Unit 3, Part 2

Using Objects from Existing Classes

Computer Science S-111
Harvard University

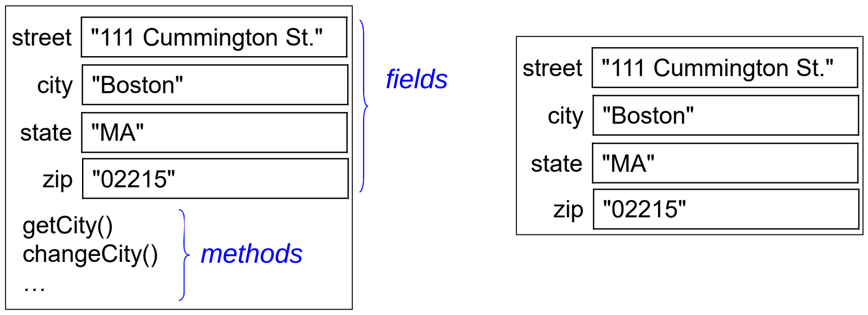
David G. Sullivan, Ph.D.

Combining Data and Operations

- The data types that we've seen thus far are referred to as *primitive* data types.
  - `int`, `double`, `char`
  - several others
- Java allows us to use another kind of data known as an *object*.
- An object groups together:
  - one or more data values (the object's *fields*)
  - a set of operations (the object's *methods*)
- Objects in a program are often used to model real-world objects.

Combining Data and Operations (cont.)

- Example: an `Address` object
  - possible fields: *street*, *city*, *state*, *zip*
  - possible operations: *get the city*, *change the city*, *check if two addresses are equal*
- Here are two ways to visualize an `Address` object:



Classes as Blueprints

- We've been using classes as containers for our programs.
- A class can also serve as a blueprint – as the definition of a new type of object.
- The objects of a given class are built according to its blueprint.
- Another analogy:
  - class = cookie cutter
    objects = cookies
- The objects of a class are also referred to as *instances* of the class.

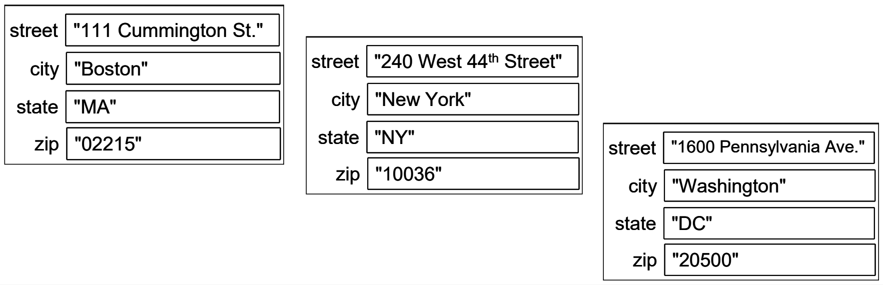
Class vs. Object

- The `Address` class is a blueprint:

```
public class Address {
    // definitions of the fields
    ...

    // definitions of the methods
    ...
}
```

- `Address` objects are built according to that blueprint:



Using Objects from Existing Classes

- Later in the course, you'll learn how to create your own classes that act as blueprints for objects.
- For now, we'll focus on learning how to use objects from existing classes.

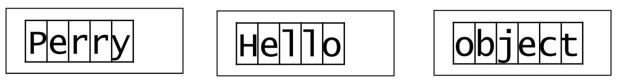
`String` Objects

- In Java, a string (like `"Hello, world!"`) is actually represented using an object.
  - data values: the characters in the string
  - operations: get the length of the string, get a substring, etc.
- The `String` class defines this type of object:

```
public class String {
    // definitions of the fields
    ...

    // definitions of the methods
    ...
}
```

- Individual `String` objects are instances of the `String` class:



Variables for Objects

- When we use a variable to represent an object, the type of the variable is the name of the object's class.
- Here's a declaration of a variable for a `String` object:

```
String name;
```

type (the class name)   *variable name*

  - we capitalize `String`, because it's a class name

Creating `String` Objects

- One way to create a `String` object is to specify a string literal:

  ```
  String name = "Perry Sullivan";
  ```

