administrivia...
- assignment 1 posted
  - due next Wednesday by midnight

- TA office hours posted

- Clicker registration
channel: 41
session: cs2420
last time...
-we refer to unspecified integer quantities as $N$
  - $N$ is the problem size
    - sorting an array of $N$ numbers
    - searching for an item in a set of $N$ items
    - inserting an item into a set of $N$ items

-amount of work done for these operations usually depends on $N$
  - work required is a function of $N$
sort1 versus sort2
Choosing an Algorithm

How important is it to pick the best algorithm for the job?

**Running Time (milliseconds)**

- **sort 1**
- **sort 2**

**N (size of list)**

- Small N
Choosing an Algorithm

(Medium problem size)

N (size of list)

Sorting Algorithm Comparison

- **Sort 1**
- **Sort 2**

Running time (milliseconds)

- From 0.01 to 0.1 milliseconds
- From 100 to 550 elements

The graph shows the running time of Sort 1 and Sort 2 as the size of the list (N) increases. Sort 1 has a steeper slope than Sort 2, indicating a higher time complexity.
Choosing an Algorithm

As \( N \) becomes large, complexity matters!

![Graph showing running time vs. list size](image)
Choosing an Algorithm

As $N$ becomes large, complexity matters!

**take away:**

as $N$ becomes large, complexity matters!
void sort1(int[] arr) {
    for(int i = 0; i < arr.length-1; i++) {
        int j, minIndex;
        for(j = i+1, minIndex = i; j < arr.length; j++)
            if(arr[j] < arr[minIndex])
                minIndex = j;
        swap(arr, i, minIndex);
    }
}

void sort2(int[] arr, int beg, int end) {
    if (end > beg + 1) {
        int piv = arr[beg], l = beg + 1, r = end;
        while (l < r) {
            if (arr[l] <= piv)
                l++;
            else
                swap(arr, l, --r);
        }
        swap(arr, --l, beg);
        sort2(arr, beg, l);
        sort2(arr, r, end);
    }
}
today...
disclaimer: this class is *not* about teaching you Java
-variables
-control flow
-reference types
-misc.
-mini lab
variables
- a **variable** is a piece of data in memory with:
  - an identifier (name)
  - a **type**
- a variable is a piece of data in memory with:
  - an identifier (name)
  - a type

- what is a type?
- **variable** is a piece of data in memory with:
  - an identifier (name)
  - a **type**

- **what is a type?**
  - a basic building block in a programming language
  - determines what kind of data a variable holds, and what operations can be performed on it
- a **variable** is a piece of data in memory with:
  - an identifier (name)
  - a **type**

- **what is a type?**
  - a basic building block in a programming language
  - determines what kind of data a variable holds, and what operations can be performed on it

- **Java defines eight primitive types**
- a **variable** is a piece of data in memory with:
  - an identifier (name)
  - a **type**

- **what is a type?**
  - a basic building block in a programming language
  - determines what kind of data a variable holds, and what operations can be performed on it

- **Java defines eight primitive types**
  - byte, short, int, long, float, double, char, boolean
- A **variable** is a piece of data in memory with:
  - an identifier (name)
  - a **type**

- **What is a type?**
  - A basic building block in a programming language
  - Determines what kind of data a variable holds, and what operations can be performed on it

- **Java defines eight primitive types**
  - byte, short, int, long, float, double, char, boolean
  - Each primitive type can hold a single value
    - `r`, 12, 2.64, true
**declaration & initialization**

- **declaring** a variable is stating that it exists  
  - assigns the variable a type and name
  
  ```java
  boolean areWeThereYet;
  ```

- **initializing** a variable gives it an initial value, and is often combined with declaring
  
  ```java
  boolean areWeThereYet = false;
  ```

- variables declared as **final** are constant and cannot be changed after initialization
  
  ```java
  final int theMeaningOfLife = 42;
  ```
assignment

- After a variable has been declared we can assign it a new value with =

  areWeThereYet = true;

- We can use arithmetic expressions with an assignment

  age = currentYear - birthYear;
arithmetic operations

- explicitly supported on primitive types
  - binary operators
  - unary operators

- Java follows common order-of-operation rules
arithmetic operations

-explicitly supported on primitive types
  -binary operators
    +, -, *, /, %
  -unary operators

