CIS 110: Introduction to Computer Programming

Lecture 12
Authoring Solid Helper Methods
(§ 4.4)

Outline

- Authoring Helper Methods
  – Pre- and post-conditions
  – Exceptions

System.out.printf

- An alternative to println/print that lets you *format* the output.

A *format specifier*. A placeholder for a thing to print.

```
System.out.printf("Example of printf: %d %.2f %s",
                  12, 1.241, "Chowder");
> Example of printf: 12 1.24 Chowder
```

Specifiers have the form: %<formatting><type>

Need to provide one argument per format specifier. They are consumed in-order.

See p. 260 of the book for more information about format specifiers.

Helper Methods

See PalindromeChecker.java

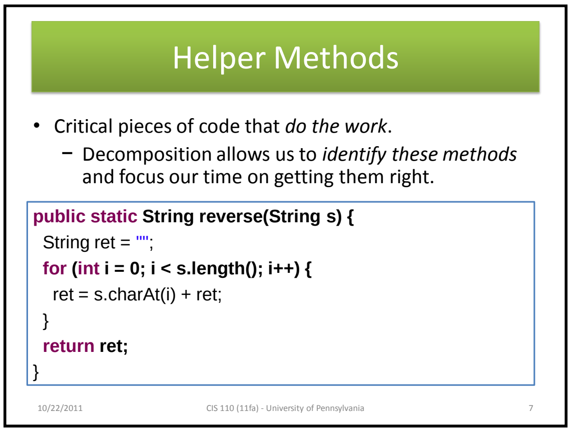
Sample Problem

- Problem: write a program that reads in a String from the user, checks to see if that String is a palindrome, and informs the user of the results of the check.

```
Example output
> Enter a string to check:
> abba
> The reverse of the line is: abba
> The line is a palindrome!
```

Our Methodology

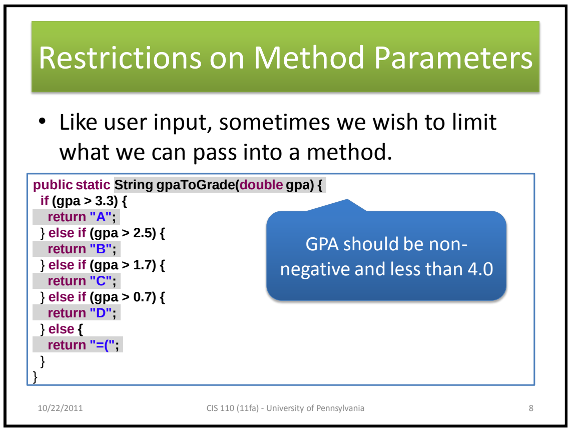
1. Try some example inputs to get a feel for the problem.
2. Start with a skeleton of the solution.
3. Decompose the problem into sub-problems.
4. Make helper methods to solve the sub-problems.
5. Use those helper methods to solve your main problem.



Helper Methods

- Critical pieces of code that *do the work*.
  - Decomposition allows us to *identify these methods* and focus our time on getting them right.

```
public static String reverse(String s) {
  String ret = "";
  for (int i = 0; i < s.length(); i++) {
    ret = s.charAt(i) + ret;
  }
  return ret;
}
```

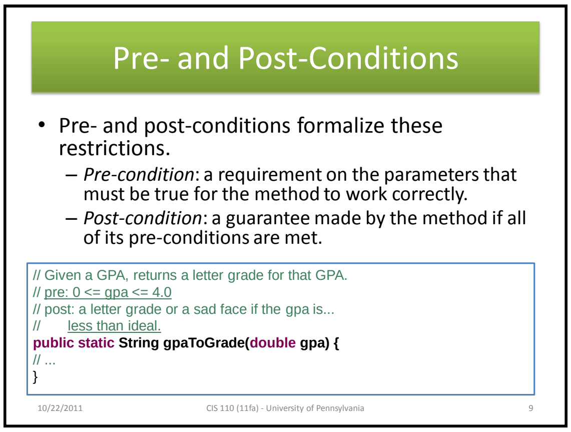


Restrictions on Method Parameters

- Like user input, sometimes we wish to limit what we can pass into a method.

```
public static String gpaToGrade(double gpa) {
  if (gpa > 3.3) {
    return "A";
  } else if (gpa > 2.5) {
    return "B";
  } else if (gpa > 1.7) {
    return "C";
  } else if (gpa > 0.7) {
    return "D";
  } else {
    return "=(";
  }
}
```

