String API

- Represents a "string" of characters
  - Found in the `java.lang` package, imported by default
- <u>**Immutable**</u>
  - Once created, value will never change
  - All "modifying" methods (e.g. `.substring(), .trim()`) construct **copies**.
- Provides methods to manipulate Strings
  - Split up, take middle portion (substrings), trim whitespace, etc.

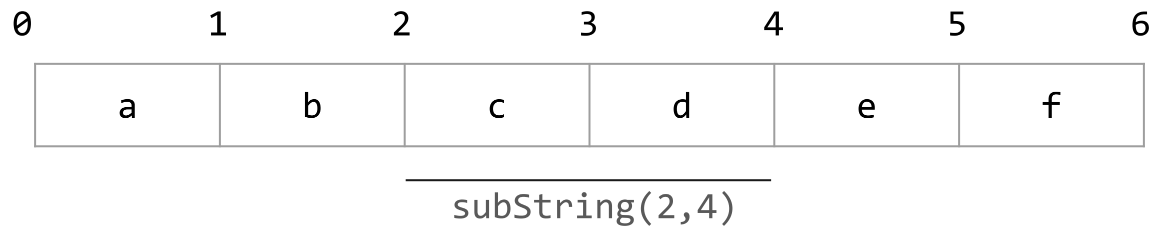
Copying may not be O(1)!

String API

Method Signature	Description	Runtime
`String[] split(String regex)`	Splits this string around matches of the given **regular expression**.	O(string-len)
`char charAt(int i)`	Returns the character value at the specified index (0-indexed).	O(1)
`String concat(String str)`	Concatenates the specified string to the end of this string. (Does not modify the original string, as they are immutable in Java.)	O(result-len)
`int length()`	Returns the length of this string.	O(1)
`String substring (int beginIndex, int endIndex)`	Returns a string that is a substring of this string. The substring begins at the specified `beginIndex` and extends to the character at index `endIndex - 1`.	O(result-len)

String API - Substrings

- `"abc".subString(2)` -> `"c"`.
  - `subString(n)` drops the first n characters/starts from the 0-indexed n-th character
- `"abcdef".subString(2,4)` -> `"cd"`
  - Starts at index 2, ends **BEFORE** index 4



String API - Substrings

- Make sure the string is long enough first
    - `"abc".subString(4)` will throw an error
        - 4 is past the end
    - `"abcdef".subString(7,9)` will throw an error
        - 7,9 also past the end
- Alternatively, first perform a test such as
    - `s.length() >= <expected-length>`
    - `s.startsWith(<some prefix>)`

String API: Equality and Comparison

Method Signature	Description	Runtime
`boolean equals(Object anObject)`	Compares this string to the specified object. (Normally used with a `String` parameter, to check if the values are the same. This is not the same as using `==`)	O(shorter-len)
`int compareTo(String other)`	Compares this string to the other String, in lexicographical (dictionary) ordering. ● this < other : Returns negative value ● this = other : Returns 0 ● this > other : Returns positive value	O(shorter-len)

String Equality: Common Mistakes

- String/Object equality in Java
  - `a == b` tests **reference** equality
    - If a and b point to the **same object in memory**
    - `"" == ("abc".substring(0,0))` (may) return false!
      - Even if both are “empty string” values
  - `a.equals(b)` tests **value** equality
    - If a and b have **equal values**
    - Or `Objects.equals(a, b)` if a is possibly null

Building a String

- Avoid repeated modifications (append/substrings) to Strings
  - e.g. Building a string in a loop
  - Java Strings are immutable
  - Every "modifying" operation has to allocate a new copy

    O(new_string_length) in both time/space
- If done in loop, can degrade to O(n^2)!

```
String s = "";
for(int i=0; i<n; i++) {
  s = s + "hello ";
}

T(n) = 6 + 12 + 18 + … 6n
     = 6(n^2+n)/2 ∈ O(n^2)
```

StringBuilder

- Solution (for repeated appends): Use StringBuilder
  - StringBuilders can be modified, without needing to reallocate
  - Only need to freeze at end (when converting toString())

```
String s = "";
for(int i=0; i<n; i++) {
  s = s + "hello ";
}     O(new_length) = O(i)


T(n) = 6 + 12 + 18 + … 6n
     = 6(n^2+n)/2 ∈ O(n^2)
```

```
StringBuilder s = new StringBuilder();
for(int i=0; i<n; i++) {
  s = s.append("hello ");
}     O(hello_length) = O(1)
String s = sb.toString();

T(n) = 6 + 6 + 6 + … + 6
     = 6n ∈ O(n)
```

StringBuilder

- Represents a mutable (modifiable) buffer of characters
  - i.e. a mutable `String`.
  - Cheap to append at rear
    - O(added_string_length) for `StringBuilder.append`

      O(new_string_length) for `String.concat` or `+`
  - Also cheap to delete near rear

Extra Note: Some of you may have seen `StringBuffer`.

