Intro to Programming II
Review

Chris Brooks

Department of Computer Science
University of San Francisco
So what have we learned about this year?
Scope refers to the area of a program where a variable can be accessed.

Java has three types of scope:

- Local scope - the variable exists only within a method.
- Object scope - the variable can be accessed from any method belonging to an object.
- Class scope - the variable can be accessed by all instances of a class.
Parameters are the variables passed into a method.

We can talk about:

- Formal parameters - these are the variables named in the method definition.
- Actual parameters - these are the variables in the method invocation.
/** This is a method definition */
public double depositFunds(double amt) {
    balance = balance + amt;
    return balance;
}

bankacct b = new bankacct();
paycheck = 100.0
/* this is a method invocation */
b.depositFunds(paycheck);
28-4: Method signature

- Specification of all data types coming in and out of a method.
  - Type and order of input parameters
  - Type of return variable

- A method signature allows the compiler to uniquely identify a method.
Consider the following method declaration:

double calculate(double a1, double a2, double a3);

Which of the following are valid calls to this method?

- calculate(3, 52.0, -5.1);
- double y = calculate(0, 1.1, 2.2);
- calculate(1.1, 2.3);
- calculate(“Hello”, 4.4, 2);
- calculate();
- calculate(3.3);
An object is a type of abstraction

It provides a way of grouping together related data and functionality.

Makes it easier to organize and extend your program.

Also gives a “black box” effect.

Users of your objects don’t need to worry about how they work internally, just how to use them.
28-7: Classes and objects

6 A class is a template or category
6 An object is a particular instance of that class.
   △ CartoonAnimals might be a class
   △ BugsBunny, Tweety are instances of that class
6 Classes let us specify behavior common to a set of objects.
As we know, classes also contain methods. Methods are pieces of code that can be invoked on an object. They allow us to encapsulate both state and behavior.
It’s also important to protect instance data from outside users.

One way to do this is by providing accessors and mutators “setters and getters”

Rather than the user modifying your object’s data directly, they use a method to do it.

- Reduces error
- Hides implementation from the user.
A constructor is a method that is called when an object is first created. Its responsibility is to initialize an object’s instance variables. It must have the same name as the class in constructs. It has no return type. It is called when 'new' is invoked.
public class Point {
    public int xval;
    public int yval;

    Point(int x, int y) {
        xval = x;
        yval = y;
    }
}

Point p = new Point(3, 4);
Multiple Constructors

It’s often helpful to be able to specify some instance variables, but not all.

For example, let’s say our circle class has a default radius of 1.

If the radius is something else, users can specify it.

Point p1 = new Point(3,3); Circle c = new Circle(p); creates a circle with radius 1 at (3,3).

Point p1 = new Point(3,3); Circle c = new Circle(p,5); creates a circle with radius 5 at (3,3).
public class Circle {
    private Point center;
    private int radius;

    public Circle(Point c) {
        center = c;
        radius = 1;
    }

    public Circle(Point c, int r) {
        center = c
        radius = r;
    }
}
Strings in Java are objects

This mean that they have a set of methods they respond to:

- `compareTo()`, `equals()`
- `indexOf()`
- `length()`
- `replace()`
- `startsWith()`, `endsWith()`
- etc
Unlike most other objects, Strings also have special behavior for creating string literals and for concatenation.

String literals are strings where the value is known at compile time:
- String s1 = “hello world”
- String s2 = “USF”
- String s3 = “I love Java”

We can create a string without calling new.
We can also use the ‘+’ symbol to concatenate strings.

- String s1 = “hello”
- String s2 = “world”
- String s3 = s1 + s2 // s3 = “hello world”

This is a phenomenon called *overloading*; an operator is redefined to provide different functionality.
We can use the Scanner class to read from a file instead of System.in

```java
try {
    Scanner sc = new Scanner(new File("studentlist"));
    while (sc.hasNext()) {
        System.out.println(sc.next());
    }
} catch (FileNotFoundException e) {
    System.out.println("File not found.");
}
```
We can use Scanner to also read from a file.

Relevant methods:
- hasNext()
- next()
- nextLine()
- nextInt()
- ...
We can use the Scanner class to read from a file instead of System.in

```java
try {
    Scanner sc = new Scanner(new File("studentlist"));
    while (sc.hasNext()) {
        System.out.println(sc.next());
    }
} catch (FileNotFoundException e) {
    System.out.println("File not found.");
}
```
Output is a little more complicated.
- No equivalent of the Scanner class.
- PrintWriter is the thing to use.
import java.io.*;

public class printtester {
    public static void main(String args[]) {
        try {
            PrintWriter p = new PrintWriter("foo");
            p.println("hello");
            p.println();
            p.close();
        } catch (FileNotFoundException e) {
            System.out.println("file not found");
        }
    }
}
More detailed tracing

Box-and-arrow tracing is nice, but too high-level sometimes.