- We create a new `String` from existing `String`s when we use the + operator to perform concatenation:

  ```
  String firstName = "Perry";
  String lastName = "Sullivan";
  String fullName = firstName + " " + lastName;
  ```

- Recall that we can concatenate a `String` with other types of values:

  ```
  String msg = "Perry is " + 6;

  // msg now represents "Perry is 6"
  ```

Using an Object's Methods

- An object's methods are different from the static methods that we've seen thus far.
  - they're called *non-static* or *instance* methods
- An object's methods *belong to* the object. They specify the operations that the object can perform.
- To use a non-static method, we have to specify the object to which the method belongs.
  - use *dot notation*, preceding the method name with the object's variable:

  ```
  String firstName = "Perry";
  int len = firstName.length();
  ```

- Using an object's method is like sending a message to the object, asking it to perform that operation.

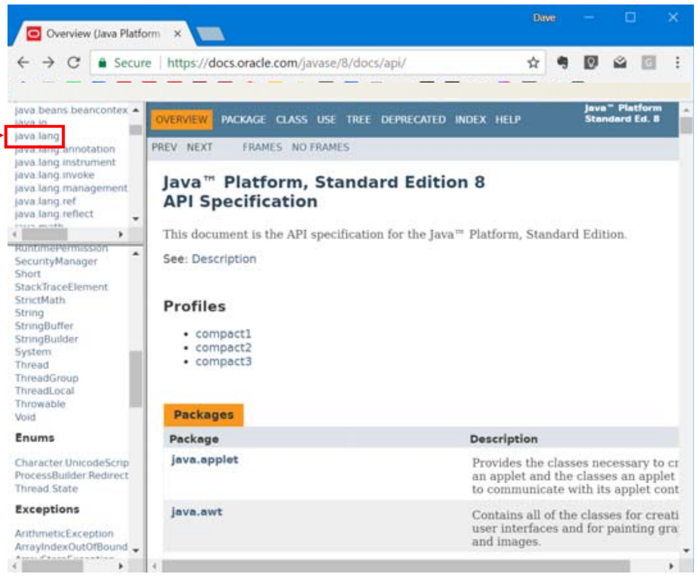
The API of a Class

- The methods defined within a class are known as the *API* of that class.
  - API = application programming interface
- We can consult the API of an existing class to determine which operations are supported.
- The API of all classes that come with Java is available here:
  `https://docs.oracle.com/javase/8/docs/api/`
  - there's a link on the resources page of the course website

Consulting the Java API

select the package name (optional)

`String` is in `java.lang`



Java™ Platform, Standard Edition 8
API Specification

This document is the API specification for the Java™ Platform, Standard Edition.

See: Description

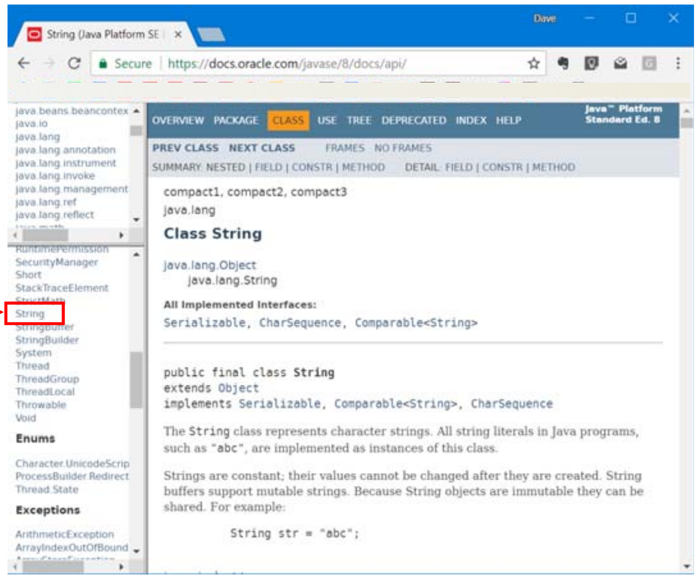
Profiles

- compact1
- compact2
- compact3

Packages

Package	Description
java.applet	Provides the classes necessary to cr an applet and the classes an applet to communicate with its applet cont
java.awt	Contains all of the classes for creati user interfaces and for painting gra and images.

