24-0: **Review**

- Memory Basics (stack vs. heap)
- Creating Objects
- Control Structures (if, while, etc)
- Mutable and Immutable objects (Strings!)
- Methods (including recursion)
- Arrays
- Static fun
- For Wednesday:
  - Linked Lists
  - Inheritance (including polymorphism)
  - Potpourri (Exceptions, etc)

24-1: **Memory Basics**

- Parameters and local variables are all stored on the stack
- Instance variables in classes are stored on the heap
  - *Pointers* to classes can be stored on the stack
- Primative types (int/boolean/double/etc – non-classes) are stored on the stack *unless they are instance variables within a class*
- Heap memory is only created if you call "new"

24-2: **Memory Basics**

```
public class Foo
{
   int x;
   int y;
}

public static void main(String args[])
{
   int x;
   int y;
   Foo f1, f2;
   //   What does the stack / heap look like?
   f1 = new Foo();
   f2 = new Foo();
   //   What about now?
}
```

24-3: **Memory Basics**

```
public class Foo       public class Bar
{                      {
   int x;                  Foo f1;
   int y;                  Foo f2;
}                      }

public static void main(String args[])
{
   int x;
   int y;
   Bar b;
   //   What does the stack / heap look like?
   b = new Bar();
   //   Now What does the stack / heap look like?
   b.f1 = new Foo();
   //   What about now?
   b.f2 = new Foo();
   //   What about now?
}
```

24-4: **Memory Basics**

```
public class Bar           public class Foo
{                          {
    public Foo f1;            int x;
    public Foo f2;            int y;
    public Bar()           }
    {
       f1 = new Foo();
       f2 = new Foo();
    }
}
public static void main(String args[])
{
   int x,y;
   Foo f;
   bar b;
   //   What does the stack / heap look like?
   b = new Bar();
   //   Now What does the stack / heap look like?
   f = new Foo();
}
```

24-5: **Memory Basics**

- Classes are *Templates*
- Need to call “new” to create instance of classes
- Can only get memory on heap by calling new

24-6: **Java Control Structures**

- If:

```
if (<test>)
   <statement>
```

or

```
if (<test>)
   <statement1>
else
   <statement2>
```

- A statement is either a single statment (terminated with ;), or a block of statements inside { }

24-7: **If test**

- What can we have as the test of an if statement?
  - Boolean valued expression

24-8: **If test**

- What can we have as the test of an if statement?
  - boolean variable
  - function that returns a boolean value
  - comparison `x < y` or `x != 0`
  - Combination of boolean expressions, using not (`!`), and (&&), or (`||`)

24-9: **Boolean Variables**

- Hold the value true or false
- Can be used in test (if, while, etc)

```
boolean b;
boolean c;
b = true;
c = b || false;
b = (3 < 10) && (5 > 11);
c = !b && c;
```

24-10: **if Gotchas**

- What is (likely) wrong with the following code?

```
if (x != 0)
   z = a / x;
   y = b / x;
```

24-11: **if Gotchas**

- What is (likely) wrong with the following code?

```
if (x != 0)
{
   z = a / x;
   y = b / x;
}
```

- Moral: Always use {} in if statements, even if they are not necessary

24-12: **while loops**

```
while(test)
{
   <loop body>
}
```

- Evaluate the test
- If the test is true, execute the body of the loop
- Repeat
- Loop body *may* be executed 0 times

24-13: **do-while loops**

```
do
{
   <loop body>
} while (<test>);
```

- Execute the body of the loop
- If the test is true, repeat

- Loop body is *always* executed at least once

24-14: **while vs. do-while**

- What would happen if:
  - Found a while loop in a piece of code
  - Changed to to a do-while (leaving body of loop and test the same)
- How would the execution be different?

