CSCI 136
Data Structures & Advanced Programming

Fall 2019

Instructors

Bill Lenhart & Samuel McCauley

Administrative Details

- Lab 1 handout is online
- Pre-lab Tasks (see Lab 1 handout)
  - Pre-Lab Step 0: Complete it by 4 pm today
  - Pre-Lab Steps 1-2: : Complete it before lab
- TA hours start tonight
  - See TA hour schedule on course website

Last Time

Basic Java elements so far

- Primitive and array types
- Variable declaration and assignment
- Some control structures
  - for, for-each, while, do-while

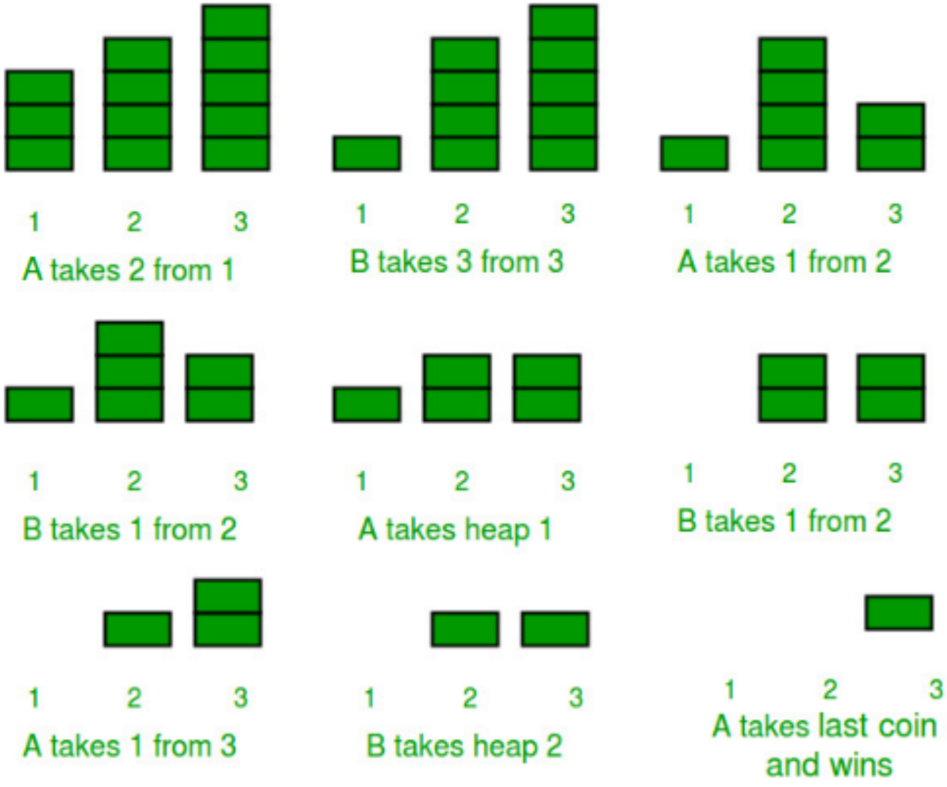
Some basic unix commands

- Compile (javac), run (java) cycle

Today

- Further examples : The Game of Nim
- Operators & operator precedence
- Expressions
- Control structures
    - Branching: if – else, switch, break, continue
    - Looping: while, do – while, for, for – each
- Discussion: Lab 1

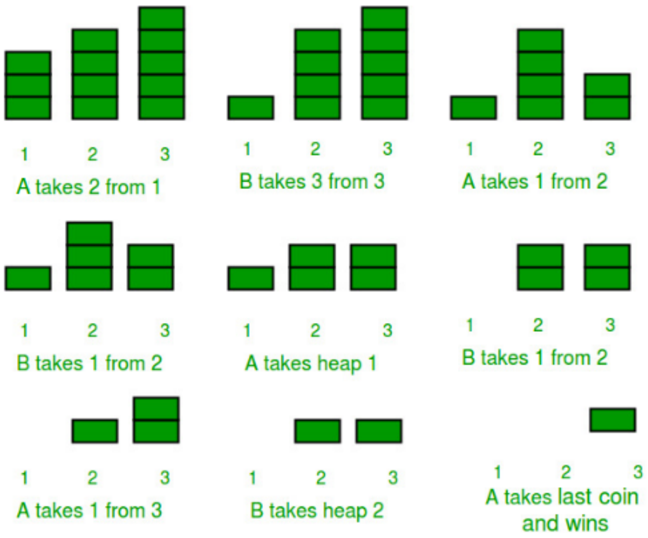
Coding Example : Nim



Courtesy geeksforgeeks.org

Nim

- A 2-player (or multi-player) game
  - Materials: Piles of coins
  - A turn: Take one or more coins from a pile
  - Winner: Player who takes final coin(s)