It is a supposedly more “threadsafe” version of `StringBuilder` , the only difference being all methods are `synchronized`.

However, that isn’t actually useful for sequences of append operations, and so `StringBuffer` is effectively deprecated.

StringBuilder

Method Signature	Description	Runtime
`char charAt(int i)`	Returns the character at index i (0-based)	O(1)
`int length()`	Returns length of current string	O(1)
`StringBuilder append (String s)`	Adds `s` to the back of the stored string. Returns `this` for method chaining. (This method has various overloads for `int, char, Object`, etc.)	O(|s|) (amortized)
`String substring (int start, int end)`	Creates a new **immutable** string, with the current contents, in the range [start,end), in 0-based indexing. Similar to `String.substring(int, int)`.	O(end-start)
`String toString()`	Creates a new **immutable** string with the current contents. (Overrides `Object.toString()`.)	O(N)

Other possibly useful methods: `delete(int, int), deleteCharAt(int), reverse()`

StringBuilder

- Suppose we have an array of Strings
  - We want to add a line number to each of them
  - Then join them into a single String

```java
String str = ""; // empty string
for (int i = 0; i < arr.length; i++) {
    str = str + "Line " + i + ": "
            + arr[i] + "\n";
}
```

```java
StringBuilder sb = new StringBuilder();
for (int i = 0; i < arr.length; i++) {
    sb.append("Line ").append(i).append(": ")
        .append(arr[i]).append("\n");
}
String str = sb.toString();
```

Note the use of **method chaining**:
`sb.append(...)` returns `sb` itself, so you can call more append methods.

“Just Print It Out”

- Solution (if directly printing output):
  - Immediately print out items, instead of building an output String
  - `System.out/BufferedWriter` will handle all of it for you
- Not always applicable.
  - Only if no need to manipulate the string further

Useful APIs: Scanner

- `Scanner` class – used for reading **input**
- Found in the `java.util` package
  - Import with `import java.util.*;`
- Declare a new Scanner object:
  - `Scanner sc = new Scanner(System.in);`
    Constructs a `Scanner` wrapping standard input.
  - **<u>DO NOT</u> construct multiple `Scanners`!**
- Read in input using the methods found in Scanner:
  - `int testCases = sc.nextInt();`
  - `double length = sc.nextDouble();`
  - `String singleWord = sc.next();`
  - `String wholeLine = sc.nextLine();`

Scanner API

"token" roughly means "word"
(surrounded by whitespace)

Method Signature	Description	Runtime
`int nextInt()`	Scans the next token of the input as an `int`. (Reads the next word of the input as an `int`.)	O(N)
`double nextDouble()`	Scans the next token of the input as a `double`. (Reads the next word of the input as an `double`.)	O(N)
`String next()`	Finds and returns the next complete token from this scanner. (Reads the next word of the input as a `String`.)	O(N)
`String nextLine()`	Advances this scanner past the current line and returns the input that was skipped. (Reads until it reaches the end of the current line.)	O(N)

N refers to the length of the input that is read.

Slides covering API will cover the most frequently used (but not all) methods of a class.

Scanner: Common Mistakes

- `Scanner` has 2 "ways to read"
  - Token/word-based:

    `nextInt()`, `nextDouble()`, `next()`, etc.

