Class

4

CSC 240 Computer Science III

Tokenizing & Parsing: From Text to Data

Dr. Si Chen (schen@wcupa.edu)

Tue / Thu · 9:30 – 10:45 AM · Anderson Hall 315
Fall 2026 · Week 3 · Tuesday, September 8, 2026
HW1 due Tue Sep 15, 11:59 PM · Quiz 1 Thursday · wcu.ninja/csc240

Recap from last class

- **Wrapper classes** (`Integer`, `Double`, …) make primitives into objects; `Integer.parseInt("42")` turns **text into a number**.
- `Character.isDigit / isLetter / toUpperCase` test and convert single chars; `charAt(i)` + `length()` scans a string.
- String methods: `indexOf`, `substring`, `startsWith`, `trim`, `replace`; **Strings are immutable** → `StringBuilder` for edits.
- Tokenizing preview: `line.split(",")` → `String[]`; `StringTokenizer` walks tokens.
- Today: tokenizing **in depth** — because every file, every line of input, every API response arrives as **one string**.

PART 1

Two tokenizers: split() and StringTokenizer

Breaking one string into pieces at a delimiter — the first step of every text-processing program

split() vs StringTokenizer — same job, two styles

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

// 1) String.split -> the whole String[]
String[] parts = line.split(" ");
System.out.println(parts.length);   // 4
System.out.println(parts[2]);       // 1815

// 2) StringTokenizer -> one token at a time
StringTokenizer st =
    new StringTokenizer(line);
System.out.println(st.countTokens()); // 4
while (st.hasMoreTokens())
    System.out.println(st.nextToken());

// several delimiters at once:
new StringTokenizer("09/08/2026", "/");
"09/08/2026".split("/"); // {"09","08","2026"}
```

- `split(delim)` returns a `String[]` — random access by index; the natural fit for **fixed-format records** (CSV).
- `StringTokenizer` (import `java.util`) hands out tokens **one at a time**: `hasMoreTokens()` / `nextToken()` / `countTokens()`.
- Default `StringTokenizer` delimiters: **space, tab, newline**. Pass a String of characters to change them.
- Gaddis teaches both; the modern default is `split`. Read both — you will meet both in the wild.

SplitVsTokenizer.java in the Class 4 bundle.

StringTokenizer, the whole API

Constructor & methods (Gaddis Table 9-x)

- `new StringTokenizer(str)` — delimiters: whitespace.
- `new StringTokenizer(str, "delims")` — each **character** in the string is a delimiter (`",;"` = comma **or** semicolon).
- `new StringTokenizer(str, delims, true)` — return the delimiters as tokens too.
- `hasMoreTokens()` → `boolean`; `nextToken()` → next `String`; `countTokens()` → how many remain.
- Calling `nextToken()` with none left throws `NoSuchElementException` — always check `hasMoreTokens()` first.

Multiple delimiters

```
String rec = "lovelace;ada,1815";
StringTokenizer st =
    new StringTokenizer(rec, ";,");
while (st.hasMoreTokens()) {
    String tok = st.nextToken();
    System.out.println(tok);
}
// lovelace
// ada
// 1815

// empty fields VANISH:
// "a,,b" -> 2 tokens (a, b)
// "a,,b".split(",") -> 3 (a, "", b)
```

PART 2

split() takes a regular expression

The delimiter is a pattern, not a character — a superpower and a trap

The delimiter is a PATTERN

```
String data = "apple, banana,cherry ,  date";

data.split(",");          // " banana", "cherry "
(messy)
data.split("\\s*,\\s*"); // "banana", "cherry"
(clean)

"one   two    three".split(" ");   // 9 tokens, 6
empty!
"one   two    three".split("\\s+");// 3 tokens