Courtesy geeksforgeeks.org

Design Doc : No-Objects Nim

State

- Array : holds pile sizes
- Number non-empty piles (or remaining coins)

Functionality

- Create the piles
- Display the piles
- Game over check
- Take a turn

Design Doc : No-Objects Nim

Functionality

- Create the piles
  - Allocate array; choose random pile sizes
- Display the piles
  - Each pile will be a row of O's
- Game over check
  - Is number of non-empty piles > 0?
- Take a turn
  - Check that move is legal
  - Update board

Pseudo-Code: No-Objects Nim

```
Nim( number_of_piles )

    createBoard(number_of_piles)
    displayBoard()

    while not gameOver()
      takeATurn()
      displayBoard()

    print("Game over!")
```

Main Method: No-Objects Nim

```
public static void main(String[] args) {
   if (args.length == 0) {
        System.out.println(
            "Usage: java Nim <number of piles>");
        System.exit(0); // Stop program
   }

   createBoard(Integer.valueOf(args[0]));
   displayBoard();
   while (! gameOver()) {
     takeATurn();
     displayBoard();
   }
   System.out.println("Game over!");
}
```

Data Declaration : No-Objects Nim

```
private static int[] board
private static int piles
private static int pilesLeft;
private static int minPileSize = 3;
private static int maxPileSize = 8;
private static Random rng = new Random();
private static Scanner in = new Scanner(System.in);
```

Create Board : No-Objects Nim

```
public static void createBoard(int size) {

   // Create the board
   piles = size;
   board = new int[piles];

   // Fill the board with randomly sized piles
   for(int i=0; i< board.length; i++)
      board[i] = minPileSize +
         rng.nextInt(maxPileSize - minPileSize + 1);

   // Every pile is non-empty
   pilesLeft = piles;
}
```

Display Board : No-Objects Nim

```
public static void displayBoard() {

   for(int i = 0; i < board.length; i++) {
     System.out.print(i + ":");

     // Display a pile
     for(int j=0; j < board[i]; j++)
        System.out.print(" O");

     // Start a new output line
     System.out.println();
   }
}
```

Take a Turn: No-Objects Nim

```
public static void takeATurn() {
   System.out.print("Enter input"); // Bad prompt!
   int pile = in.nextInt(); // Using Scanner object
   int num = in.nextInt();

   while (pile >= board.length || board[pile] == 0 ||
        board[pile] < num ) {
      System.out.print("Enter input");
      pile = in.nextInt();
      num = in.nextInt();
   }

   board[pile] -= num;
   if (board[pile] == 0) pilesLeft--;
}
```

Notes: No-Objects Nim

- Because we don’t create Nim objects
  - All data elements are *static*
    - Don’t belong to a given object of type Nim
  - All methods are *static*
    - Do not work on a given object of type Nim
- But objects are used
  - `rng` is an object of type `Random`
  - `in` is an object of type `Scanner`
- We need to tell Java where they are

```
import java.util.Random;

import java.util.Scanner;
```

- Note: `piles` isn’t needed: use `board.length`!

