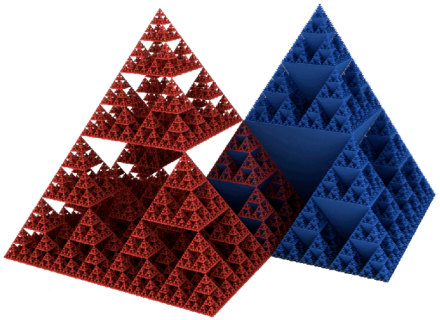
COMPSCI 101
Principles of Programming

Lecture 12

Return statements

Comparing Strings

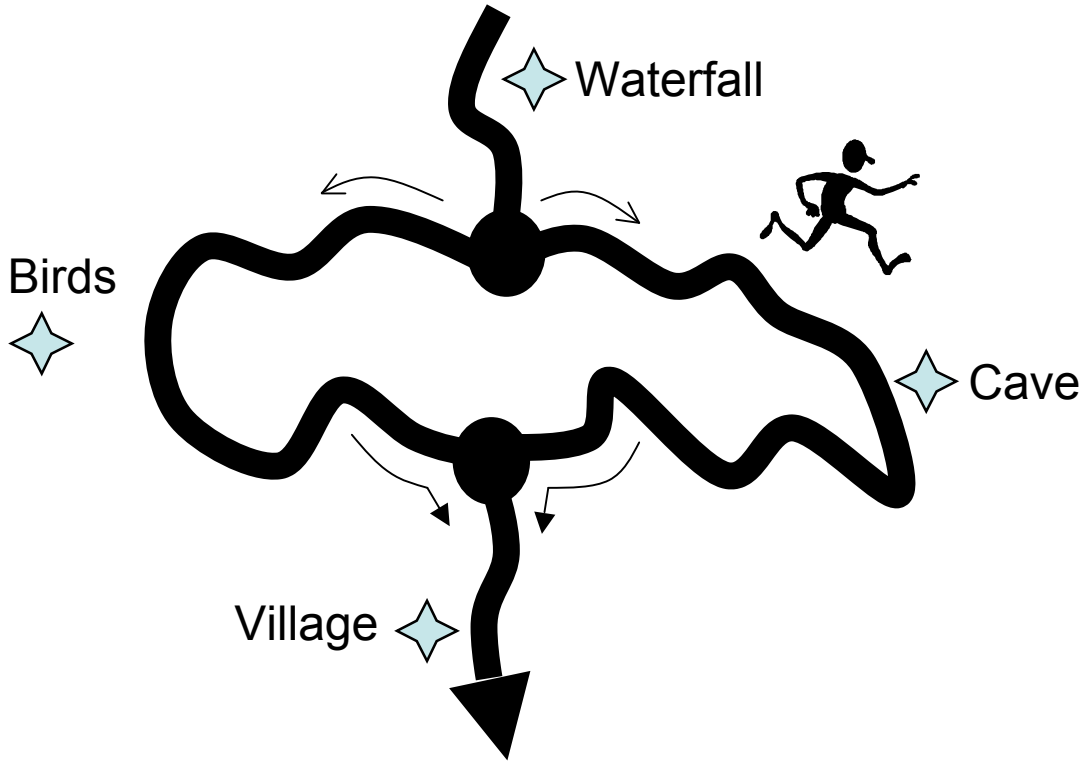
Assignment 3 Preview



Review of If Statements

Choosing between 2 options

Choose one option or the other, but not both.



`if-else` statements

Discussed so far:

- If “something” is true then do “this”

Using an optional else clause

- If “something” is true then do “this” otherwise do “that”

```
public void start() {
        System.out.println( “Waterfall” );

        if (likesBirds) {
                System.out.println( “Birds” );
        } else {
                System.out.println( “Cave” );
        }
        System.out.println( “Village” );
}
```

Syntax for `if-else`

if statements have an *optional* else

Formal Syntax:

```
if ( condition ) {
  statement_A1;
  ...
  statement_An;
} else {
  statement_B1;
  ...
  statement_Bn;
}
```

*If the condition is **true**, then statements in block A will be executed*

*If the condition is **false**, then statements in block B will be executed*

“Return” statements allow methods to act like functions

Return

Returning values

- Allows a method to pass information back to the invoking code
- Can only **return** a single value (primitive or object)
- Method is executed then the return value **is used in place of** the method call

Formal Syntax

*private <**returnType**> methodIdentifier(… parameters …) {*

.. code goes here

***return** <value>;*

}

*the value returned must be the **same type** as the one declared in the method header*

```
private String getName() {
        return "Fred Fish";
}
```

Methods that return values

Must return the **same type** as the type declared in the method header

```
private int getAge() {
        return "23";
}
```