// THE classic bug: "." = ANY character
"192.168.1.25".split(".");    // length 0 (!!)
"192.168.1.25".split("\\.");  //
{"192","168","1","25"}
"192.168.1.25".split("[.]");  // same (char class)
```

- `\s` = any whitespace, `+` = one or more, `*` = zero or more. In Java source you write `"\\s+"` (escape the backslash).
- `"\\s*,\\s*"` = "a comma with optional spaces around it" — trims fields for free.
- **Metacharacters** `. * + ? | ( ) [ ] { } ^ $ \` mean something in a regex. To split on a literal dot: `"\\."` or `"[.]"`.
- Rule: if `split(".")` gives you an empty array, you just met this bug.

RegexDelimiters.java — run it and watch the token counts.

Regex cheat sheet: the 20% you need

Characters

`.` any char

`\d` digit `\w` word char

`\s` whitespace

`[abc]` one of `[^abc]` none of

`[a-z]` range

Repetition

`*` 0 or more

`+` 1 or more

`?` 0 or 1

`{3}` exactly 3

`{2,5}` between 2 and 5

Structure

`|` or: `"cat|dog"`

`( )` group

`^` start `$` end

`\\.` literal dot (in Java source)

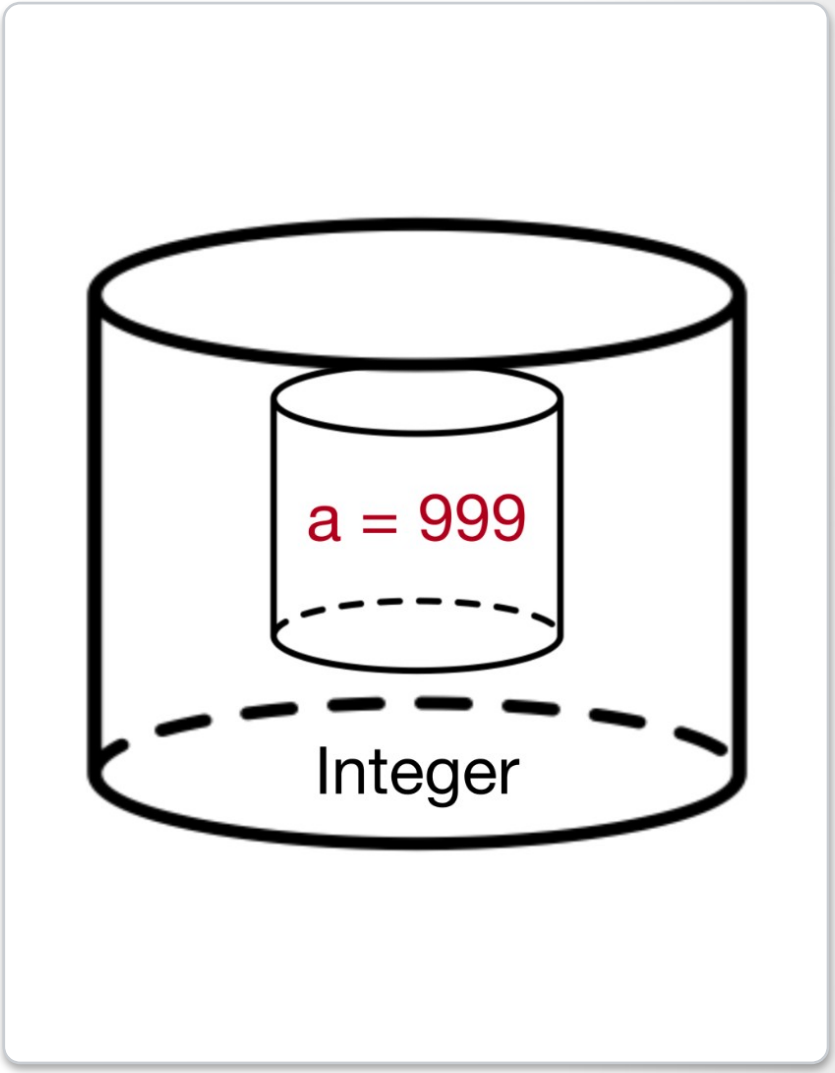
Use `Pattern.quote(",")` to escape safely

PART 3

Parsing: tokens are text until you convert them

parseInt / parseDouble, trim, validation, and what happens when the text is not a number

Remember the can: text → int lives inside Integer



From my 2019 Class 2 deck: the wrapper "can" holding a primitive.

- A token like `"92"` is **text**. `Integer.parseInt("92")` gives the **int** 92; `Double.parseDouble("88.5")` gives 88.5.
- Those are **static** methods on the wrapper classes (my Class 2/3 "can" picture: the primitive inside the object).
- `Integer.valueOf("92")` returns an **Integer object** (boxed) instead — handy for `ArrayList<Integer>`.
- Going back: `String.valueOf(92)`, `Integer.toString(92)`, or just `"" + 92`.

Parsing safely