Operators

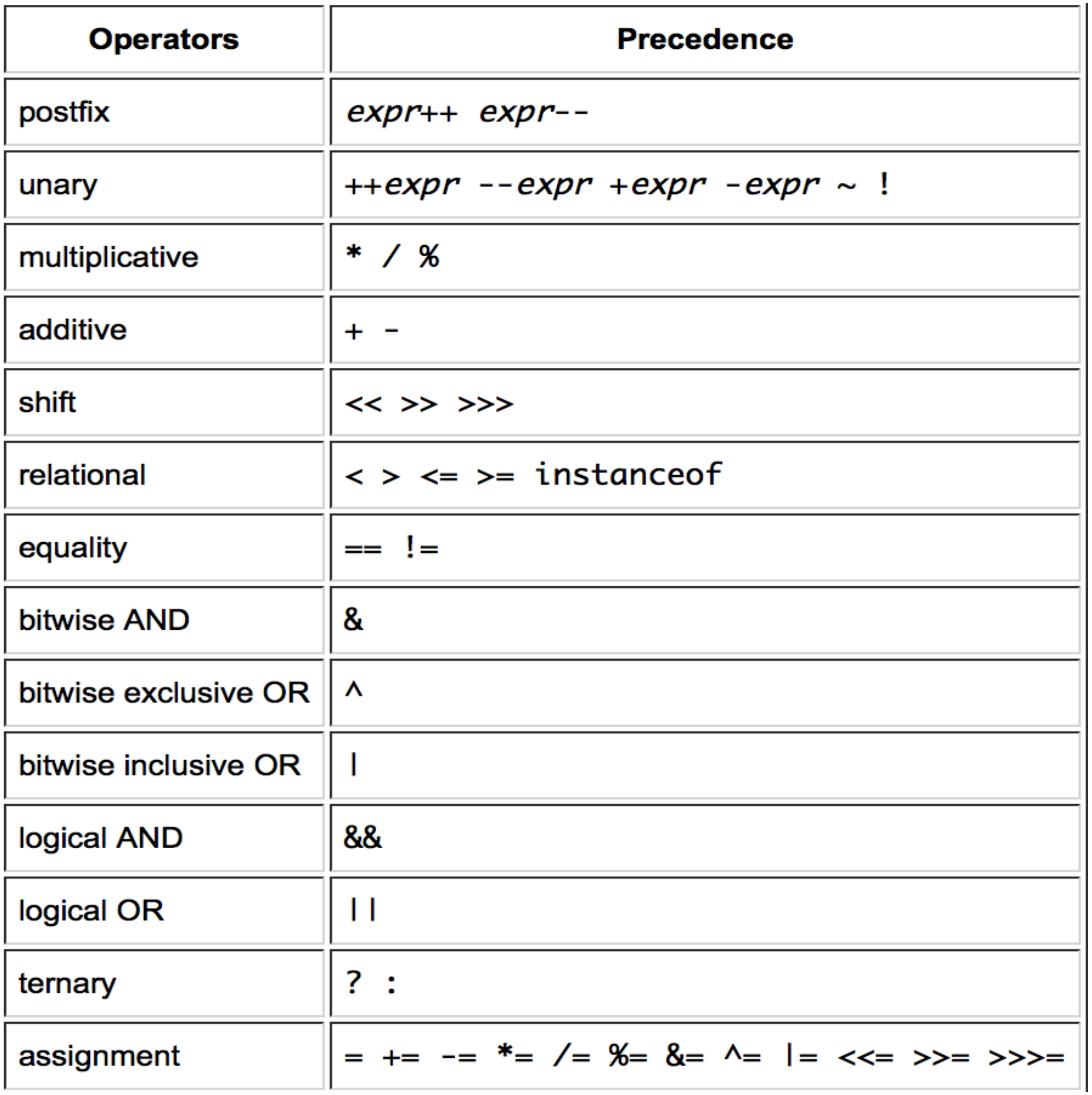
Java provides a number of built-in *operators* including

- Arithmetic operators: +, -, *, /, %
- Relational operators: ==, !=, <, **≤, >, ≥**
- Logical operators &&, || (don’t use &, |)
- Assignment operators =, +=, -=, *=, /=, ...

Common unary operators include

- Arithmetic: - (prefix); ++, -- (prefix and postfix)
- Logical: ! (not)

Operator Precedence in Java



Operators	Precedence
postfix	*expr*++ *expr*--
unary	++*expr* --*expr* +*expr* -*expr* ~ !
multiplicative	* / %
additive	+ -
shift	<< >> >>>
relational	< > <= >= instanceof
equality	== !=
bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	|
logical AND	&&
logical OR	||
ternary	? :
assignment	= += -= *= /= %= &= ^= |= <<= >>= >>>=

Operator Gotchas!

- There is no exponentiation operator in Java.
  - The symbol ^ is the *bitwise or* operator in Java.
- The *remainder* operator % is the same as the mathematical 'mod' function for *positive* arguments,
  - For **negative** arguments **it is not**: -8 % 3 = -2
- The logical operators && and || use *short-circuit evaluation*:
  - Once the value of the logical expression can be determined, no further evaluation takes place.
  - E.g.: If n is 0, then `(n != 0) && (k/n > 3)`, will yield false without evaluating k/n. Very useful!

