Class 3

CSC 240 Computer Science III

Chapter 9: Text Processing & Wrapper Classes

Dr. Si Chen (schen@wcupa.edu)

Tue / Thu · 9:30 – 10:45 AM · Anderson Hall 315
Fall 2026 · Week 2 · Thursday, September 3, 2026
HW1 is out · wcu.ninja/csc240

Review:
Static Methods

Static Methods

- Methods can also be declared static by placing the `static` keyword between the access modifier and the return type of the method.

```
public static String toString(int[] nums)
{…}
```

- When a class contains a static method, it is not necessary to create an instance of the class in order to use the method.

```
int[] numbers = {1, 5, 8, 7, 10};
String numbersInString = Arrays.toString(numbers);
```

Static Methods

- Static methods are convenient because they may be called at the class level.
- They are typically **used to create utility classes**, such as the `Math` class in the Java Standard Library.
- Static methods may not communicate with instance fields, only static fields.

```
1  public class Metric
2  {
3
4      public static double milesToKilometers(double m)
5      {
6          return m * 1.609;
7      }
8
9
10     public static double kilometersToMiles(double k)
11     {
12         return k / 1.609;
13     }
14 }
```

Review:
ArrayList Class

The `ArrayList` Class

- Similar to an array, an `ArrayList` allows object storage
- Unlike an array, an `ArrayList` object:
  - **Automatically expands** when a new item is added
  - **Automatically shrinks** when items are removed
- Requires:

```
import java.util.ArrayList;
```

Creating an `ArrayList`

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

Notice the word `String` written inside angled brackets <>

This specifies that the `ArrayList` can hold `String` objects.

If we try to store any other type of object in this <*String*> `ArrayList`, an error will occur.

ArrayList accepts **only reference types** as its element, **not primitive datatypes**. When trying to do so it produces a compile time error.

But I want to create a ArrayList to store integers (or other primitive data types)

How to bypass this problem?

Answer: Wrapper Class

PART 1

Wrapper classes

Turning primitives into objects — and the handy static methods that come with them

Wrapper classes: a primitive, wrapped in an object

```
int      -> Integer      double  -> Double
char     -> Character    boolean -> Boolean

// ArrayList needs objects, so wrappers make this legal:
ArrayList<Integer> nums = new ArrayList<Integer>();
nums.add(5);               // AUTO-BOXING: 5 -> Integer
int first = nums.get(0);  // AUTO-UNBOXING: Integer -> int

// Very useful STATIC methods (no object needed):
int n = Integer.parseInt("42");      // String -> int
double d = Double.parseDouble("3.5");// String -> double
```

Each primitive has a **wrapper class**: `int`→`Integer`, `double`→`Double`, etc.

Java **auto-boxes** (`5` → `Integer`) and **auto-unboxes** for you — it feels seamless.

Wrapper objects are **immutable** — you cannot change the value once created.

`Integer.parseInt(...)` / `Double.parseDouble(...)` convert **text to numbers** — you will use these constantly.

Wrappers are in java.lang — no import needed. parseInt/parseInt are static utilities.

PART 2

The Character class

Testing and converting individual characters

Character: test and convert a single char

```
Character.isDigit('7')      // true   — is it
0-9?
Character.isLetter('A')     // true   — is it
a-z / A-Z?
Character.isLetterOrDigit('#') // false
Character.isUpperCase('a')  // false
Character.isSpaceChar(' ')  // true

Character.toUpperCase('a')  // 'A'
Character.toLowerCase('Z')  // 'z'

// walk a String character by character with
charAt(i):
for (int i = 0; i < s.length(); i++)
    if (Character.isDigit(s.charAt(i)))
count++;
```

The `Character` class wraps a `char` and provides **testing** methods: `isDigit`, `isLetter`, `isLetterOrDigit`, `isUpperCase`, `isLowerCase`, `isSpaceChar`.

Conversion methods: `toUpperCase`, `toLowerCase` (return the converted char).

Combined with `s.charAt(i)` and `s.length()`, you can **scan a string** and classify each character.

PART 3

String methods

Searching, extracting, and transforming text

Searching a String

Find

`str.startsWith("Four")` → does it begin with it?

`str.endsWith(".java")` → does it end with it?

`str.indexOf("is")` → **first** position (or −1).

`str.lastIndexOf("is")` → **last** position.

All are **case-sensitive**.

Extract & transform

`str.charAt(0)` → the char at an index.

`str.substring(2, 5)` → chars 2..4 (a substring).

`str.replace("a","A")` → a new string.

`str.toUpperCase()` / `toLowerCase()`.

`str.trim()` → strip surrounding spaces.

Why StringBuilder? Strings are immutable

The problem

A `String` **cannot be changed** — every "edit" makes a **new** String.

Building text in a loop with `+` creates **many throwaway objects** — slow for big text.

`String s = ""; for(...) s += x;` is a classic performance trap.

The fix: StringBuilder

```
`StringBuilder` is a **mutable** string
you can modify in place:
sb.append("Hello");
sb.append(" ").append(name);
sb.insert(0, ">> ");
sb.reverse();
String result = sb.toString();
```

Tokenizing: splitting text into pieces

```
String line = "Ada,Lovelace,1815";

// modern way: String.split (regex delimiter)
String[] parts = line.split(",");
// parts = {"Ada", "Lovelace", "1815"}
System.out.println(parts[0]);   // Ada

// classic way: StringTokenizer
StringTokenizer st = new StringTokenizer(line, ",");
while (st.hasMoreTokens())
    System.out.println(st.nextToken());
```

Tokenizing = breaking a string into pieces (tokens) at a **delimiter**.

`split(",")` returns a `String[]` — the modern, common approach.

`StringTokenizer` walks tokens with `hasMoreTokens()` / `nextToken()`.

This is how you parse CSV lines, user input, and file records.

In-class: process a name record

IN-CLASS ⏱ 8 min · discuss

Given `String rec = "lovelace,ada,1815";`, write the logic (sketch it):

- **Split** the record on the comma into its three fields.
- Capitalize the **last name** and **first name** (first letter upper).
- Use `Integer.parseInt` to get the **year** as an int; compute age in 2026.

Discuss: would you build the formatted output with `+` or a `StringBuilder`? Why?

Hints:

- rec.split(",") -> String[3]
- Capitalize: Character.toUpperCase(s.charAt(0)) + s.substring(1)
- Integer.parseInt(parts[2])
- For one line, + is fine; for a loop over many records, StringBuilder.

This is the TestScoreReader pattern from Chapter 9 — and HW1.

HW1 + before Tuesday

HW1 is assigned today (D2L / course site) — text processing with the Chapter 9 tools.

Practice: write a small program that **counts the digits and letters** in a user-entered string.

Read **Gaddis Chapter 9** sections on `StringBuilder` and the numeric wrapper classes.

Course site: **wcu.ninja/csc240**. Next: we finish Ch. 9 and begin **Chapter 10 (Inheritance)**.