CIS 110: Introduction to Computer Programming

Lecture 3

Express Yourself

(§ 2.1)

Outline

1. Data representation and types
2. Expressions

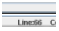
Administrivia

- http://www.cis.upenn.edu/~cis110
- Sign up for Piazza!
- New lab section: Lab 214, Th 5-6 PM.
- Last call for move/swap/register requests.
- Lab assignment #1: due at the start of lab.
- HW #1 out: due next Monday (online).
- A note on feeling lost and help!

Homework 1

- Reproduce song lyrics that have a certain structure to them.
  - *Capture structure* and *eliminate redundancy*.
- Only use classes, static methods, and printlns
- We grade on *correctness* and *design*.
  - Correctness: “Does your output match *exactly* with the desired output from the write-up?”
  - Design: “Is your solution well-designed?”

Homework Design Guidelines

- Does your code meet the design goals stated in the write-up?
  - HW 1: did you capture structure and eliminate redundancy as much as reasonably possible?
- Does your code meet our style guidelines?
  - Consistent indentation, naming, etc.
  - Method comments, file-header comment.
  - No “work” done directly in `main`.
  - 80 characters at most per line. 

    - Standard with historical roots: 80 line terminals!
  - Good style is like flossing!

Data Representation and Types

The Digital Realm

- Computers store data as sequences of bits
  - Bits are just 0s and 1s
  - E.g., 0101 1101 could be
    - The *integer* 93 (interpreted as a *binary integer)*
    - The *real number* 1.3*e^{-43} (interpreted as an *IEEE 754 floating point number*)
    - The *character* ‘]’ (interpreted as a *Unicode character*)
- How do we know how to interpret a series of bits stored in memory?

A Type for Every Datum

- *Types* distinguish between different interpretations of data.
  - Interpreting 0101 1101 as a
    - `int` gives us the integer 93.
    - `double` gives us the floating-point number 1.3*e^{-43}.
    - `char` gives us the character ‘]’.
- `int`, `double`, and `char` are *primitive types.*
  - Other primitive types: `boolean`, `byte`, `float`.
    - We’ll talk about `boolean` later, ignore the rest.

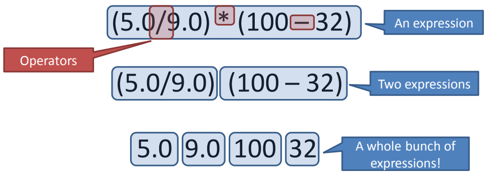
Java is a High-level Language

- With Java, we rarely (if ever) need to deal with data at the level of 1s and 0s.
  - We work with `int`s, `double`s, `char`s, directly.
    - 93, 1.3*e^{-43}, and ‘]’ instead of 0101 1101.
- However, data representation still influences the behavior of some operations…!

Expressions

What is an Expression?

- An expression is a *value* or a set of operators *that produces a value* that your program can use
  - e.g., an arithmetic calculation



(5.0/9.0) * (100 – 32)

An expression

Operators

(5.0/9.0) (100 – 32)

Two expressions

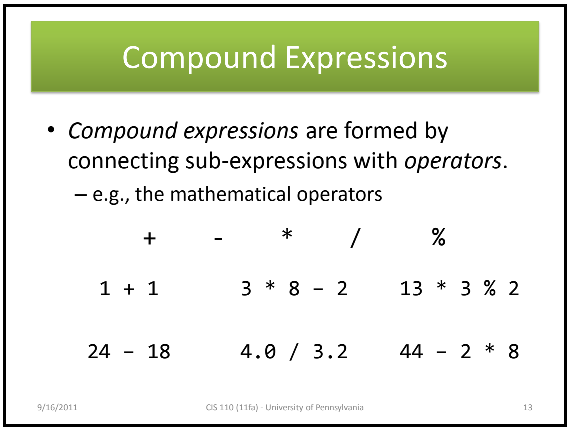
5.0 9.0 100 32

A whole bunch of expressions!

Literal Expressions

- *Literal expressions* evaluate to the value they literally stand for.