Consulting the Java API



compact1, compact2, compact3
java.lang

Class String

java.lang.Object
    java.lang.String

All Implemented Interfaces:
Serializable, CharSequence, Comparable<String>

```
public final class String
extends Object
implements Serializable, Comparable<String>, CharSequence
```

The String class represents character strings. All string literals in Java programs, such as "abc", are implemented as instances of this class.

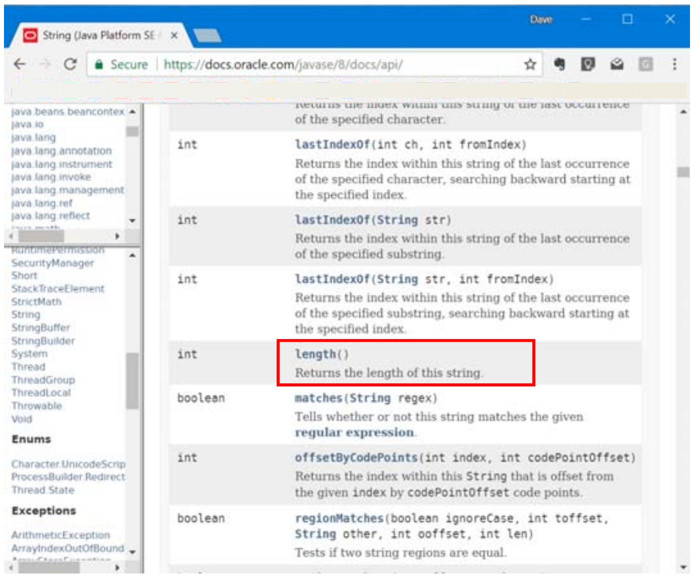
Strings are constant; their values cannot be changed after they are created. String buffers support mutable strings. Because String objects are immutable they can be shared. For example:

```
String str = "abc";
```

select the class name

Consulting the Java API (cont.)

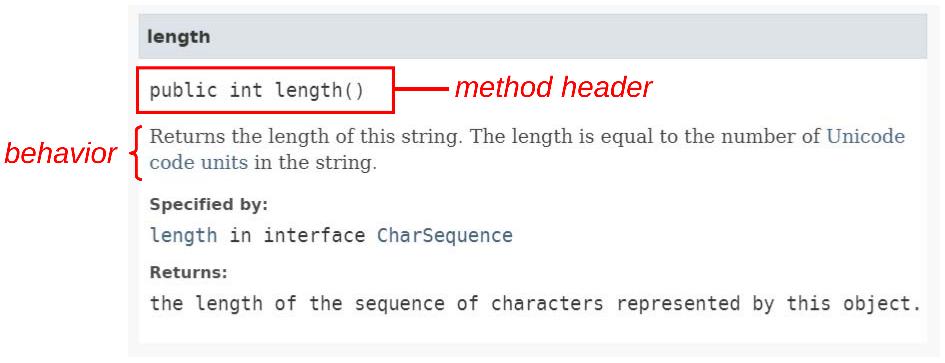
- Scroll down to see a summary of the available methods:



	Returns the index within this string of the last occurrence of the specified character.
int	lastIndexOf(int ch, int fromIndex) Returns the index within this string of the last occurrence of the specified character, searching backward starting at the specified index.
int	lastIndexOf(String str) Returns the index within this string of the last occurrence of the specified substring.
int	lastIndexOf(String str, int fromIndex) Returns the index within this string of the last occurrence of the specified substring, searching backward starting at the specified index.
int	length() Returns the length of this string.
boolean	matches(String regex) Tells whether or not this string matches the given **regular expression**.
int	offsetByCodePoints(int index, int codePointOffset) Returns the index within this String that is offset from the given index by codePointOffset code points.
boolean	regionMatches(boolean ignoreCase, int toffset, String other, int ooffset, int len) Tests if two string regions are equal.

Consulting the Java API (cont.)

- Clicking on a method name gives you more information:



length

```
public int length()
```

Returns the length of this string. The length is equal to the number of Unicode code units in the string.