-Java follows common order-of-operation rules
arithmetic operations

-explicitly supported on primitive types
  -binary operators
    +, -, *, /, %
  -unary operators
    - (negation), ++ (increment), -- (decrement)

-Java follows common order-of-operation rules
arithmetic operations

- explicitly supported on primitive types
  - binary operators
    +, -, *, /, %
  - unary operators
    - (negation), ++ (increment), -- (decrement)

- Java follows common order-of-operation rules

  unary ops : highest
arithmetic operations

- explicitly supported on primitive types
  - binary operators
    +, -, *, /, %

  - unary operators
    - (negation), ++ (increment), -- (decrement)

- Java follows common order-of-operation rules

  unary ops : highest
  *, /, % : high
arithmetic operations

- explicitly supported on primitive types
  - binary operators
    +, -, *, /, %
  - unary operators
    - (negation), ++ (increment), -- (decrement)

- Java follows common order-of-operation rules

  unary ops : highest
  *, /, % : high
  +, - : low
arithmetic operations

-explicitly supported on primitive types
  -binary operators
    +, -, *, /, %
  -unary operators
    - (negation), ++ (increment), -- (decrement)

-Java follows common order-of-operation rules

  unary ops : highest
    *, /, % : high
    +, - : low
    = : lowest
type conversion

- **widening conversions** convert data to another type that has the same or more bits of storage

- **narrowing conversions** convert data to another type that has fewer bits of storage
type conversion

- *widening conversions* convert data to another type that has the same or more bits of storage
  - short $\rightarrow$ int
  - int $\rightarrow$ long
  - int $\rightarrow$ float

- *narrowing conversions* convert data to another type that has fewer bits of storage
type conversion

-widening conversions convert data to another type that has the same or more bits of storage
  short  -> int
  int     -> long
  int     -> float

-narrowing conversions convert data to another type that has fewer bits of storage
  double -> float
  float  -> int
type conversion

-widening conversions convert data to another type that has the same or more bits of storage
  - short -> int
  - int   -> long
  - int   -> float

-narrowing conversions convert data to another type that has fewer bits of storage
  - double -> float
  - float  -> int

what do you have to be careful about here?
type conversion

-java uses widening conversion when an operator is applied to operands of different types (called promotion)
type conversion

- java uses widening conversion when an operator is applied to operands of different types (called promotion)

```
2.2 * 2
1.0 / 2
double x = 2;
"count = " + 4
```
type conversion

- Java uses widening conversion when an operator is applied to operands of different types (called promotion)

\[
\begin{align*}
2.2 \ast 2 & \quad \text{evaluates to 4.4} \\
1.0 \div 2 & \\
\text{double } x = 2; & \\
\text{“count = “} + 4 &
\end{align*}
\]
type conversion

- Java uses widening conversion when an operator is applied to operands of different types (called promotion)

```
2.2 * 2  evaluates to 4.4
1.0 / 2  evaluates to 0.5
double x = 2;
“count = “ + 4
```
type conversion

- java uses widening conversion when an operator is applied to operands of different types (called promotion)

\[
\begin{align*}
2.2 \times 2 & \quad \text{evaluates to 4.4} \\
1.0 / 2 & \quad \text{evaluates to 0.5} \\
\text{double } x = 2; & \quad \text{assigns 2.0 to } x \\
"count = " + 4 & \\
\end{align*}
\]
type conversion

java uses widening conversion when an operator is applied to operands of different types (called promotion)

- $2.2 * 2$ evaluates to 4.4
- $1.0 / 2$ evaluates to 0.5
- `double x = 2;` assigns 2.0 to `x`
- "count = " + 4 evaluates to "count = 4"
type conversion

- Java uses widening conversion when an operator is applied to operands of different types (called promotion)

\[
\begin{align*}
2.2 \times 2 & \quad \text{evaluates to 4.4} \\
1.0 \div 2 & \quad \text{evaluates to 0.5} \\
\text{double } x = 2; & \quad \text{assigns 2.0 to } x \\
\text{"count = “} + 4 & \quad \text{evaluates to “count = 4”}
\end{align*}
\]
mixing types

- conversions are done on one operator at a time in the order the operators are evaluated
mixing types

-conversions are done on one operator at a time in the order the operators are evaluated