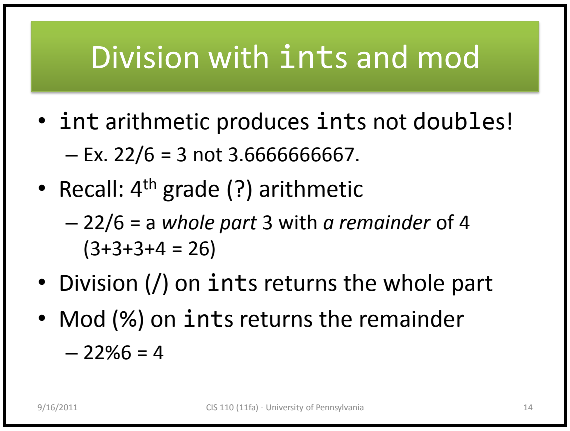
int	0	45	-137	0xF31
double	0.15	8.1	55.0	-13.2
char	‘Q’	‘\n’	‘\’’	‘\\’
boolean	true	false		



Compound Expressions

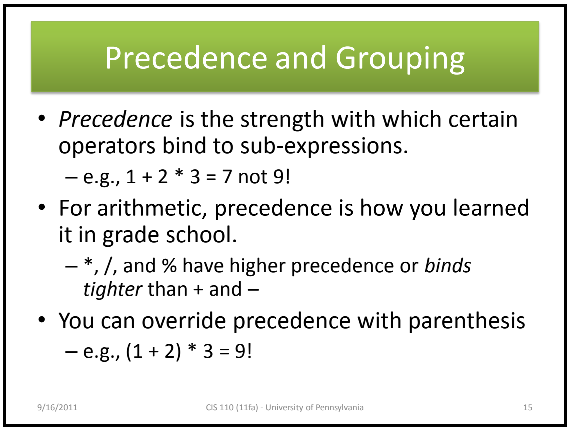
- *Compound expressions* are formed by connecting sub-expressions with *operators*.
  - e.g., the mathematical operators

```
  +     -     *     /     %

 1 + 1       3 * 8 – 2    13 * 3 % 2

24 – 18      4.0 / 3.2    44 – 2 * 8
```



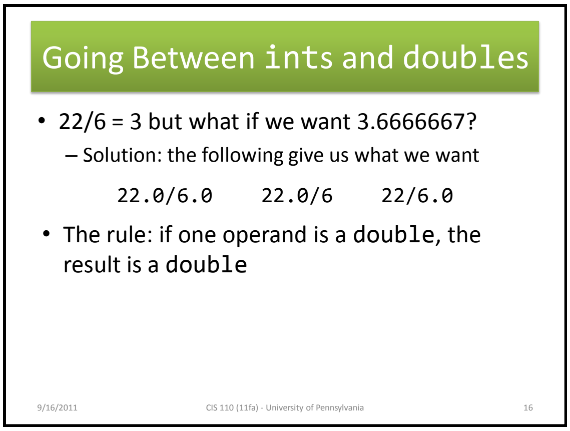
Division with `int`s and mod

- `int` arithmetic produces `int`s not `double`s!
  - Ex. 22/6 = 3 not 3.6666666667.
- Recall: 4th grade (?) arithmetic
  - 22/6 = a *whole part* 3 with *a remainder* of 4 (3+3+3+4 = 26)
- Division (/) on `int`s returns the whole part
- Mod (%) on `int`s returns the remainder
  - 22%6 = 4



Precedence and Grouping

- *Precedence* is the strength with which certain operators bind to sub-expressions.
  - e.g., 1 + 2 * 3 = 7 not 9!
- For arithmetic, precedence is how you learned it in grade school.
  - *, /, and % have higher precedence or *binds tighter* than + and –
- You can override precedence with parenthesis
  - e.g., (1 + 2) * 3 = 9!