Specified by:

`length` in interface `CharSequence`

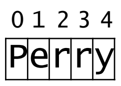
Returns:

`the length of the sequence of characters represented by this object.`

- From the header, we can determine:
  - the return type: `int`
  - the parameters we need to supply: the empty `()` indicates that `length` has no parameters

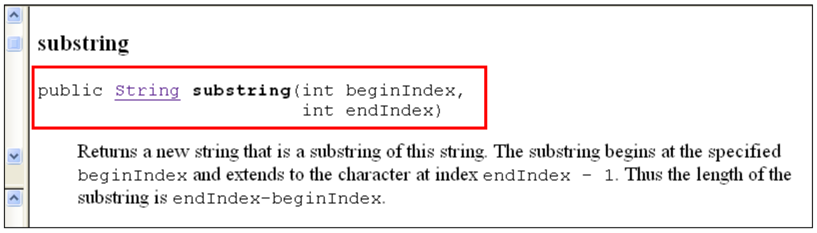
Numbering the Characters in a `String`

- The characters are numbered from left to right, starting from 0.



- The position of a character in a string is known as its *index*.
  - `'P'` has an index of 0 in `"Perry"`
  - `'y'` has an index of 4

`substring` Method



substring

```
public String substring(int beginIndex,
                        int endIndex)
```

Returns a new string that is a substring of this string. The substring begins at the specified `beginIndex` and extends to the character at index `endIndex - 1`. Thus the length of the substring is `endIndex-beginIndex`.

<u>`String substring(int beginIndex, int endIndex)`</u>

- return type: ?
- parameters: ?
- behavior: returns the substring that:
  - begins at `beginIndex`
  - ends at `endIndex` – 1

`substring` Method (cont.)

- To extract a substring of length *N*, you can just figure out `beginIndex` and do:

  `substring(beginIndex, beginIndex + `*N*`)`

  - example: consider again this string:

    ```
    String name = "Perry Sullivan";
    ```

    To extract a substring containing the first 5 characters, we can do this:

    ```
    String first = name.substring(0, 5);
    ```

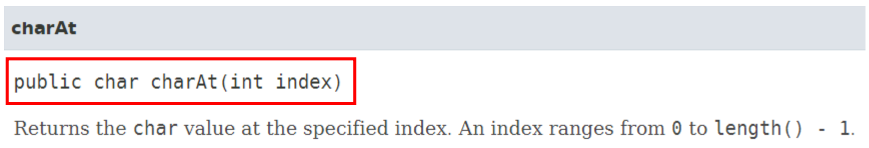
Review: Calling a Method

- Consider this code fragment:

```
String name = "Perry Sullivan";
int start = 6;
String last = name.substring(start, start + 8);
```

- Steps for executing the method call:
  1. the actual parameters are evaluated to give:
     ```
     String last = name.substring(6, 14);
     ```
  2. a frame is created for the method, and the actual parameters are assigned to the formal parameters
  3. flow of control jumps to the method, which creates and returns the substring `"Sullivan"`
  4. flow of control jumps back, and the returned value replaces the method call:
     ```
     String last = "Sullivan";
     ```

How should we fill in the blank?



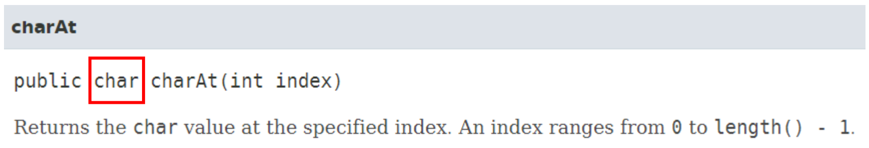
charAt

`public char charAt(int index)`

Returns the `char` value at the specified index. An index ranges from `0` to `length() - 1`.

```
String s = "Strings have methods inside them!";
int len = s.length();
____________________  // get the last character in s
```

charAt Method


charAt

`public char charAt(int index)`

Returns the `char` value at the specified index. An index ranges from `0` to `length() - 1`.

- The `charAt()` method that we use for indexing returns a `char`, not a `String`.
- We have to be careful when we use its return value!
  - example: what does this print?

```
String name = "Perry Sullivan";
System.out.println(name.charAt(0) +
   name.charAt(6));
```

charAt Method

- Here's how we can fix this:

```
String name = "Perry Sullivan";
System.out.println(name.charAt(0) + "" +
   name.charAt(6));
                ⬇

System.out.println('P' + "" +
   'S');
                ⬇

System.out.println("PS");
```

Another `String` Method

`String toUpperCase()`
returns a new `String` in which all of the letters in the original `String` are converted to upper-case letters

- Example:

```
String warning = "Start the problem set ASAP!";
System.out.println(warning.toUpperCase());
                                  ↓
System.out.println("START THE PROBLEM SET ASAP!");
```

- `toUpperCase()` creates and returns a new `String`.
  It does *not* change the original `String`.
- In fact, it's *never* possible to change an existing `String` object.
- We say that `String`s are *immutable* objects.