Doesn’t let us keep track of how memory is allocated.

We will also do more detailed tracing of programs to see what’s actually happening.
The run-time environment refers to the way in which memory is used/arranged.

Memory is arranged as a sequence of addresses.

Each address refers to a word in memory.

We can break the runtime environment into four sections:
- Program code: Where the program itself resides.
- Global data area: Global and static data is stored here.
- Run-time stack: This contains an activation record for each method that is called.
- Heap: Dynamically-allocated data (with new or malloc) is stored here.
An activation record sets a context for a method’s execution.

It contains:
- Space for all parameters, including ‘this’, a pointer to the object itself.
- Space for a return value
- Space for local variables.

Each time a method is called, its activation record is pushed onto the stack.

When the method exits, its activation record is removed.
The *symbol table* is responsible for mapping variable names to addresses. This is how the Java interpreter knows the value that is currently associated with a variable.
Simplifying assumptions:
- Code section begins at address 0
- Global data at 1000
- Runtime stack starts at 2000
- Heap starts at 5000
- Integers get 4 bytes
- Chars get 2 bytes
- Floats get 8 bytes
Java provides built-in support for *arrays*

- An array is a set of objects that are sequentially arranged in memory.
- Size of the array is declared when it is created.

**Example:**

```java
int [] atBats;
int [] runs[];
```

This declares the array, but not its contents.
We need to use 'new' to actually allocate memory for the array.

```java
int x = sc.nextInt();
atBats = new int[10];
rungs = new int[x];
```

This means that the reference to the array will be allocated on the stack, while the array itself will be allocated on the heap.
Advantages:
- All memory is contiguous
- Can 'jump' directly to any element of the array.

Disadvantages:
- Hard to resize or add elements.
Last time, we learned about how to use arrays in Java.

Java also provides an ArrayList class that can help manage arrays.

ArrayList is a generic container.

That means that we can use the same container to store different kinds of elements.
An example:

```java
ArrayList band = new ArrayList(3);
band.add(``john'');
band.add(``paul'');
band.add(``george'');
band.add(``ringo'');
```
Notice that we declared an initial array size.

The array is then able to grow *dynamically* beyond that.

It’s better to allocate in advance if we can.

We can also remove elements, and access them.

We can also add in the middle of a list: `band.add(1, \"ringo\")`;
get(index) lets us access the element at a particular index.

Elements in an ArrayList are stored as *Objects*.

This means that we need to cast them back to Strings.

Can’t store primitives.

```java
String name = (String) band.get(2);
```
We can use indexOf to find where an element is located.

```java
int index = band.indexOf(``ringo'')
band.remove(index)
```
Linked lists have the opposite advantages and disadvantages

Advantages:
- Easy to insert and remove
- Easy to resize

Disadvantages:
- Elements are not stored sequentially
- Finding the nth element is slower.
The general idea:

Each element of the list will “know” who the next element is.

Let’s try this as a class.
So how do we do this in Java?

Our Element class needs to have two components:
- The data that we want to store in the list.
- A pointer to the next element in the list.
public class ListItem {
    public Object data;
    public ListItem next;

    public ListItem(Object d) {
        data = d;
        next = null;
    }
}
ListItems hook together like a chain.

All we need to do is keep track of the beginning of the chain.

No need to allocate everything ahead of time.
The LinkedList will be responsible for hanging onto the 'head' of the list and providing methods for working with the list.

- Insert()
- InsertAt(index)
- get(index)
- remove(index)
- find(object)
public class LinkedList {
    public ListItem head = null;
    public void insert(Object o) { ... }
    public void insertAt(Object o, int index) { ... }
    public Object get(int index) { ... }
    public Object remove(int index) { ... }
    public int find(Object o) { ... }
}
So how do we add something to the front of a linked list?

- Have the new thing point to the currently-first ListItem
- Point 'head' to our new thing.

```java
public void insert(Object o) {
    ListItem l = new ListItem(o);
    l.next = head;
    head = l;
}
```
What about adding something into the middle of a list? If we want an item to go between current elements 5 and 6, then we need our new item to point to 6, and 5 to point to the new element. How do we code this?
public void insertAt(Object o, int index) {
    // first find the place to insert it.
    ListItem pointer = head;
    ListItem l = new ListItem(o);
    for (int i = 0; i < index -1; i++) {
        pointer = pointer.next;
    }
    l.next = pointer;
    pointer = l;
}
The programs you’ve built so far (lexer and parser) are examples of synchronous input.

△ You prompt for input, then read input with a Scanner.

Programs with a graphical user interface (GUI) typically require asynchronous input

△ A user can provide input at any time.