```
String record = "Ada,92,88.5";
String[] f = record.split(",");
int    quiz = Integer.parseInt(f[1]);       // 92
double exam = Double.parseDouble(f[2]);     // 88.5

Integer.parseInt(" 42 ");        //
NumberFormatException!
Integer.parseInt(" 42 ".trim()); // 42 -> ALWAYS trim

try {
    int n = Integer.parseInt("ninety");
} catch (NumberFormatException e) {
    System.out.println("bad: " + e.getMessage());
}

// or validate first with the Character class:
static boolean isAllDigits(String s)
    { ...Character.isDigit(s.charAt(i))... }
```

ParseNumbers.java — includes the isAllDigits helper.

- Whitespace, `$`, commas inside numbers (`"1,200"`) all make `parseInt` **throw** `NumberFormatException`.
- Two defenses: **validate first** (scan with `Character.isDigit`) or **catch the exception** (`try`/`catch` — Chapter 11 does this properly; the pattern is shown so you recognize it).
- `trim()` every token that came from a file or a user. Every time.
- HW1 Q3: the gas-price CSV has a **header line** and **dates** — decide which fields to parse and which to keep as text.

Scanner is a tokenizer too

```
// Scanner reads tokens from a String, File, or
System.in
Scanner sc = new Scanner("3 apples cost 4.50
dollars");
int count = sc.nextInt();        // 3
String item = sc.next();         // "apples"
sc.next();                       // skip "cost"
double price = sc.nextDouble();  // 4.50

// change the delimiter (a regex, like split):
Scanner csv = new Scanner("Ada,Lovelace,1815")
                  .useDelimiter(",");

// check BEFORE you parse -> no exception:
Scanner mixed = new Scanner("10 twenty 30");
while (mixed.hasNext())
    if (mixed.hasNextInt()) sum += mixed.nextInt();
    else mixed.next();       // skip the word
```

- You have used `Scanner` on `System.in` since CSC 141 — it is a **tokenizer** with typed `next…()` methods.
- `nextInt()` = `next()` + `parseInt()` in one step (and it throws `InputMismatchException` on bad text).
- `hasNextInt()` / `hasNextDouble()` let you **look before you leap** — the cleanest validation.
- `useDelimiter(regex)` turns Scanner into a CSV reader. Files: `new Scanner(new File("x.csv"))`.

ScannerTokens.java — same problems, third tool. Pick the one that reads best.

PART 4

From a record to an object

Parsing is where text processing meets encapsulation

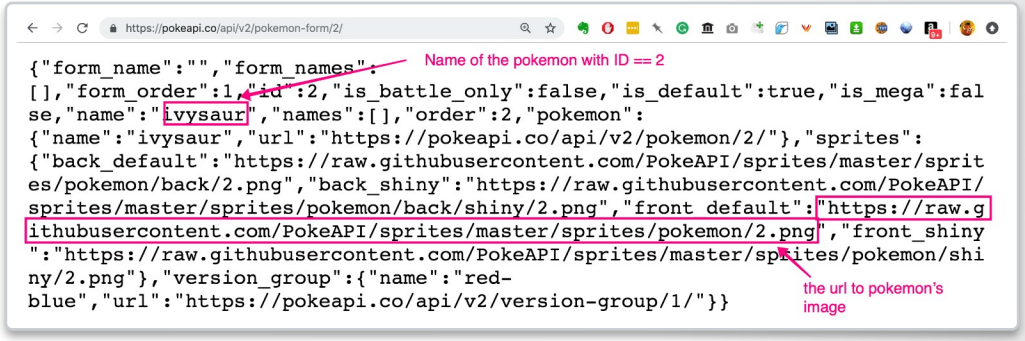
One line of CSV → one Person object

```
class Person {
    private String first, last;
    private int birthYear;

    public Person(String f, String l, int y) { ... }

    // static FACTORY: text in, object out
    public static Person fromCsv(String line) {
        String[] f = line.split(",");
        return new Person(f[0].trim(), f[1].trim(),