`indexOf` Method

`int indexOf(char ch)`

- return type: `int`
- parameter list: `(char ch)`
- returns:
  - the index of the first occurrence of `ch` in the string
  - -1 if the `ch` does not appear in the string
- examples:

```
String name = "Perry Sullivan";
System.out.println(name.indexOf('r'));
System.out.println(name.indexOf('X'));
```

The Signature of a Method

- The *signature* of a method consists of:
  - its name
  - the number and types of its parameters

$$\texttt{public String }\underbrace{\texttt{substring(int beginIndex, int endIndex)}}_{\textit{the signature}}$$

- A class cannot include two methods with the same signature.

Two Methods with the Same Name

- There are actually two `String` methods named `substring`:

  ```
  String substring(int beginIndex, int endIndex)
  ```

  ```
  String substring(int beginIndex)
  ```

  - returns the substring that begins at `beginIndex` and continues to the end of the string
- Do these two methods have the same signature?
- Giving two methods the same name is known as *method overloading*.
- When you call an overloaded method, the compiler uses the number and types of the actual parameters to figure out which version to use.

Console Input Using a `Scanner` Object

- We've been printing text in the console window.
- You can also ask the user to enter a value in that window.
  - known as console input
- To do so, we use a type of object known as a `Scanner`.
  - recall PS 2

Packages

- Java groups related classes into *packages*.
- Many classes are part of the `java.lang` package.
  - examples: `String`, `Math`
  - We don't need to tell the compiler where to find these classes.
- If a class is in another package, we need to use an `import` statement so that the compiler will be able to find it.
  - put it *before* the definition of the class
- The `Scanner` class is in the `java.util` package, so we do this:

```
import java.util.*;

public class MyProgram {
    ...
```

Creating an Object

- `String` objects are different from other objects, because we're able to create them using literals.
- To create an object, we typically use a special method known as a *constructor*.
- Syntax:

```
<variable> = new <ClassName>(<parameters>);
or
<type> <variable> = new <ClassName>(<parameters>);
```

- To create a `Scanner` object for console input:

```
Scanner console = new Scanner(System.in);
```

the parameter tells the constructor that we want the `Scanner` to read from the *standard input* (i.e., the keyboard)

`Scanner` Methods: A Partial List

- `String next()`
  - read in a single "word" and return it
- `int nextInt()`
  - read in an integer and return it
- `double nextDouble()`
  - read in a floating-point value and return it
- `String nextLine()`
  - read in a "line" of input (could be multiple words) and return it

Example of Using a `Scanner` Object

- To read an integer from the user:

```
Scanner console = new Scanner(System.in);
int numGrades = console.nextInt();
```

- The second line causes the program to pause until the user types in an integer followed by the [ENTER] key.
- If the user only hits [ENTER], it will continue to pause.
- If the user enters an integer, it is returned and assigned to `numGrades`.
- If the user enters a non-integer, an exception is thrown and the program crashes.

Example Program: `GradeCalculator`

```
import java.util.*;

public class GradeCalculator {
   public static void main(String[] args) {
        Scanner console = new Scanner(System.in);

        System.out.print("Points earned: ");
        int points = console.nextInt();
        System.out.print("Possible points: ");
        int possiblePoints = console.nextInt();

        double grade = points/(double)possiblePoints;
        grade = grade * 100.0;

        System.out.println("grade is " + grade);
    }
}
```

Important Note About Console Input

- When writing an interactive program that involves user input in methods other than `main`, you should:
  - *create a **single*** `Scanner` *object in **the first line** of the* `main` *method*
  - pass that object into any other method that needs it
- This allows you to avoid creating multiple objects that all do the same thing.
- It also facilitates our grading, because it allows us to provide a series of inputs using a file instead of the keyboard.

Important Note About Console Input (cont.)