    Reads word up to next **whitespace** character.
  - Line-based:

    `nextLine()`

    Reads all the way up to next **newline**.
- Word-based & line-based don't mix well

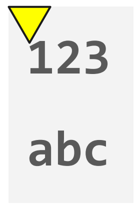
Scanner: Mixing Token- & Line-based Methods

- Let’s say we are given the input:

```
123
abc
```

Scanner: Mixing Token- & Line-based Methods

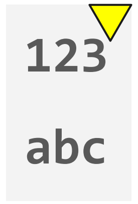
- We construct a `Scanner` around standard input.
- Initially, the cursor is right before '**`1`**'.



```
123
abc
```

Scanner: Mixing Token- & Line-based Methods

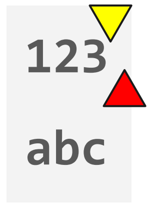
- Calling `nextInt()` reads a word `"123"`
  - Results in `123`, an `int`
- Now, the cursor is at the end of the first line.

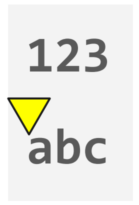


```
123
abc
```

Scanner: Mixing Token- & Line-based Methods

- Calling `nextLine()` now would give me the **remainder** of the line
  - i.e. the empty string
  - Not the next line `"abc"`
- Now, the cursor is at the start of the second line:





In general, after calling a word-based `next*()` method, before we start reading lines:

You can clean up the remainder of the line with a `nextLine()` call, which you ignore the result of.

Useful APIs: PrintStream (i.e. System.out)

- `PrintStream` class – used for writing **output**
  - Found in the `java.io` package
- Use the existing `System.out`

  Already a `PrintStream` wrapping standard output.
  - No need to create a separate one *for now*
- Write output using the methods:
  - `System.out.println(100); // Prints "100", and then a new line`
  - `System.out.print("asd"); // Prints "asd" and STAYS on the same line`
  - `System.out.println();    // Prints just a new line`
  - `System.out.printf("5 + 5 = %d\n", 5+5);`

    `// Prints "5 + 5 = 10", followed by a new line`

PrintStream (i.e. System.out)

Method Signature	Description	Runtime
`System.out.print(String str)`	Prints a `String`.	O(N)
`System.out.println(String str)`	Prints a `String` and then terminate the line. (Prints a `String`, followed by a newline character '\n')	O(N)
`System.out.printf(String str)`	(Emulates the `printf` function in C programming language.)	O(N)

N refers to the length of the output.

`System.out` is an instance of the `PrintStream` class.

You can refer to the API documentation on `PrintStream` to explore more methods.

Common Mistakes: Input Format

- Ensure when you test
  - What you key in, matches the input format **exactly.**
  - Your output **exactly** matches the expected output.
    - A missing punctuation mark may be tiny, but makes all the difference.
    - An extra space or newline usually is tolerated

Common Mistakes: Scanner

- `nextInt()/nextDouble()/next()`
  - Token/word-based
  - Up to next whitespace
- `nextLine()`
  - Line-based
  - Up to next newline
- Token/word-based methods may leave leftover bits of the current line

Scanner

After reading a word/token, there may be remainder of line left over.

This `nextLine()` call will read the **remainder of the current line**, not the following line.

```java
Scanner s = new Scanner(System.in);
int n = s.nextInt();

for(int i=0; i<n; i++) {
     int k = s.nextInt(); // s.nextLine();

     String name = s.nextLine();

     for(int j=0; j<k; j++) {
          String item = s.nextLine();
          /* snip */
     }
}
```

Common Mistakes: String Equality

- String/Object equality in Java
  - `a == b` tests **reference** equality
    - If a and b point to the **same object in memory**
    - `"" == ("abc".substring(0,0))` (may) return false!
      - Even if both are “empty string” values
  - `a.equals(b)` tests **value** equality
    - If a and b have **equal values**
    - Or `Objects.equals(a, b)` if a is possibly null

String Equality

== compares exact String objects!
May not return true, even if same value.

```java
Scanner s = new Scanner(System.in);
int n = s.nextInt();

for(int i=0; i<n; i++) {
    int k = s.nextInt(); s.nextLine();

    String name = s.nextLine();

    for(int j=0; j<k; j++) {
        String item = s.nextLine();
        if("potato" == item) {
            /* snip */
        }
    }
}
```

Non-Buffered IO & Large Inputs/Outputs

- `Scanner`
  - Easy to use, but is quite slow (due to use of regexes)
- `System.out.print*`
  - Will immediately give the value to the OS to display (i.e. unbuffered)
  - May use up a lot of time if called repeatedly