24-15: **while vs. do-while**

- What would happen if:
  - Found a while loop in a piece of code
  - Changed to to a do-while (leaving body of loop and test the same)
- How would the execution be different?
  - If the while loop were to execute 0 times, do-while will execute (at least!) one time
  - If the while loop were to execute 1 or more times, *should* to the same thing ...
    - ... except if the test had side effects

24-16: **Side Effects**

```
class BoolFun
{
   private int size;
   boolean calledTooMuch()
   {
       return (++size > 4)
   }
}
// in main:
BoolFun bf = new BoolFun();
while (!bf.calledTooMuch())
{
   System.out.print("x");
}
```

24-17: **Side Effects**

```
class BoolFun
{
   private int size;
   boolean calledTooMuch()
   {
       return (++size > 4)
   }
}
// in main:
BoolFun bf = new BoolFun();
do
{
   System.out.print("x");
} while (!bf.calledTooMuch());
```

24-18: **for loops**

```
for (<init>; <test>; <inc>)
{
   <body>
}
```

- Equivalent to:

```
<init>
while(<test>)
{
   <body>
   <inc>
}
```

24-19: **for loops**

```
for (number = 1; number < 10; number++)
{
   System.out.print("Number is " + number);
}
```

- Equivalent to:

```
number = 1;
while(number < 10)
{
   System.out.print("Number is " + number);
   number++;
}
```

24-20: **Strings**

- Strings in Java are objects
- Contain both methods and data
  - Data is the sequence of characters (type **char**) that make up the string
  - Strings have a whole bunch of methods for string processing

24-21: **Strings**

- Strings in Java are objects
  - Strings are stored on the heap, like all other objects
  - Data is stored as an array of characters
  - Strings are immutable (once created, can't be changed)

24-22: **String Literals**

```
String s;
s = "Dog";
```

- ”Dog” is called a *String Literal*
  - Anything in quotation marks is a string literal
  - `System.out.println("Hello There")`

24-23: **String Literals**

- Any time there is a string literal in your code, there is an implicit call to “new”

- A new string object is created on the heap
- Data is filled in to match the characters in the string literal
- Pointer to that object is returned

```
String s;
s = "MyString"; // Implicit call to new here!
```

24-24: **Stack vs. Heap I**

```
public void foo()
{
    int x = 99;
    char y = 'c';
    String z = "c";
    String r = "cat";
    float w = 3.14;
}
```

24-25: **Immutable Strings**

- Strings are *immutable*
- Once you create a string, you can't change it.

```
String s = "Car";   // Create a block of memory containing 'car'
                    // Return a pointer to this block of memory

unknown.foo(s);     // This function can't mess with contents of s

System.out.println(S);   // s is guaranteed to be "Car" here
```

24-26: **Immutable Strings**

- String *objects* are immutable
    - Once a string object is created, it can't be changed
- String *variables* can be changed
    - Create a new String object, assign it to the variable

```
String s = "dog";
s = "cat";
```

24-27: **"Mutable" Objects**

```
public class ICanChange
{
    private int x;

    public ICanChange(int initialX)
    {
        this.x = initialX;
    }
    public int getX()
    {
        return this.x;
    }
    publc void setX(int newX)
    {
        this.x = newX;
    }
}
```

24-28: **"Mutable" Objects**

```
ICanChange c = new ICanChange(4);
c.setX(11);  // Changed the value in object
             //  c points to
System.out.println(c.getX());
```

- Created an object of type ICanChange
- Changed the data within that object

24-29: **“Mutable” Objects**

```
ICanChange c = new ICanChange(4);
c = new ICanChange(11);
System.out.println(c.getX());
```

- Created an object of type ICanChange, with value 4
- Created a *new* object of type ICanChange, with value 11
    - Throw away the old object

24-30: **“Mutable” Objects**

```
ICanChange c = new ICanChange(4);
StrangeClass s = new StrangeClass(); // Don’t know what this does ...

s.foo(c);

System.out.println(c.getX());
```

24-31: **“Mutable” Objects**

```
public class StrangeClass
{
    void foo(ICanChange a)
    {
        a.setX(99);
    }
}
```

24-32: **“Immutable” Object**

```
public class ICantChange
{
    private int x;

    public ICanChange(int initialX)
    {
        this.x = initialX;
    }
    public int getX()
    {
        return this.x;
    }
}
```

24-33: **“Immutable” Object**

```
ICantChange c = new ICantChange(13);
System.out.println(c.getX());
c = new ICantChange(37);
System.out.println(c.getX());
```

- Create a new object, have c point to this new object
- Old object didn’t change, but the value of c did ....