3 / 2 * 3.0 + 8 / 3
mixing types

-conversions are done on one operator at a time in the order the operators are evaluated

\[ \frac{3}{2} \times 3.0 + \frac{8}{3} = 5.0 \]
mixing types

-conversions are done on one operator at a time in the order the operators are evaluated

\[
\begin{align*}
3 / 2 \times 3.0 + 8 / 3 &= 5.0 \\
2.0 \times 4 / 5 + 6 / 4.0 &= 5.0
\end{align*}
\]
mixing types

-conversions are done on one operator at a time in the order the operators are evaluated

\[
\begin{align*}
3 / 2 & \times 3.0 + 8 / 3 & = 5.0 \\
2.0 & \times 4 / 5 + 6 / 4.0 & = 3.1
\end{align*}
\]
mixing types

- String concatenation has the same precedence as + – and is evaluated left to right
mixing types

- String concatenation has the same precedence as + – and is evaluated left to right

\[ 1 + "x" + 4 \]
mixing types

- String concatenation has the same precedence as + – and is evaluated left to right

\[ 1 + "x" + 4 \quad "1x4" \]
mixing types

- String concatenation has the same precedence as + − and is evaluated left to right

\[
1 + "x" + 4 \quad "1x4"
\]

\[
"2+3=" + 2 + 3
\]
mixing types

- String concatenation has the same precedence as + - and is evaluated left to right

\[
\begin{align*}
1 + \text{“x”} + 4 & \quad \text{“1x4”} \\
\text{“2+3=“} + 2 + 3 & \quad \text{“2+3=23”}
\end{align*}
\]
mixing types

- String concatenation has the same precedence as + – and is evaluated left to right

```
1 + "x" + 4   "1x4"
"2+3=" + 2 + 3   "2+3=23"
1 + 2 + "3"
```
- String concatenation has the same precedence as `+` and `-` and is evaluated left to right

\[
\begin{align*}
1 + \text{"x"} + 4 & \quad \text{"1x4"} \\
\text{"2+3="} + 2 + 3 & \quad \text{"2+3=23"} \\
1 + 2 + \text{"3"} & \quad \text{"33"}
\end{align*}
\]
mixing types

- String concatenation has the same precedence as + - and is evaluated left to right

1 + "x" + 4 "1x4"

"2+3=" + 2 + 3 "2+3=23"

1 + 2 + "3" "33"

"2*3=" + 2 * 3
mixing types

- String concatenation has the same precedence as + – and is evaluated left to right

\[
\begin{align*}
1 + "x" + 4 &= "1x4" \\
"2+3=" + 2 + 3 &= "2+3=23" \\
1 + 2 + "3" &= "33" \\
"2*3=" + 2 * 3 &= "2*3=6"
\end{align*}
\]
mixing types

-String concatenation has the same precedence as +− and is evaluated left to right

\[
\begin{align*}
1 + "x" + 4 & \quad \Rightarrow \quad "1x4" \\
"2+3=" + 2 + 3 & \quad \Rightarrow \quad "2+3=23" \\
1 + 2 + "3" & \quad \Rightarrow \quad "33" \\
"2*3=" + 2 * 3 & \quad \Rightarrow \quad "2*3=6" \\
4 - 1 + "x" & \\
\end{align*}
\]
String concatenation has the same precedence as + - and is evaluated left to right

1 + "x" + 4 = "1x4"
"2+3=" + 2 + 3 = "2+3=23"
1 + 2 + "3" = "33"
"2*3=" + 2 * 3 = "2*3=6"
4 - 1 + "x" = "3x"
mixing types

- String concatenation has the same precedence as + - and is evaluated left to right

\[
\begin{array}{ccc}
1 + "x" + 4 & \rightarrow & "1x4"\\
"2+3=" + 2 + 3 & \rightarrow & "2+3=23"\\
1 + 2 + "3" & \rightarrow & "33"\\
"2*3=" + 2 * 3 & \rightarrow & "2*3=6"\\
4 - 1 + "x" & \rightarrow & "3x"\\
"x" + 4 - 1 & \rightarrow & "3x"
\end{array}
\]
mixing types

- String concatenation has the same precedence as + − and is evaluated left to right

```
1 + "x" + 4       "1x4"
"2+3=" + 2 + 3    "2+3=23"
1 + 2 + "3"       "33"
"2*3=" + 2 * 3    "2*3=6"
4 - 1 + "x"       "3x"
"x" + 4 - 1       error
```
type casting

