

# Intro to Programming II

## Review

Chris Brooks

Department of Computer Science  
University of San Francisco

Department of Computer Science — University of San Francisco — p. 1/77

### 25-2: Slides

- So what have we learned about this semester?

Department of Computer Science — University of San Francisco — p. 2/77

### 25-3: Scope

- 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.

Department of Computer Science — University of San Francisco — p. 3/77

### 25-4: Parameters

- 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.

Department of Computer Science — University of San Francisco — p. 4/77

### 25-5: Example

```
/** 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);
```

Department of Computer Science — University of San Francisco — p. 5/77

### 25-6: 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.

Department of Computer Science — University of San Francisco — p. 6/77

### 25-7: Example

- 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);`

### 25-8: Objects

- 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.

### 25-9: Classes and objects

- A class is a template or category
- An object is a particular instance of that class.
  - `CartoonAnimals` might be a class
  - `BugsBunny`, `Tweety` are instances of that class
- Classes let us specify behavior common to a set of objects.

### 25-10: Methods

- 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*.

### 25-11: Data hiding

- 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.

### 25-12: Constructors

- 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.

### 25-13: Example

```
public class Point {
    public int xval;
    public int yval;

    Point(int x, int y) {
        xval = x;
        yval = y;
    }
}
Point p = new Point(3,4);
```

### 25-14: 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).

### 25-15: Multiple Constructors

```
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;
    }
}
```

### 25-16: Strings in Java

- Strings in Java are objects
- This means that they have a set of methods they respond to:
  - `compareTo()`, `equals()`
  - `indexOf()`
  - `length()`
  - `replace()`
  - `startsWith()`, `endsWith()`
  - etc

### 25-17: Overloaded operators

- 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`.

### 25-18: Overloaded operators

- 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.

### 25-19: Using Scanner to read from files

- We can use the Scanner class to read from a file instead of System.in

```
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.");
}
```

### 25-20: Working with Files

- We can use Scanner to also read from a file.
- Relevant methods:
  - hasNext()
  - next()
  - nextLine()
  - nextInt()
  - ...

### 25-21: Using Scanner to read from files

- We can use the Scanner class to read from a file instead of System.in

```
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.");
}
```

### 25-22: File output

- Output is a little more complicated.
  - No equivalent of the Scanner class.
- PrintWriter is the thing to use.

### 25-23: File Output

```
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");
        }
    }
}
```

### 25-24: 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.

## 25-25: Run-time environment

- 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.

## 25-26: Activation Records

- 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.

## 25-27: Symbol table

- 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.

## 25-28: Activation Records

- 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

## 25-29: Arrays

- 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:

```
int []atBats;
int runs[]
```
- This declares the array, but not its contents.

## 25-30: Arrays

- We need to use 'new' to actually allocate memory for the array.

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

### 25-31: Advantages and disadvantages of arrays

- Advantages:
  - All memory is contiguous
  - Can 'jump' directly to any element of the array.
- Disadvantages:
  - Hard to resize or add elements.

### 25-32: The ArrayList class

- 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.

### 25-33: The ArrayList class

- An example:

```
ArrayList band = new ArrayList(3);
band.add("john");
band.add("paul");
band.add("george");
band.add("ringo");
```

### 25-34: The ArrayList class

- 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");`

### 25-35: Accessing elements

- `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.

```
String name = (String)band.get(2);
```

### 25-36: Finding elements

- We can use `indexOf` to find where an element is located.
- `remove` lets us remove things.

```
int index = band.indexOf("ringo");
band.remove(index);
```

### 25-37: Linked Lists

- 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.

Department of Computer Science — University of San Francisco — p. 37/77

### 25-38: Linked Lists

- The general idea:
- Each element of the list will “know” who the next element is.
- Let’s try this as a class.

Department of Computer Science — University of San Francisco — p. 38/77

### 25-39: List elements

- 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.

Department of Computer Science — University of San Francisco — p. 39/77

### 25-40: List elements

```
public class ListItem {
    public Object data;
    public ListItem next;

    public ListItem(Object d) {
        data = d;
        next = null;
    }
}
```

Department of Computer Science — University of San Francisco — p. 40/77

### 25-41: Arranging ListItems

- 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.

Department of Computer Science — University of San Francisco — p. 41/77

### 25-42: The LinkedList class

- 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)

Department of Computer Science — University of San Francisco — p. 42/77

### 25-43: The LinkedList class

```
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) { ... }
}
```

### 25-44: Adding

- 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.

```
public void insert(Object o) {
    ListItem l = new ListItem(o);
    l.next = head;
    head = l;
}
```

### 25-45: Adding

- 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?

### 25-46: Adding

```
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;
}
```

### 25-47: Synchronous vs Asynchronous input

- 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.

### 25-48: GUI parts

- A GUI consists of:
  - Components
  - Events
  - Listeners

## 25-49: GUI parts

- Components generate events (usually in response to user input)
- Listeners wait for and handle these events
- Typically by invoking a method.

## 25-50: Inheritance Review

- Inheritance allows us to reuse existing code.
- Allows us to define a hierarchy of classes.
- *Base class* has the most general behavior
- *Derived classes* have more specific behavior.