24-34: **“Immutable” Object**

```
ICantChange c = new ICantChange(13);
Strange s = new Strange();

s.foo(c);
System.out.println(c.getX());
```

- Do we know anything about what the println will ouput?

24-35: **“Immutable” Objects**

```
public class Strange
{
    void foo(ICantChange icc)
    {
       // We can't change the value of x stored in icc
       //  directly (private, no setters)
       //
       //  Best we can do is change what icc points to ...
       icc = new ICantChange(99);
       //   icc.getX() would return 99 here, but what about
       //    the calling function?
    }
}
```

24-36: **Methods**

- Classes can contain both data and methods
- When a method is called:
  - Calculate values of all method parameters (including implicit this parameter)
  - Copy values of parameters into activation record of new method
  - Execute method, using activation record for local variables
  - When method is completed, pop activation record off the stack

24-37: **Methods**

```
class MyClass           public static void main(String args[])
{                       {
   public int x;           int x;
   public int y;           MyClass c = new MyClass();
                           x = 3;
   int foo(int w)          c.x = 4;
   {                       c.y = 5;
     int q;                x = c.foo(x);
     q = x + w;            System.out.println(x);
     return q+y;         }
   }
}
```

24-38: **Methods**

```
class MyClass           public static void main(String args[])
{                       {
   public int x;           int x;
   public int y;           MyClass c = new MyClass();
                           x = 3;
   int foo(int w)          c.x = 4;
   {                       c.y = 5;
     int q;                x = c.bar(x);
     q = x + w;            System.out.println(x);
     return q+y;         }
   }
   int bar(int p)
   {
       return foo(x) + foo(p);
   }
}
```

24-39: **Collections of Data**

- We want to bundle a bunch of values together

  - Want to represent several different values using a single variable
- We can:
  - Create a class with several instance variables
  - Create an array

24-40: **Arrays**

- Arrays are objects
- Access elements using [] notation
- Need to declare the size of the array when it is created
- Can't change the size of an array once it is created
- Get the length of the array using public length instance variable

24-41: **Arrays**

- Two ways to declare arrays:

```
<typename>[] variableName;
<typename> variableName[];
```

- Examples:

```
int A[];    // A is an array of integers
int[] B;    // B is an array if integers
String C[];  // C is an array of strings
```

24-42: **Arrays: New**

- Like all other objects, Arrays are stored on the heap
- `int A[]` just allocates space for a pointer
- Need to call new to create the actual array

```
new <type>[<size>]
```

24-43: **Arrays: New**

- Show contents of memory after each line:

```
int A[];
int B[];
A = new int[10];
B = new int[5];
A[7] = 4;
B[2] = 5;
B[5] = 13;  /// RUNTIME ERROR!
```

24-44: **Arrays: Copying**

```
int A[] = new int[SIZE];
int B[] = new int[SIZE];

// Code to store data in B
A = B;
```

- What do you think this code does?
- What happens when we assign *any* object to another object?

24-45: **Arrays: Copying**

```
int A[] = new int[SIZE];
int B[] = new int[SIZE];

// Code to store data in B
A = B;
```

- How could we copy the data from B into A
- (A and B should point to different memory locations, have same values

24-46: **Arrays: Copying**

```
int A[] = new int[SIZE];
int B[] = new int[SIZE];

// Code to store data in B
for (int i = 0; i <  B.length; i++)
{
   A[i] = B[i];
}
```

24-47: **Array: Copying**

```
int A[] = new int[5];
int B[] = new int[5];
int C[];

for (int i = 0; i < 5; i++)
    A[i] = i;

for (int i = 0; i < 5; i++)
    B[i] = A[i];

C = A;

B[2] = 10;
C[2] = 15;
```

24-48: **Arrays of Objects**

- We can have arrays of objects, as well as arrays of integers

```
...
Point pointArray[] = new Point[10];
pointArray[3].setX(3);
```

- What happens?
  - (refer to Java documentation for Point objects)

24-49: **Arrays of Objects**

```
Point pointArray[] = new Point[10];

for (int i = 0; i < 10; i++)
{
   pointArray[i] = new Point(i, i);
}
```

- How would you calculate the average `x` value of all elements in the array?