- **type casting** tells Java to convert one type to another
  - convert an **int** to a **double** to force floating-point division
    
    double average = (double) 12 / 5;

- truncate a **double** to an **int**
  
  int feet = (int) (28.3 / 12.0);
assignment operators

- basic assignment operator
  =

- combined assignment/arithmetic operators
  +=, -=, *=, /=

- increment/decrement operators can be stand-alone statements
  i++; 
i--; 
++i; 
--i;
assignment operators

- basic assignment operator
  =

- combined assignment/arithmetic operators
  +=, -=, *=, /=

- increment/decrement operators can be stand-alone statements

  ```java
  int i = 3;
  i++; // int j = i++;
  i--; // System.out.println(i+" "++j);
  ++i;
  --i;
  ```
assignment operators

- basic assignment operator
  =

- combined assignment/arithmetic operators
  +=, -=, *=, /=

- increment/decrement operators can be stand-alone statements

  i++;          int i = 3;
  i--;          int j = i++;
  ++i;          System.out.println(i+" "+j);
  --i;          int i = 3;
  int j = ++i;
  System.out.println(i+" "+j);
relational and logical ops

- results are always boolean

- relational ops supported for number and character types (and equality for boolean)
  - >, <, >=, <=, ==, !=

- logical ops supported for boolean
  - &&, ||, !

- precedence (all lower than arithmetic):
relational and logical ops

- results are always boolean

- relational ops supported for number and character types (and equality for boolean)
  >, <, >=, <=, ==, !=

- logical ops supported for boolean
  &&, ||, !

- precedence (all lower than arithmetic):
  >, <, >=, <= : highest
relational and logical ops

- results are always boolean

- relational ops supported for number and character types (and equality for boolean)
  >, <, >=, <=, ==, !=

- logical ops supported for boolean
  &&, ||, !

- precedence (all lower than arithmetic):
  >, <, >=, <= : highest
  ==, != : high
- Relational and Logical Ops

- Results are always boolean

- Relational ops supported for number and character types (and equality for boolean)
  - >, <, >=, <=, ==, !=

- Logical ops supported for boolean
  - &&, ||, !

- Precedence (all lower than arithmetic):
  - >, <, >=, <= : highest
  - ==, != : high
  - && : low
relational and logical ops

- results are always boolean

- relational ops supported for number and character types (and equality for boolean)
  
  \[>, <, \geq, \leq, ==, !=\]

- logical ops supported for boolean
  
  \[\&\&, \mid\mid, !\]

- precedence (all lower than arithmetic):

  \[\geq, <, \leq, =, !=, \&\&, \mid\mid, !\]

  - highest
  - high
  - low
  - lowest
control flow
control flow determines how programs make decisions about what to do, and how many times to do it

- decision making: if-else, switch-case
- looping: for, while, do-while
- jumping: break, continue, return
- exceptions: try-catch, throw
switch statements

-similar to an if-else-if statement

```
switch (integer expression)
{
    case <integer literal>:
        list of statements...

    case <integer literal>:
        ...
}
```
switch statements

-execution begins on the ___ case that matches the value of the switch variable

-execution continues until _____ is reached
switch statements

-execution begins on the first case that matches the value of the switch variable

-execution continues until _______ is reached
switch statements

- execution begins on the first case that matches the value of the switch variable

- execution continues until break is reached
switch statements

-execution begins on the first case that matches the value of the switch variable

-execution continues until break is reached
  -even continues through other cases!
  -usually want a break after every case
switch statements

-execution begins on the **first** case that matches the value of the switch variable

-execution continues until **break** is reached
  -even continues through other cases!
  -usually want a **break** after every case

-switches can use the **default** keyword
  -if no cases were hit, execute the **default** case
  -similar to an **else** at the end of a long line of **if-else-if**
exceptions

-an exception is a special event that interrupts the control of the program

-exceptions are “thrown” explicitly by the code

-use a try block to wrap any code that might throw an exception

-a catch block immediately follows a try block

-execution of the program jumps inside the catch block if an exception occurred within the try block
try
{
    FileReader in = new FileReader("fakefile.txt");
}
catch(FileNotFoundException e)
{
    System.out.println("file does not exist");
}
catch(Exception e) // a less specific error occurred
{
    System.err.println(e.getMessage());
}
throwing exceptions

if (arraySize < 0)
    throw new NegativeArraySizeException();
arr = new int[arraySize];

-why don’t we need an else? (what are we assuming?)
throwing exceptions

if (arraySize < 0)
    throw new NegativeArraySizeException();
arr = new int[arraySize];

- why don’t we need an else? (what are we assuming?)
- there are many many subclasses of exceptions…
throwing exceptions

if (arraySize < 0)
    throw new NegativeArraySizeException();
arr = new int[arraySize];

- why don’t we need an else? (what are we assuming?)

