (organizing complex data)
- Algorithms (standard ways of completing common tasks)

We're going to build some **really cool** programs. And have a lot of fun!

Resources

- **TWO** sections a week
- Tons of TAs!
- The IPL (and my office hours!)
- Practice-It

Asking for help is not a sign of weakness; it's a sign of strength.

Course Website

<http://cs.uw.edu/143>

Section

We have **two** sections a week.

Each section has a set of problems; turn in **at least one** set of problems each week for credit.

Grading

- 47% programming projects, 3% section problems, 20% midterm, 30% final
- Weekly programming projects assigned **Fridays**, due on **Thursdays**
- 5 “free late days”; -2 points for subsequent days late; up to **3** days late on each hw

What does it mean for a program to be “correct”?

A program is only correct if it is **internally** correct and **externally** correct.

What does this code do?

```
1 --( __, __, __ ) { __ / __ <= 1 ? _ ( __, __ + 1, __ ) : ! ( __ % __ ) ? _ ( __, __ + 1, 0 ) : __ % __ == __ / __ && ! __ ? ( printf ( "%d\t", __ / __ ), _ ( __, __ + 1, 0 ) ) : __ % __ > 1 && __ % __ < __ / __ ? _ ( __, 1 + __, __ + ! ( __ / __ % ( __ % __ ) ) ) : __ < * __ ? _ ( __, __ + 1, __ ) : 0 ; } main () { _ ( 100, 0, 0 ) ; }
```

What is External Correctness?

The code does the right thing on **all** inputs.

What is Internal Correctness?

The code is...

- easy to read
- well documented
- well formatted
- efficient
- ...

Internal correctness matters, because:

- Do you want a job at a software engineering company?
- Do you want to ever reuse your code later?
- Do you want to ever write a large program? (Like a game, maybe)
- Important people think it does:
 - Programs must be written for people to read, and only incidentally for machines to execute. (Abelson & Sussman)

Goals For Internal Correctness

- Make non-obvious code obvious via comments.
- Document all features, limitations, design decisions.
- Make your code easy for someone else to read
- Explain what your methods, classes, etc. are supposed to do

Grading will be on both external and internal correctness!



Words Exercise

Write code to read a file and display its words in reverse order.

(Bad) Solution with Arrays

```
1 String[] words = new String[1000];
2 int i = 0;
3
4 Scanner inp = new Scanner(new File("words.txt"));
5 while (inp.hasNext()) {
6     String word = inp.next();
7     words[i] = word;
8     i++;
9 }
10 for (int j = i - 1; j >= 0; j--) {
11     System.out.println(words[j]);
12 }
```



Arrays are one way to store **many** values of the same type (`int`, `String`, `DrawingPanel`, etc.).

```
int[] arr = new int[8];
```

```
arr: 

|        |        |        |        |        |        |        |        |
|--------|--------|--------|--------|--------|--------|--------|--------|
| 12     | 49     | -2     | 26     | 2      | 6      | 26     | 11     |
| arr[0] | arr[1] | arr[2] | arr[3] | arr[4] | arr[5] | arr[6] | arr[7] |


```

“Element #3 is 26”

“arr has size 8”

Limitations of Arrays

- Fixed, upfront size (once you create the array, it will remain that size)
- Adding and removing can get complicated
- No methods (and weird “.length” syntax)

Functionality for arrays is in the `Arrays` class:

- `Arrays.copyOf`
- `Arrays.equals`
- `Arrays.sort`
- `Arrays.toString`

Collections

Collections store **many** pieces of data of **the same type**.

In Java, collections are in the `util` package:

```
import java.util.*;
```

Different collections have different properties:

- “Data ordered by indices”
- “Sorted data”
- “Data without duplicates”
- etc.

Lists

A **list** is a collection of elements ordered by a 0-based index.