- Example:

```
public class MyProgram {
    public static void main(String[] args) {
        Scanner console = new Scanner(System.in);
        String str1 = getString(console);
        String str2 = getString(console);
        System.out.println(str1 + " " + str2);
    }

    public static String getString(Scanner console) {
        System.out.print("Enter a string: ");
        String str = console.next();
        return str;
    }
}
```

What's Wrong with the Following?

```
import java.util.*;

public class LengthConverter {
    public static void main(String[] args) {
        Scanner console = new Scanner(System.in);
        int cm = (int)(getInches(console) * 2.54);
        System.out.println(getInches(console)
          + " inches = " + cm + " cm");
    }

    public static int getInches(Scanner console) {
        System.out.print("Enter a length in inches: ");
        int inches = console.nextInt();
        return inches;
    }
}
```

Exercise: Analyzing a Name: First Version

```
public class NameAnalyzer {
    public static void main(String[] args) {
        String name = "Perry Sullivan";
        System.out.println("full name = " + name);

        int length = name.length();
        System.out.println("length = " + length);

        String first = name.substring(0, 5);
        System.out.println("first name = " + first);

        String last = name.substring(6);
        System.out.println("last name = " + last);
    }
}
```

Making the Program More General

- Would the code work if we used a different name?

```
import java.util.*;

public class NameAnalyzer {
    public static void main(String[] args) {
        Scanner console = new Scanner(System.in);
        String name = console.nextLine();
        System.out.println("full name = " + name);

        int length = name.length();
        System.out.println("length = " + length);

        String first = name.substring(0, 5);
        System.out.println("first name = " + first);

        String last = name.substring(6);
        System.out.println("last name = " + last);
    }
}
```

Breaking Up a Name

- Given a string of the form "*firstName lastName*", how can we get the first and last names, without knowing how long it is?
- Pseudocode for what we need to do:
- What `String` methods can we use? Consult the API!
- Code:

Static Methods for Breaking Up a Name

- How could we rewrite our name analyzer to use separate methods for extracting the first and last names?

```
public static _________ firstName(_______________) {




}

public static _________ lastName(_______________) {




}
```

Using the Static Methods

- Given the methods from the previous slide, what would the `main` method now look like?

```
public static void main(String[] args) {
    Scanner console = new Scanner(System.in);
    String name = console.nextLine();
    System.out.println("full name = " + name);

    int length = name.length();
    System.out.println("length = " + length);






}
```

Processing a String One Character at a Time

- Write a method for printing the name vertically, one char per line.

```
import java.util.*;

public class NameAnalyzer {
    public static void main(String[] args) {
        Scanner console = new Scanner(System.in);
        String name = console.nextLine();
        System.out.println("full name = " + name);
        ...
        printVertical(name);
    }

    public static _____ printVertical(_______________){

        for (int i = 0; i < _______________; i++) {



        }
    }
}
```

Scanner Objects and Tokens

- Most `Scanner` methods read one *token* at a time.
- Tokens are separated by whitespace (spaces, tabs, newlines).
  - example: if the user enters the line

    ```
    wow, I slept for 9 hours!\n
    ```

    newline character, which you get when you hit [ENTER]

    there are six tokens:
    - `wow,`
    - `I`
    - `slept`
    - `for`
    - `9`
    - `hours!`

`Scanner` Objects and Tokens (cont.)

- Consider the following lines of code:

```
System.out.print("Enter the length and width: ");
int length = console.nextInt();
int width = console.nextInt();
```

- Because the `nextInt()` method reads one token at a time, the user can either:
  - enter the two numbers on the same line, separated by one or more whitespace characters

    ```
    Enter the length and width: 30 15
    ```

  - enter the two numbers on different lines

    ```
    Enter the length and width: 30
    15
    ```

`nextLine` Method

- The `nextLine()` method does not just read a single token.
- Using `nextLine` can lead to unexpected behavior, for reasons that we'll discuss later on.
- Avoid it for now!

Additional Terminology

- To avoid having too many new terms at once, I've limited the terminology introduced in these notes.
- Here are some additional terms related to classes, objects, and methods:
  - *invoking* a method = calling a method
  - method *invocation* = method call
  - the *called object* = the object used to make a method call
  - *instantiate* an object = create an object
  - *members* of a class = the fields and methods of a class