- there are many many subclasses of exceptions…

- you can even define your own!

public class BadnessOccurred extends Exception {
    ...
}
reference types
- all non-primitive types are reference types
- all non-primitive types are reference types

- a reference is a variable that stores the memory address where an object (a group of values) resides
- all non-primitive types are **reference types**

- a **reference** is a variable that stores the memory address where an object (a group of values) resides

```java
Point p1, p2, p3;
p1 = new Point(7,19);
p2 = p1;
p2.x = 2;
```
-all non-primitive types are **reference types**

-a **reference** is a variable that stores the memory address where an object (a group of values) resides

Point p1, p2, p3;
p1 = new Point(7,19);
p2 = p1;
p2.x = 2;
- All non-primitive types are reference types

- A reference is a variable that stores the memory address where an object (a group of values) resides

Point p1, p2, p3;
p1 = new Point(7, 19);
p2 = p1;
p2.x = 2;
- all non-primitive types are **reference types**

- a **reference** is a variable that stores the memory address where an object (a group of values) resides

```java
Point p1, p2, p3;
p1 = new Point(7, 19);
p2 = p1;
p2.x = 2;
```
- All non-primitive types are **reference types**

- A **reference** is a variable that stores the memory address where an object (a group of values) resides

```java
Point p1, p2, p3;
p1 = new Point(7,19);
p2 = p1;
p2.x = 2;
```
- all non-primitive types are **reference types**

- a **reference** is a variable that stores the memory address where an object (a group of values) resides

Point p1, p2, p3;
p1 = new Point(7, 19);
p2 = p1;
p2.x = 2;
reference declaration

- declaration of a reference variable only provides a name to reference an object — *it does not create an object*

- *after* `Point p1;` the value stored in `p1` is _____
reference declaration

- declaration of a reference variable only provides a name to reference an object — *it does not create an object*

- *after* `Point p1;` the value stored in `p1` is *null*
reference declaration

- declaration of a reference variable only provides a name to reference an object — it does not create an object

- after Point p1; the value stored in p1 is null

- the new keyword is used to construct an object
  Point p1 = new Point();
  Point p2;
  p2 = new Point();
reference declaration

- declaration of a reference variable only provides a name to reference an object — *it does not create an object*

-after `Point p1;` the value stored in `p1` is `null`

-the `new` keyword is used to construct an object

```
Point p1 = new Point();
Point p2;
p2 = new Point();
```

why are `()` needed?
operations on reference types

-operations on references: =, ==, !=
  -equality operators compare addresses
operations on reference types

- operations on references: =, ==, !=
  - equality operators compare addresses

- what does \texttt{p2 == p1} return?

```java
Point p1, p2, p3;
p1 = new Point(7,19);
p2 = p1;
```
operations on reference types

-operations on objects: ., instanceof
  -the . operator is used to select a method that is applied to an object, or an individual component of an object

Point p1, p2, p3;
p1 = new Point(7,19);
p2 = p1;
operations on reference types

-operations on objects: ., instanceof
  -the . operator is used to select a method that is applied to an object, or an individual component of an object

```java
Point p1, p2, p3;
p1 = new Point(7,19);
p2 = p1;
```

**What does** `p2.firstValue()` **return?**
operations on reference types

- operations on objects: ., instanceof
  - the . operator is used to select a method that is applied to an object, or an individual component of an object

```java
Point p1, p2, p3;
p1 = new Point(7,19);
p2 = p1;
```

**what does** p2.firstValue() **return?**

**what does** p1 instanceof Point **return?**
String

- String is the only reference type for which operator overloading is allowed (+ and +=)
  - but you can’t define your own operator overloads!

- String objects are immutable

- to compare String objects use equals and compareTo methods — not ==, !=, <, or >
  - why?