- It supports add/remove from anywhere!
- The size isn't fixed!
- There are multiple implementations; first, `ArrayList`

- Suppose we have an ArrayList with values: [1, 2, -6]:

Step 0:

1	2	-6
---	---	----

 ...

0 1 2

- Insert 5 at index 2:

Step 1:

1	2	5	-6
---	---	---	----

 ...

0 1 2 3

- Add 0 at the beginning:

Step 2:

0	1	2	5	-6
---	---	---	---	----

 ...

0 1 2 3 4

- Get index 3:

`arrayList.get(3)` → 5

<code>add(val)</code>	Appends val to the end of the list
<code>add(idx, val)</code>	Puts val at index idx ; all elements at indices idx and larger get shifted forward
<code>get(idx)</code>	Returns the value at index idx
<code>set(idx, val)</code>	Replaces the value at index idx with val
<code>remove(idx)</code>	Removes and returns the value at index idx ; all elements at higher indices get shifted backward
<code>clear()</code>	Removes all elements from the list
<code>size()</code>	Returns the number of elements in the list
<code>indexOf(val)</code>	Returns the smallest index such that <code>get(idx).equals(val)</code> , or -1 if there is no such index
<code>toString()</code>	Returns a string representation of the list such as <code>[3, 42, -7, 15]</code>



Recall that we can create arrays of different types:

```
{1, 2, 5, 2}           {"hi", "banana"}  
(new int[4])          (new String[2])
```

Since the array initializations specify the **type** of the elements, the declaration for `ArrayList`'s should too:

```
[1, 2, 5, 2]          ["hi", "banana"]  
(new ArrayList<Integer>) (new ArrayList<String>)
```

`ArrayList` is a **generic** class which means that it can handle any type you want! Java knows the type by what you put in `<>`:

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

```
String[] arr = new String[5];
arr[0] = "hi";
arr[1] = "bye";
String s = arr[0];
for (int i=0; i < arr.length; i++) {
    if (names[i].contains("b")) {...}
}

→ ArrayList<String> list = new ArrayList<String>();
→ list.add("hi");
→ list.add("bye");
→ String s = list.get(0);
→ for (int i = 0; i < list.size(); i++) {
→     if (list.get(i).contains("b")) {...}
→ }
```

Note that these two pieces of code have **different** loop bounds:

`arr.length == 5`

`list.size() == 2`

ArrayList is just another type (like DrawingPanel or String)!

```
1 public void methodName(..., ArrayList<Type> name, ...) { ... }
2 public ArrayList<Type> methodName(...) { ... }
```

The following takes in an ArrayList and returns a new list containing only the words that start with x:

```
1 public ArrayList<String> startingWithX(ArrayList<String> list) {
2     ArrayList<String> newList = new ArrayList<String>();
3     for (int i=0; i < list.length; i++) {
4         if (list.get(i).startsWith("x")) {
5             newList.add(list.get(i));
6         }
7     }
8     return newList;
9 }
```



Words Exercise... Now with more ArrayList!

Write code to read a file and display its words...

- 1 in reverse order (but using an ArrayList)
- 2 with all words ending in "s" capitalized
- 3 with all words ending in "s" removed

```
1 /* Read in the words */
2 ArrayList<String> allWords = new ArrayList<String>();
3 Scanner input = new Scanner(new File("words.txt"));
4 while (input.hasNext()) {
5     String word = input.next();
6     allWords.add(word);
7 }
8
9 /* Display in Reverse Order */
10 for (int i = allWords.size() - 1; i >= 0; i--) {
11     System.out.println(allWords.get(i));
12 }
```



```
1  /* Remove All Words Ending in 's' */
2  for (int i = 0; i < allWords.size(); i++) {
3      String word = allWords.get(i);
4      if (word.endsWith("s")) {
5          allWords.remove(i);
6
7          /* This is the tricky part; since we removed a word,
8             * we're actually at the SAME index again! */
9          i--;
10     }
11 }
```