This requires a different model of programming.
A GUI consists of:
- Components
- Events
- Listeners
Components generate events (usually in response to user input)

Listeners wait for and handle these events

Typically by invoking a method.
Inheritance Review

1. Inheritance allows us to reuse existing code.
2. Allows us to define a hierarchy of classes.
3. *Base class* has the most general behavior
4. *Derived classes* have more specific behavior.
public class Person
{
    public String lastName;
    public String id;
    public void eat() { };
    public void sleep() { };
};
public class Professor extends Person{
    public String officeNum;
    public void teach() {}
    public void grade() {}
    public void forget() {}
}
What if we wanted to make a Student class that was a subclass of Person?
△ methods attendClass, doHomework()

What if we also wanted to make a GradStudent that was a subclass of Student?
△ New instance variable: public Professor advisor.

Note that Student is an *is-a* relationship, and advisor is a *has-a* relationship.
On Monday, you did a lab in which you created a Shape class

△ This had an area() method

You then subclassed it with Circle and Rectangle classes.

One problem with this: you may not want users to ever create Shapes

△ You just want anything that inherits from Shape to have an area() method.
Solution: define Shape as an *abstract class*

```java
public abstract class Shape {
    public int locX;
    public int locY;
    public abstract double area(); // note: semi-colon, no method body
}
```
An abstract class is one that has one or more abstract methods.

- Can also have concrete methods.

Classes that subclass from an abstract class must override all abstract methods.

An abstract class therefore provides a common *interface* for a set of subclasses
What if we have this situation:

```java
public class A {
    public void m1() {
    }
    public void m2() {
    }
}

public class B extends A {
    public void m2() {
    }
}
```

B bex = new B();
A aex = new B();
bex.m1();
aex.m1();
bex.m2();

Which methods are called?
Dynamic binding

Java resolves this via *dynamic binding*

- The actual type of an object is determined at runtime.
- That object’s class is searched for the corresponding method.
- If the method doesn’t exist in that class, the parent class is checked.

What is the advantage of dynamic binding?

What is the disadvantage?
Interfaces allow us to specify methods that an object is guaranteed to respond to, without specifying an implementation.

A class can implement as many interfaces as it wants.

```java
public interface FlyingThing {
    public void fly();
}

public class Bat extends Animal implements FlyingThing {
    public void fly() {
        System.out.println("I'm flying!");
    }
}
Interfaces let us specify which methods an object should respond to, without specifying how they should respond.

This provides polymorphism - each object responds to a method in the appropriate way.

A class can implement as many interfaces as it wants.
Recursion is a fundamental problem-solving technique

- Involves decomposing a problem into:
  - A base case that can be solved directly
  - A recursive step that indicates how to handle more complex cases.

A common recursive example is factorial:

```java
long factorial(int input)
    if (input == 1)
        return 1;
    else
        return input * factorial(input - 1);
```
A more interesting example is the Towers of Hanoi. It’s hard to write an iterative program to solve this, but the recursive version is startlingly simple:

```c
void towers(int ndisks, Tower startTower, Tower goalTower, Tower tempTower)
{
    if (ndisks == 0)
        return;
    else
        towers(ndisks - 1, startTower, tempTower, goalTower);
    moveDisk(startTower, goalTower);
    towers(ndisks - 1, tempTower, goalTower, startTower);
}
```
Trees are a useful recursive data structure.

If we keep them sorted, we can find elements more quickly than in a list.

examples:
28-62: Tree Terminology

- Parent / Child
- Leaf node
- Root node
- Edge (between nodes)
- Path
- Ancestor / Descendant
- Depth of a node \( n \)
  - Length of path from root to \( n \)
- Height of a tree
  - (Depth of deepest node) + 1
Implementing a tree

Treenode class:

```java
public class TreeNode {
    private Comparable data;
    private TreeNode left;
    private TreeNode right;

    public Object getData() { return data; }
}
```
Binary Trees

For each node n, (value stored at node n) > (value stored in left subtree)

For each node n, (value stored at node n) < (value stored in right subtree)
public TreeNode insert(TreeNode tree, Comparable elem) {
    if (tree == null) {
        return new TreeNode(elem);
    } else {
        if (elem.compareTo(tree.getData()) < 0) {
            tree.setLeft(insert(tree.left(), elem));
            return tree;
        } else {
            tree.setRight(insert(tree.right(), elem));
            return tree;
        }
    }
}
How long does it take to:

- Find the nth element in a linked list.
- Find the nth element in an array.
- Insert an object at the front of a linked list
- Insert an object at the front of an array.
- Remove an element from a linked list
- Remove an element from and array.
So, how long does it take to find something in a tree?

- What if the tree is perfectly balanced?
- What if it’s completely unbalanced?
So how would we count the number of nodes in a tree?
If the tree is null, return 0.

Otherwise, return 1 + the number of nodes in the left subtree + the number of nodes in the right subtree.

Exercise: add a countNodes() method to our TreeNode class.
C is a compiled language
- Produces a binary that executes on one architecture/OS

Java compiles to an intermediate representation (bytecodes)
- A Java program can be executed by a Java interpreter on any system.
C is not object-oriented

- Functions rather than methods
- No classes or objects.
- Structs can be used to group data, but not to associate methods.
28-72: Things that are the same in C and Java

- primitive types
  - int, char, double (no boolean, though)
- if/else
  - including && and ||
- while
- for
- blocks delimited with { }
- comments are /* ... */
28-73: *Things that are different in C and Java*

- No built-in String class
- No classes/objects/methods
- Memory allocation
- No garbage collection
- Much fewer standard libraries
- Java has references; C allows you to directly manipulate pointers.
The biggest difference between C and Java is the use of pointers.

A pointer is the actual address that a variable is stored at.