- other useful String methods:
  - length, charAt, substring
String s1 = "Hello";
String s2 = s1;
s1 = "Good-bye";
System.out.println(s2);
String s1 = "Hello";
String s2 = s1;
s1 = "Good-bye";
System.out.println(s2);
String s1 = "Hello"; 
String s2 = s1; 
s1 = "Good-bye"; 
System.out.println(s2);
String s1 = "Hello";
String s2 = s1;
s1 = "Good-bye";
System.out.println(s2);
String s1 = "Hello";
String s2 = s1;
s1 = "Good-bye";
System.out.println(s2);
String

- **String** is the only reference type for which operator overloading is allowed (+ and +=)
  - but you can’t define your own operator overloads!

- **String** objects are immutable

- To compare **String** objects use **equals** and **compareTo** methods — not ==, !=, <, or >
  - why?

- Other useful **String** methods:
  - length, charAt, substring
arrays

- an array is a mechanism for storing a collection of identically typed entities

- in Java, arrays behave like objects

- the [] operator indexes an array, accessing an individual entity — bounds checking is performed automatically

- by default, array elements are initialized 0 (primitive types) and null (reference types)

Point[] refArray = new Point[10];
double[] primArray = {3.14, 2.2, -9.8};
ArrayList

- The ArrayList class (from the Collections library) mimics an array and allows for dynamic expansion.

- The get, set methods are used in place of [ ] for indexing.

- The add method increases the size by one and adds a new item.

- ArrayList may only be used with reference types.

```java
ArrayList<String> a = new ArrayList<String>(1);
a.set(0, "hi");
a.add("there");
```
misc.
parameter passing

- Java uses **call-by-value** parameter passing
  - ie. a copy is created

  ```java
  int i = 4;
  modifyInt(i);
  System.out.println(i); // what does this print?
  ```
parameter passing

- Java uses **call-by-value** parameter passing
  - ie. a copy is created

  ```java
  int i = 4;
  modifyInt(i);
  System.out.println(i); // what does this print?
  ```

- what does this mean for references?

  ```java
  Point p = new Point(1, 2);
  modifyPoint(p);
  System.out.println(p.x); // prints?
  ```
main

-when a program is run, the main method is invoked

public static void main(String[] args)

-the parameters of main can be set using command-line arguments
classes & constructors

- **a class** consists of **fields** (aka. variables) that store data and **methods** that operate on that data

- fields and methods may be **public** or **private**

- the **constructor** controls how an object is created and initialized
  - multiple constructors may be defined, taking different parameters
  - if none is defined, a default constructor is generated
    - initializes primitive fields to 0, and reference fields to null
classes & constructors

- a class consists of fields (aka. variables) that store data and methods that operate on that data

- fields and methods may be public or private

- the constructor controls how an object is created and initialized
  - multiple constructors may be defined, taking different parameters
  - if none is defined, a default constructor is generated
    - initializes primitive fields to 0, and reference fields to null

the difference between field and variable:
http://docs.oracle.com/javase/tutorial/java/nutsandbolts/variables.html
this

- **this is a reference to the current object**
- useful in avoiding self-assignment
this

- **this** is a reference to the current object
  - useful in avoiding self-assignment

// transfer all money from rhs to current account
public void finalTransfer( Account rhs )
{
    dollars += rhs.dollars;
    rhs.dollars = 0;
}
this

- this is a reference to the current object
  - useful in avoiding self-assignment

Account account1;
Account account2;
...
account2 = account1;
account1.finalTransfer( account2 );

// transfer all money from rhs to current account
public void void finalTransfer( Account rhs )
{
    dollars += rhs.dollars;
    rhs.dollars = 0;
}
this

- this is a reference to the current object
  - useful in avoiding self-assignment

```java
Account account1;
Account account2;
...
account2 = account1;
account1.finalTransfer( account2 );

// transfer all money from rhs to current account
public static void finalTransfer( Account rhs )
{
    if ( this == rhs )
        return;
    dollars += rhs.dollars;
    rhs.dollars = 0;
}
```
next time...
- **reading**
  - chapters 3 & 4

- **homework**
  - assignment 1 is up
  - due next Wednesday at midnight
  - *must complete on your own!*

- **lab**
  - testing

- **clicker questions start next week**