Buffered IO

- Faster but more complicated IO methods exist
- Some take-home assignment requires buffered/“fast” IO
  - Using `Scanner/System.out` will result in exceeding time limit
  - Rough rule of thumb:
    - If you are reading/writing in 10^5-10^6 words/characters/things
    - Then you probably want fast IO

BufferedReader API

- Provides a more efficient way for reading input (input **buffering**)
  - Non-buffered:
    - Every time you read some short word
    - Request from OS, for one short chunk each time
  - Buffered:
    - Request from OS, one large chunk/**buffer** at once
    - Slice it up word by word when needed

BufferedReader API

- Found in `java.io` package, need to use following line to import

  ```
  import java.io.BufferedReader;
  import java.io.InputStreamReader;
  ```

- Declare a new `BufferedReader` object in main method

  ```
  BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
  ```

  Wraps `System.in`, to provide buffering functionality.

- Read in input using methods found in `BufferedReader`

BufferedReader API

Method Signature	Description	Runtime
`String readLine()`	Reads a line of text. (Reads until it reaches the end of line.)	O(N)

Other methods exist but may not be as useful.

Can combine with `String.split` to obtain “words”.

PrintWriter/BufferedWriter API

- Provides a faster way for writing output
- Found in `java.io` package, need to use following line to import

```
import java.io.PrintWriter;
import java.io.BufferedWriter;
import java.io.OutputStreamWriter;
```

- Declare a new `PrintWriter` object in main method

```
PrintWriter pw = new PrintWriter(new BufferedWriter(
    new OutputStreamWriter(System.out) ));
```

Wraps `System.out`, to provide buffering and printing functionalities.

PrintWriter API

Method Signature	Description	Runtime
`void print(String s)`	Prints a string.	O(N)
`void println(String s)`	Prints a string and then terminates the line (with '`\n`').	O(N)
`void printf(String s)`	(Emulates the function in C programming language.)	O(N)
`void flush()`	Flush the stream. (Prints the current content of the writer to the screen)	O(1)
`void close()`	Closes the stream and releases any system resources associated with it. (Calls `flush()`, then closes the writer. The writer cannot be used again.)	O(1)

Slides covering API will cover the most frequently used (but not all) methods of a class.

PrintWriter/BufferedWriter API

- Used the same way as System.out
    - Delays printing until a `flush()` or `close()` method is called
    - Avoid repeated switching between printing and computation
    - Saves time
- **Always call `flush()` or `close()` on the PrintWriter before exiting your program**
    - If not, some output may not be printed

Safe Flushing/Closing

```java
/* Unsafe (may lose last part of output) */

PrintWriter out = new PrintWriter(
    new BufferedWriter(
        new OutputStreamWriter(System.out)
    )));

// Program code...
out.close();
```

```java
/* Try-with-resources */

try(PrintWriter out = new PrintWriter(
    new BufferedWriter(
        new OutputStreamWriter(System.out)
    ))) {
    // Program code...
}
```

```java
/* Try-finally */

PrintWriter out = null;
try {
    out = new PrintWriter(new BufferedWriter(
        new OutputStreamWriter(System.out)
    ));

    // Program code...
} finally {
    if(out != null) {
        out.close();
    }
}
```

Kattio

- Pre-packaged version of all the stuff in the previous slides
- For input, it provides its own methods (next slide)
- For output, it uses the same methods as PrintWriter (previous slide)
    - Remember to flush/close at end!
- Available at https://github.com/kattis/kattio
    - If used, submit **only** your source code (without Kattio.java).

Kattio.java API

Method Signature	Description	Runtime
`int getInt()`	Reads the next token in the input as an integer	O(N)
`long getLong()`	Reads the next token in the input as a long	O(N)
`double getDouble()`	Reads the next token in the input as a double	O(N)
`String getWord()`	Reads the next token in the input as a string	O(N)

Output methods are inherited from PrintWriter.

NOTE: No line-based methods. (If needed, use BufferedReader directly.)

Safe Flushing/Closing (Kattio)

```
/* Unsafe (may lose last part of output) */

Kattio io = new Kattio(System.in);

// Program code…

out.close();
```

```
/* Try-with-resources */

try(Kattio io = new Kattio(System.in)) {
    // Program code...
}
```

```
/* Try-finally */

Kattio io = null;
try {
    io = new Kattio(System.in)

    // Program code...
} finally {
    if(io != null) {
        io.close();
    }
}
```