24-50: **Arrays of Objects**

- How would you calculate the average `x` value of all elements in the array?

```
Point pointArray[] = new Point[10];

// Fill in pointArray
//

double sum = 0.0;
for (int i = 0; i < pointArray.length; i++)
{
   sum = sum + pointArray[i].getX();
}
sum = sum / pointArray.length;
```

24-51: **2D Arrays**

- We can create 2D arrays as well as 1D arrays
  - Like matrices
- 2D array is really just an array of arrays

24-52: **2D Arrays**

```
int x[][];   // Declare a 2D array
int[][] y;   // Alternate way to declare 2D array

x = new int[5][10];  // Create 50 spaces
y = new int[4][4];   // create 16 spaces
```

24-53: **2D Arrays**

```
int x[][];           // Declare a 2D array
x = new int[5][5];   // Create 25 spaces

x[2][3] = 11;
x[3][3] = 2;
x[4][5] = 7;   // ERROR! Index out of bounds
```

24-54: **2D Arrays**

- How would we create a 9x9 array, and set every value in it to be 3?

24-55: **2D Arrays**

- How would we create a 9x9 array, and set every value in it to be 3?

```
int board[][];
board = new int[9][9];
for (int i = 0; i < 9; i++)
   for int (j = 0; j < 9; j++)
       board[i][j] = 3;
```

24-56: **Using Arrays**

- Need to declare array size before using them
- Don't always know ahead of time how big our array needs to be
- Allocate more space than we need at first
- Maintain a second size variable, that has the number of elements in the array we actually care about
- Classes that use arrays often will have an array instance variable, and a size instance variable (how much of the array is used)

24-57: **Arrays**

- On board: What memory looks like for the following:

```
public static void main(String args)
{
    int A[];
    int x;
    String B[];      <-- Show memory here
    A = new int[5];
    B = new String[3];
                     <-- Show memory here
}
```

24-58: **Recursion**

- The way function calls work give us a fantastic tool for solving problems
    - Make the problem slightly smaller
    - Solve the smaller probelm *using the very function that we are writing*
    - Use the solution to the smaller problem to solve the original problem

24-59: **Recursion**

- What is a really easy (small!) version of the problem, that I could solve immediately? (Base case)
- How can I make the problem smaller?

- Assuming that I could magically solve the smaller problem, how could I use that solution to solve the original problem (Recursive Case)

24-60: **Recursion**

- Example: Factorial
  - $n! = n * (n-1) * (n-2) * ... * 3 * 2 * 1$
  - $5! = 5 * 4 * 3 * 2 * 1 = 120$
  - $8! = 8 * 7 * 6 * 5 * 4 * 3 * 2 * 1 = 40320$
- What is the base case? That is, a small, easy version of the problem that we can solve immediately?

24-61: **Recursion – Factorial**

- Example: Factorial
  - $n! = n * (n-1) * (n-2) * ... * 3 * 2 * 1$
- What is a small, easy version of the problem that we can solve immediately?
  - 1! == 1.

24-62: **Recursion – Factorial**

- How do we make the problem smaller?
  - What's a smaller problem than $n!$?
  - (only a *liitle* bit smaller)

24-63: **Recursion – Factorial**

- How do we make the problem smaller?
  - What's a smaller problem than $n!$?
  - $(n-1)!$
- If we could solve $(n-1)!$, how could we use this to solve $n!$?