```c
int main(void) {
    char *testStr = "hello world";
    printf("\%s\n", testStr);
    printf("\%d\n", *testStr);
    printf("\%d\n", *testStr + 3);
}
```
To declare a pointer to a variable, use *

```c
char *hello = “hello world”;
```

To get at the data, use the variable name

```c
printf(“%s”, hello);
```

To *dereference* the data and use the address, add a the ‘*’

to the front of the variable name.

```c
printf(“%d”, *hello);
```
28-76: Dynamically allocating objects

- In Java, you allocate objects by using new.
- In C, you use malloc
  - Big difference: you need to specify the size of the memory chunk you want and the number of chunks.

```c
/* make an array of 10 integers */
int *i = (int *)malloc(10, sizeof(int));
```
6 In Java, primitives are passed by value.

6 With references, it’s a little trickier.
   △ For objects, a copy of the reference is passed.
   △ This means that methods called on that reference affect the same global object.
   △ But, if the reference itself is changed, only the local copy is affected.
In C, things are more straightforward.

Variables are passed by value.

To pass by reference, you can provide a reference to the variable using &.
Let’s start with a variable:

\[\text{int } x\]

To refer to the address that \(x\) is stored at, we use the address operator

\[&x\]

To create a pointer to an integer, we use the \(*\) operator:

\[\text{int } \ast \text{iptr;}\]

\(\text{iptr}\) is a variable of type \(\text{int } \ast\) - that is, it’s the address of an integer.

So, we can do:

\[\text{iptr }= \&x;\]

(the variable \(\text{iptr}\), which is an address of an \(\text{int}\), is set to the address of \(x\), which is an \(\text{int}\) )
So, & is used to get from a variable to its address.

- `scanf`
- calling a function with pass by reference

* is used when you want to start with an address, and then create a variable.

- dynamically allocating memory
- being called with pass by reference
In Java, you create new data types by creating classes.

- Classes have member variables and associated methods.
- You can control access, and inherit.

C has structs.

- Member variables only (no methods)
- No means of hiding information (public/private)
typedef struct {
    char name[80];
    int id;
    char DOB[80];
} Person;

Notice:
△ typedef - this declares a new type, which is a struct.
△ The name of the new type is after the definition
△ Ends with a semicolon.
Many times, you want to have an array with more than one dimension.

- A 2D game board.
- An array of strings.
- A bitmap representing a graphic object.

In C, this is represented as an array of arrays.
What if we don’t know ahead of time how big our array should be? Then we need to use malloc to allocate memory on the fly. In this case, we treat our 2D array as an array of pointers (or, an array of arrays)

```
int **intArray;
```

intArray is a pointer to an array of pointers.
We start out using malloc as usual (almost):

```c
int **intArray = (int **)malloc(10 * sizeof(int *));
```

But, none of those pointers point to anything yet.

We have to go through and use malloc to allocate space for each of those arrays as well.

```c
for (i = 0; i < 10; i++)
    intArray[i] = (int *)malloc(10 * sizeof(int));
```
6 In C, you work with files by accessing a file pointer.

6 This is declared like this:

```c
FILE *fptr;
```

(notice the caps)
fopen() opens a file and returns a file pointer.

It takes two arguments:
- The file name
- A string indicating whether we’re opening for reading or writing.
For example

FILE *fp = fopen("myfile","r");

opens myfile for reading.
fscanf is used to read from a file.

Works exactly like scanf, except that the first argument is the file pointer.

You can also use getc - it returns the integer representing the next character in the file.

getc() returns EOF if you’re at the end of the file.
To do this, we need to specify that main() will receive arguments.

Remember, main is just another function.

It takes its arguments in a special form:

```c
int main(int argc, char **argv)
```

- argc is the number of command line arguments
- argv is an array of strings, one for each argument.
- argv[0] is the name of the program.