## 25-51: Example

```
public class Person
{
    public String lastName;
    public String id;
    public void eat() { };
    public void sleep() { };
}
```

## 25-52: Example

```
public class Professor extends Person
{
    public String officeNum;
    public void teach() { };
    public void grade() { };
    public void forget() { };
}
```

## 25-53: Example

- 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.

## 25-54: Abstract classes

- 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.

## 25-55: Abstract classes

- Solution: define Shape as an *abstract class*

```
public abstract class Shape
{
    public int locX;
    public int locY;
    public abstract double area(); // note: semi-colon, no method body
}
```

## 25-56: Abstract classes

- 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

## 25-57: Dynamic binding

- What if we have this situation:

```
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?

## 25-58: 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?

## 25-59: Interfaces

- *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.

```
public interface FlyingThing {
    public void fly();
}
public class Bat extends Animal implements FlyingThing {
    public void fly() {
        System.out.println("I'm flying!");
    }
}
```

## 25-60: Interfaces

- 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.

## 25-61: Recursion

- 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:

```
long factorial(int input)
if (input == 1)
    return 1;
else
    return input * factorial(input - 1);
```

## 25-62: Recursion

- 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:

```
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);
```

## 25-63: Trees

- Trees are a useful recursive data structure.
- A tree is either a leaf, or it's a tree with left and right children which are also trees.
- Almost every tree algorithm uses recursion.

## 25-64: 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

## 25-65: Implementing a tree

- A struct representing a tree containing strings:

```
typedef struct treeNode *treeptr;

typedef struct treeNode {
    char *data;
    treeptr left;
    treeptr right;
} treeNode;
```

- We need to declare a type called "treeptr", because otherwise the C compiler doesn't know what a treeNode is until the definition is processed.

## 25-66: Binary Search Trees

- 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)

### 25-67: Adding methods to BST

```
void insert(treenodeptr root, char *newdata) {
    treeNode *newNode;

    if (strcmp(root->data, newdata) <= 0) {
        if (root->left == NULL) {
            newNode = makeNode(newdata);
            root->left = newNode;
            return;
        } else {
            return insert(root->left, newdata);
        }
    } else {
        if (root->right == NULL) {
            newNode = makeNode(newdata);
            root->right = newNode;
            return;
        } else {
            return insert(root->right, newdata);
        }
    }
}
```

Department of Computer Science — University of San Francisco — p. 67/77

### 25-68: Lists and arrays

- 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 an array.

Department of Computer Science — University of San Francisco — p. 68/77

### 25-69: Searching in trees

- 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?

Department of Computer Science — University of San Francisco — p. 69/77

### 25-70: Counting nodes

- So how would we count the number of nodes in a tree?

Department of Computer Science — University of San Francisco — p. 70/77

### 25-71: Counting nodes

- 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.

Department of Computer Science — University of San Francisco — p. 71/77

### 25-72: Introduction to C

- 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.

Department of Computer Science — University of San Francisco — p. 72/77

## 25-73: Introduction to C

- 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.

## 25-74: 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 /\* ... \*/

## 25-75: 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.

## 25-76: 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.

```
int
main(void) \{
    char *testStr = "hello world";

    printf("\%s\n", testStr);
    printf("\%d\n", *testStr);
    printf("\%d\n", *testStr + 3);
\}
```

## 25-77: Pointers

- To declare a pointer to a variable, use \*
  - char \*hello = "hello world";
- To get at the data, use the variable name
  - printf("%S", hello);
- To *dereference* the data and use the address, add a the '\*' to the front of the variable name.
  - This produces an integer.
  - printf("%d", \*hello);

## 25-78: 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.

```
/* make an array of 10 integers */
int *i = (int *)malloc(10, sizeof(int));
```

## 25-79: Parameters in Java

- In Java, primitives are passed by value.
- 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.

## 25-80: Parameters in C

- In C, things are more straightforward.
- Variables are passed by value.
- To pass by reference, you can provide a reference to the variable using &.

## 25-81: References vs Pointers

- Let's start with a variable:
  - `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:
  - `int *iptr;`
- `iptr` is a variable of type `int *` - that is, it's the address of an integer.
- So, we can do:
  - `iptr = &x;`
  - (the variable `iptr`, which is an address of an int, is set to the address of `x`, which is an int.)

## 25-82: References vs Pointers

- 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

## 25-83: Structs

- 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)

## 25-84: Structs example

```
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.

## 25-85: Multidimensional Arrays

- 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.

## 25-86: Arrays of pointers

- 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.

## 25-87: Arrays of pointers

- We start out using malloc as usual (almost):

```
int **intArray = (int **)malloc(10 * sizeof(int *));
```
- We used malloc to create an array of 10 int *pointers*
- 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.

```
for (i = 0; i < 10; i++)
    intArray[i] = (int *)malloc(10 * sizeof(int));
```

## 25-88: Working with files

- In C, you work with files by accessing a file pointer.
- This is declared like this:

```
FILE *fptr;
```

(notice the caps)

## 25-89: Opening files for reading

- 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.

## 25-90: Opening files for reading

- For example

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

opens myfile for reading.

#### 25-91: Reading from a file

- 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.

#### 25-92: Working with command line arguments

- 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:

```
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.