Integer.parseInt(f[2].trim()));
    }
    public int ageIn(int year) { return year -
birthYear; }
    @Override public String toString() {
        return last.toUpperCase() + ", " + first; }
}
// main: for (String line : lines)
//           people.add(Person.fromCsv(line));
```

- Put the parsing in **one place** — a static `fromCsv` method — so the format is defined once.
- The rest of the program works with **objects** (`p.ageIn(2026)`), never with `f[2]` again.
- This is the engineering discipline from Class 2: **encapsulate the messy part**.
- Next week: `Student extends Person` — the object you parse can be a **subclass** (Chapter 10).

CsvRecordParser.java — the pattern HW1 Q3 and every future file-reading program should follow.

Text processing in the real world: an API response



- Web APIs return **one long string** (JSON). To a Java program it is just text — until you parse it.
- My 2019 Pokémon Viewer: find `"name":"` with `indexOf`, then `substring` up to the next `"` → `ivysaur`.
- Same tools, same pattern: **locate** → **extract** → **convert**. (Libraries like Gson do this for you later; the idea is identical.)
- Every AI chatbot you use does the reverse: it **tokenizes** your text first — GPT-style models split English into tokens of roughly 4 characters each.

PokeAPI response, annotated in my 2019 Class 4 deck.

Parsing is where "almost right" code fails quietly

66%

of developers say their top AI frustration is answers that are "almost right, but not quite"

Stack Overflow Developer Survey 2025

45%

say debugging AI-generated code is more time-consuming than expected

Stack Overflow Developer Survey 2025

~4

characters per token: how GPT-style models tokenize English before they "read" it

OpenAI tokenizer documentation

`split(".")`, a header line parsed as data, `parseInt` on `" 42"`, `"a,,b"` losing a field — none of these produce a compile error, and an AI will happily generate all of them. Test your parser on the **first line, the last line, and one malformed line** before you trust it (or the model that wrote it).

In-class: parse a date, then a Pokémon

IN-CLASS ⏱ 10 min

- **A (5 min).** Given `String date = "09/08/2026";` write code that prints `September 8, 2026`. Steps: split on `/`, `parseInt` the pieces, use a `String[] MONTHS` array. Why `parseInt` before printing the day?
- **B (5 min).** Given the JSON string `s` above, write the two lines that extract the value of `"name"` using `indexOf` and `substring`. Hint: search for `"\"name\":\""`, then for the next `"`.
- **Discuss:** which of `split`, `StringTokenizer`, `Scanner` would you use for each? Why?

A:

```
String[] p = date.split("/");
int m = Integer.parseInt(p[0]);
int dd = Integer.parseInt(p[1]);
println(MONTHS[m-1] + " " +
dd
   + ", " + p[2]);   // "08" -> 8
```

B:

```
int i = s.indexOf("\"name\":\"")
        + 8;
int j = s.indexOf("\"", i);
String name = s.substring(i, j);
```

HW1 checkpoint + before Thursday

- **HW1 due Tuesday, Sep 15, 11:59 PM** on D2L. Q3 GasPrices is **today's lecture**: skip the header, `split(",")`, `parseDouble`, arrays. Start it tonight.
- **Quiz 1 Thursday** (first 15 min): Chapter 9 — wrapper classes & parse methods, `Character`, String methods, `StringBuilder`, `split` / `StringTokenizer`. Closed notes, no AI.
- Practice: read `scores.csv` (posted) and print each student's average — one line may contain a bad score. Handle it.
- Thursday: **integrated text-processing problems** — sentence capitalizer, password verifier, phone formatter, file of scores, Pokémon search.
- Course site: **wcu.ninja/csc240** — Class 4 code zip is posted.

Sources

- Gaddis, Starting Out with Java: From Control Structures through Data Structures, 4th ed., Ch. 9 (Tokenizing Strings, Wrapper Classes, StringBuilder).
- Java SE 17 API: String.split, java.util.StringTokenizer, java.util.Scanner (useDelimiter, hasNextInt), java.util.regex.Pattern.
- PokeAPI (pokeapi.co) response screenshot and Pokémon Viewer exercise: Dr. Si Chen, CSC 240 Spring 2019 Class 3–4 decks.
- Wrapper "can" illustration: Dr. Si Chen, CSC 240 Spring 2019 Class 2 deck.
- Stack Overflow, "2025 Developer Survey — AI" (66% "almost right" frustration; 45% debugging AI code time-consuming).
- OpenAI, "What are tokens and how to count them?" (≈4 characters per token for English text).