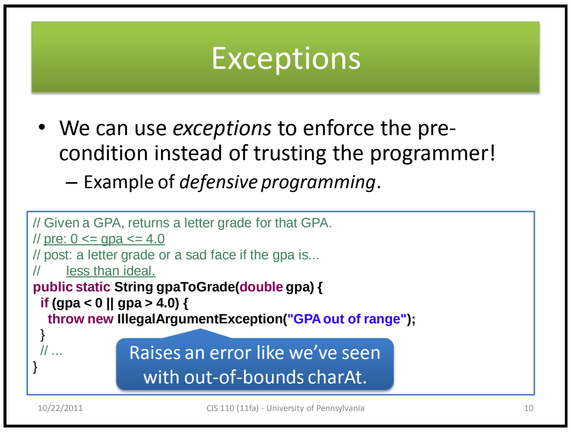
GPA should be non-negative and less than 4.0



Pre- and Post-Conditions

- Pre- and post-conditions formalize these restrictions.
  - *Pre-condition*: a requirement on the parameters that must be true for the method to work correctly.
  - *Post-condition*: a guarantee made by the method if all of its pre-conditions are met.

```
// Given a GPA, returns a letter grade for that GPA.
// pre: 0 <= gpa <= 4.0
// post: a letter grade or a sad face if the gpa is...
//       less than ideal.
public static String gpaToGrade(double gpa) {
// ...
}
```

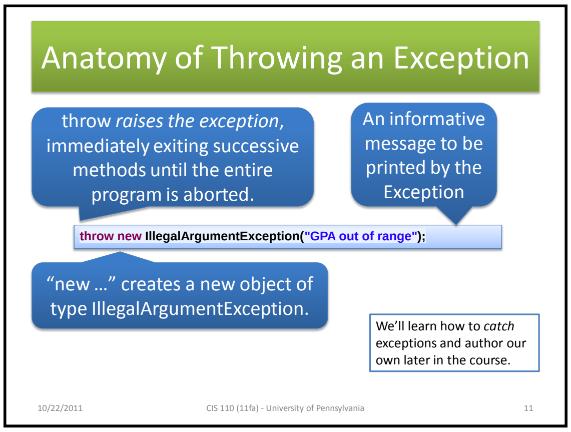


Exceptions

- We can use *exceptions* to enforce the pre-condition instead of trusting the programmer!
  - Example of *defensive programming*.

```
// Given a GPA, returns a letter grade for that GPA.
// pre: 0 <= gpa <= 4.0
// post: a letter grade or a sad face if the gpa is...
//       less than ideal.
public static String gpaToGrade(double gpa) {
  if (gpa < 0 || gpa > 4.0) {
    throw new IllegalArgumentException("GPA out of range");
  }
  // ...
}
```

Raises an error like we've seen with out-of-bounds charAt.



Anatomy of Throwing an Exception

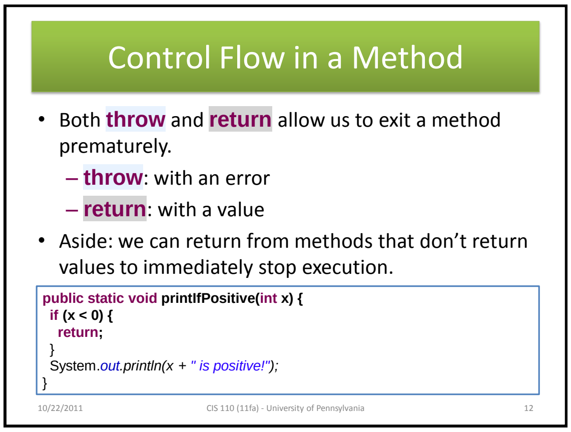
throw *raises the exception*, immediately exiting successive methods until the entire program is aborted.

An informative message to be printed by the Exception

```
throw new IllegalArgumentException("GPA out of range");
```

"new …" creates a new object of type IllegalArgumentException.

We'll learn how to *catch* exceptions and author our own later in the course.



Control Flow in a Method

- Both **throw** and **return** allow us to exit a method prematurely.
  - **throw**: with an error
  - **return**: with a value
- Aside: we can return from methods that don't return values to immediately stop execution.

```
public static void printIfPositive(int x) {
  if (x < 0) {
    return;
  }
  System.out.println(x + " is positive!");
}
```