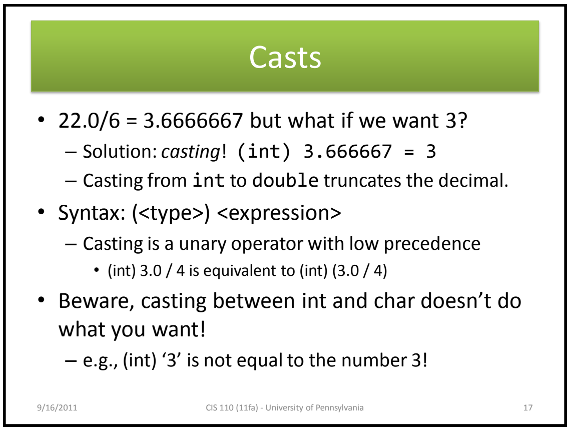


Going Between `int`s and `double`s

- 22/6 = 3 but what if we want 3.6666667?
  - Solution: the following give us what we want

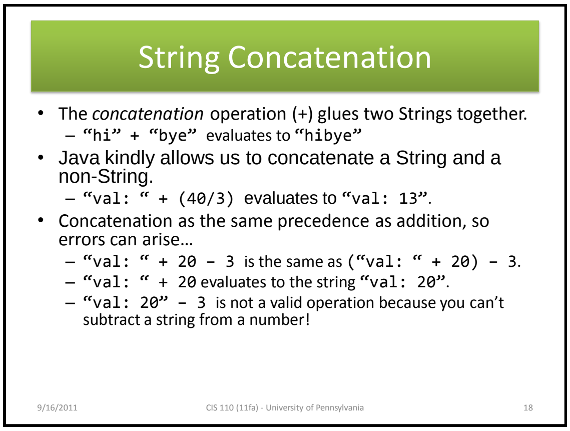
```
22.0/6.0    22.0/6    22/6.0
```

- The rule: if one operand is a `double`, the result is a `double`



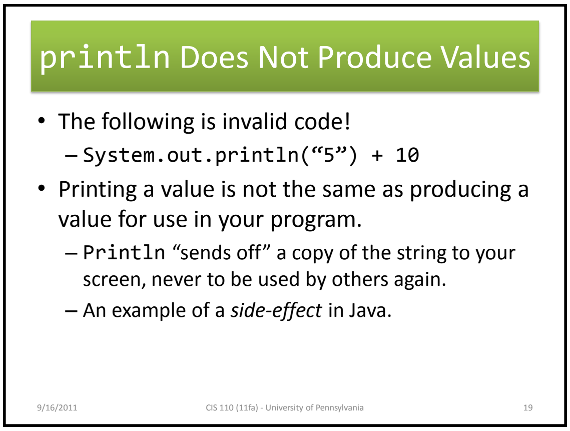
Casts

- 22.0/6 = 3.6666667 but what if we want 3?
  - Solution: *casting*! `(int) 3.666667 = 3`
  - Casting from `int` to `double` truncates the decimal.
- Syntax: (<type>) <expression>
  - Casting is a unary operator with low precedence
    - (int) 3.0 / 4 is equivalent to (int) (3.0 / 4)
- Beware, casting between int and char doesn't do what you want!
  - e.g., (int) '3' is not equal to the number 3!



String Concatenation

- The *concatenation* operation (+) glues two Strings together.
  - `“hi” + “bye”` evaluates to `“hibye”`
- Java kindly allows us to concatenate a String and a non-String.
  - `“val: “ + (40/3)` evaluates to `“val: 13”`.
- Concatenation as the same precedence as addition, so errors can arise…
  - `“val: “ + 20 – 3` is the same as `(“val: “ + 20) – 3`.
  - `“val: “ + 20` evaluates to the string `“val: 20”`.
  - `“val: 20” – 3` is not a valid operation because you can't subtract a string from a number!



`println` Does Not Produce Values

- The following is invalid code!
  - `System.out.println("5") + 10`
- Printing a value is not the same as producing a value for use in your program.
  - `Println` "sends off" a copy of the string to your screen, never to be used by others again.
  - An example of a *side-effect* in Java.