Expressions

Expressions are either:

- literals, variables, invocations of non-void methods, or
- statements formed by applying operators to them

An expression returns a value

- `3+2*5 - 7/4        // returns 12`
- `x + y*z – q/w`
- `(- b + Math.sqrt(b*b – 4 * a * c) )/( 2* a)`
- `( n > 0) && (k / n > 2) // computes a` *`boolean`*

Expressions

Assignment operator also forms an expression

- x = 3;   // assigns x the value 3 and returns 3
- So y = 4 * (x = 3) sets x = 3 and y = 12 (and returns 12)

Boolean expressions let us control program *flow of execution* when combined with *control structures*

Example

```
– if ( (x < 5) && (y !=0 ) ) {...}
– while (! loggedIn) { ... }
```

Control Structures

Select next statement to execute based on value of a boolean expression. Two flavors:

- Looping structures: `while, do/while, for`
  - Repeatedly execute same statement (block)
- Branching structures: `if, if/else, switch`
  - Select one of several possible statements (blocks)
  - Special: break/continue: exit a looping structure
    - break: exits loop completely
    - continue: proceeds to next iteration of loop

while & do-while

Consider this code to flip coin until heads up...

```
Random rng = new Random();
int flip = rng.nextInt(2), count = 0;
while (flip == 0) {  // count flips until “heads”
   count++;
   flip = rng.nextInt(2);
}
```

...and compare it to this

```
int flip, count = 0;
do {                 // count flips until “heads”
   count++;
   flip = rng.nextInt(2);
} while (flip == 0) ;
```

For & for-each

Here’s a typical **for** loop example

```
int[] grades = { 100, 78, 92, 87, 89, 90 };
int sum = 0;
for( int i = 0; i < grades.length; i++ )
    sum += grades[i];
```

This **for** construct is equivalent to

```
int i = 0;
while ( i < grades.length ) {
    sum += grades[i];
    i++;
}
```

Can also write

```
for (int g : grades ) sum += g;
// called for-each construct
```

Loop Construct Notes

- The body of a **while** loop may not ever be executed
- The body of a **do – while** loop always executes at least once
- **For** loops are typically used when number of iterations desired is known in advance. E.g.
  - Execute loop exactly 100 times
  - Execute loop for each element of an array
- The **for-each** construct is often used to access array (and other collection type) values when *no updating* of the array is required
  - We'll explore this construct more later in the course

If/else

```
if (x > 0)        // There is exactly 1 "if" clause
      y = 1 / x;
else if (x<0) {   // 0 or more "else if" clauses
      x = - x;
      y = 1 / x;
}
else              // at most 1 "else" clause
      System.out.println(“Can’t divide by 0!”);
```

The single statement can be replaced by a *block:* any sequence of statements enclosed in {}

switch

Example: Encode clubs, diamonds, hearts, spades as 0, 1, 2, 3

```
int x = myCard.getSuit(); // a fictional method
switch (x) {
   case 1: case 2:
      System.out.println("Your card is red");
      break;
   case 0: case 3:
      System.out.println("Your card is black");
      break;
   default:
      System.out.println("Illegal suit code!");
      break;
}
```

Break & Continue

Suppose we have a method `isPrime` to test primality

Find first prime > 100

```
for( int i = 101; ; i++ )
    if ( isPrime(i) ) {
        System.out.println( i );
        break;
    }
```

Print primes < 100

```
for( int i = 1; i < 100 ; i++ ) {
    if ( !isPrime(i) )
        continue;
    System.out.println( i );
}
```

Lab 1

- Purpose
  - Exercise your Java skills by programming a game
  - Learn some new tools
    - Terminal command-line interface to Unix
    - Atom program editor
    - GitHub version control system
  - Learn some code development habits
    - Design documents
    - Pseudo-code

Lab 1

- GitHub
  - Cloud support for file storage with *version control*
  - Basic commands
    - git clone – make a local copy of an existing repository
    - git add – add files to local copy of repository
    - git rm – remove a file from local copy
    - git commit – commit staged changes
    - git push – update master repository with committed changes in local repository
    - git pull – update local repository from master

Lab 1

- CoinStrip Game
  - Two-player coin-moving game (let's play!)
  - Essentials
    - Decide on game representation
    - Build the board
      - Random coin locations
    - Allow players to take turns
      - Enter, check, process a move
    - Congratulate the winner!

Summary

Java

- Writing ”no-objects” code: Nim
- More on conditional control flow
  - Switch, break, continue
- Using classes from external packages
  - Random, Scanner
  - Import statement
- Use of static for non-object-based data and methods
- Lab 1 overview

Lecture Ends Here