24-64: **Recursion – Factorial**

- How do we make the problem smaller?
  - What's a smaller problem than $n!$?
  - $(n-1)!$
- If we could solve $(n-1)!$, how could we use this to solve $n!$?
  - $n! = (n-1)! * n$

24-65: **Recursion – Factorial**

```
int factorial(int n)
{
   if (n == 1)
   {
      return 1;
   }
   else
   {
      return n * factorial(n - 1);
   }
}
```

24-66: **Recursion – Factorial**

- 0! is defined to be 1
- We can modify `factorial` to handle this case easily

24-67: **Recursion – Factorial**

- 0! is defined to be 1
- We can modify `factorial` to handle this case easily

```
int factorial(int n)
{
   if (n == 0)
   {
      return 1;
   }
   else
   {
      return n * factorial(n - 1);
   }
}
```

24-68: **Recursion**

- To solve a recursive problem:
  - Base Case:
    - Version of the problem that can be solved immediately
  - Recursive Case
    - Make the problem smaller
    - Call the function recursively to solve the smaller problem
    - Use solution to the smaller problem to solve the larger problem

24-69: **Recursion – Tips**

- When writing a recursive function
  - Don't think about how the recursive function works all the way down
  - Instead, *assume that the function just works for a smaller problem*
    - Recursive Leap of Faith
  - Use the solution to the smaller problem to solve the larger problem

24-70: **Recursion – NumDigits**

- Write a method that returns the number of base-k digits in a number n
  `int numDigits(int n, int k)`
  - `numDigits(20201, 10) == 5`

- `numDigits(34, 10) == 2`
- `numDigits(3050060,7) == 7`
- `numDigits(137, 2) == 8`

- What is the base case?
- How can we make the problem smaller?
- How can we use the solution to the smaller problem to solve the original problem?

24-71: **Recursion – NumDigits**

```
int numDigits(int n, k)
{
   if (n < k)
   {
      return 1;
   }
   else
   {
      return 1 + numDigits(n / k);
   }
}
```

24-72: **Recursion Problems**

- Write a recursive function that returns the smallest value in the first `size` elements of an array of integers
  - `int minimum(int A[], int size)`

24-73: **Recursion Problems**

```
int minimum(int A[], int size)
{
   if (size == 0)
      return null;
   if (size == 1)
      return A[0];
   int smallest = minimum(A, size - 1);
   if (smallest < A[size - 1])
      return smallest;
   else
      return A[size - 1];
}
```

24-74: **Static**

- Normally, can only call methods on classes when we created an instance of the class
  - Methods can rely on instance variables to work properly
  - Need to create an instance of a class before there are any instance variables
  - What would the size() method return for an ArrayList if there was not an instance to check the size of?

24-75: **Static**

- Some methods don't operate on an instance of the class – pure functions that don't use instance variables at all
  - Math functions like min, or pow
  - parseInt – takes a string as an input parameter, and returns the integer value of the string parseInt("123") returns 123
- Seems silly to have to instantiate an object to use these methods
- static to the rescue!

24-76: **Static**

- If we declare a method as static, it does not rely on an instance of the class
- Can call the method without creating an instance first
- Use the *Class Name* to invoke (call) the method

```
double x  = Math.min(3.4, 6.2);
double z = Math.sqrt(x);
```

24-77: **Consants**

- Having 3.14159 appearing all over your code is considered bad style
  - Could end up using different values for pi in different places (3.14159 vs. 3.1415926)
  - If you want to change the value of pi (to add more digits, for instance), need to search through all of your code to find it
- In general, any time you have a "magic number" (that is, an arbitrary numeric literal) in your code, it should probably be a symbolic constant instead.
- The "final" modifier is used to prevent you from changing the value of a variable

24-78: **Consants**

```
class Calendar
{
    final int MONTHS_IN_YEAR = 12;
    final int DAYS_IN_WEEK = 12;
    final int DAYS_IN_YEAR = 365;

    // Methods that use the above constants
}
```

- Every instance of class Calendar will contain those 3 variables
- Somewhat wasteful
- Need to instantiate an object of type Calendar to access them

24-79: **Consants: Static**

- We can declare variables to be static as well as methods
- Typically used for constants (Math.pi, Math.e)
- Access them using class name, not instance name (just like static methods)
- You should *only* use static variables for constants
  - public static final float pi = 3.14159;

24-80: **Globals: Static**

- It is technically possible to have a variable that is public and static, but not final
  - Can be accesed anywhere
  - Can be *changed* anywhere
- While the compiler will allow it, this is (usually!) a *very bad* idea. Why?