Incorrect
types don't match

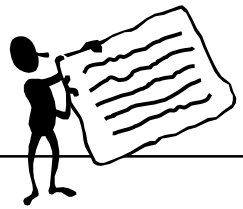
Every method that returns a value must use the keyword "**return**"

- Value returned must be **same type** as declared in signature

```
private int getAge() {
        System.out.println(23);
}
```

Incorrect
does not return value

Example



Calculating the average of two integers

- Pass the integers as parameters to the method
- Calculate the average
- Return the solution

```
private double getAverage(int a, int b) {
   double result = (a + b) / 2.0;
   return result;
}
```

These are two different ways of writing the same method. Any expression of the correct type can follow the keyword "return" (e.g. a variable or a formula)

```
private double getAverage(int a, int b) {
   return (a + b) / 2.0;
}
```

Calling a method that returns a value

Given the following method definition:

```
private double getAverage(int a, int b) {
   return (a + b) / 2.0;
}
```

We would **NOT** *call* the method like this:

```
public void start() {
   getAverage(10, 15);
}
```

12.5

this is equivalent to writing the statement:

```
12.5;
```

which does not make sense

We should ***do something*** with the value that is returned

Calling a method that returns a value

When we call the `getAverage` method, we need to **process** the result in some way. We can process the result in any way that we could process a value of the same type as the return type of the method.

For example, we can assign the result to a variable of the correct type:

```
double result;
result = getAverage(10, 15);
```

12.5

Calling a method that returns a value

Or we could use the result as part of some expression:

```
System.out.println(100 * getAverage(10, 15));
```

12.5

Temperature conversion

The **`convertFahrenheitToCelsius()`** method definition is not **as general** as it could be:

```
public class MyProgram {

        private void convertFahrenheitToCelsius() {
                System.out.print("Enter temperature (F): ");
                int input = Integer.parseInt(Keyboard.readInput());
                int output = (int)((5.0/9)*(input - 32));
                System.out.print("Celsius = " + output + "\n");
        }

        public void start() {
                convertFahrenheitToCelsius();
        }
}
```

- input must come from the keyboard
- output always goes to the screen

Temperature conversion

We can make the **`convertFahrenheitToCelsius()`** method more general by passing input to it via parameters, and receiving output from it as a return value:

```
private int convertFahrenheitToCelsius(int degF) {
        int degC = (int)((5.0/9)*(degF - 32));
        return degC;
}
```

Now, the input value can come from anywhere and the output value can be processed in any way we like.

Checking whether two String variables "contain" the same string of characters.

Comparing Strings

What is the output of the following code?

```
public class MyProgram {

   public void start() {

      String a = "put";
      String b = "computer";

      String c = b.substring(3,6);

      System.out.println(a + " " + c + " " + (a == c));

   }
}
```

put put false

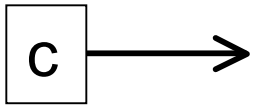
Comparing Strings

Why?

"String" in Java is not a primitive data type, rather it is a class.

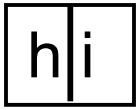
When we create an instance of String, it is an object.
When we declare an object variable, we don't reserve space for that object, rather we reserve space for a pointer.

String c;



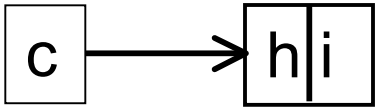
When we create an object, then we actually reserve space for that object.

String.new("hi")



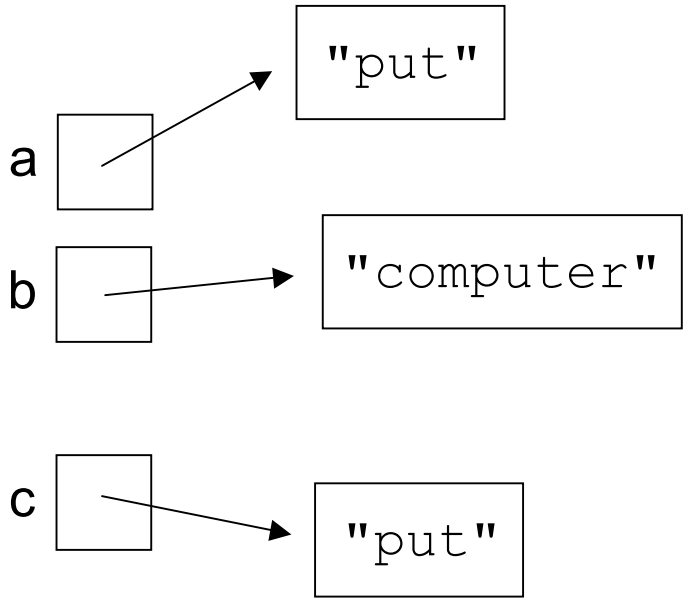
When we assign an object to an object variable, we cause the variable to point to the object.

c = "hi";



Comparing Strings

"*a* == *c*" asks whether *a* points to the same object as *c* points to.

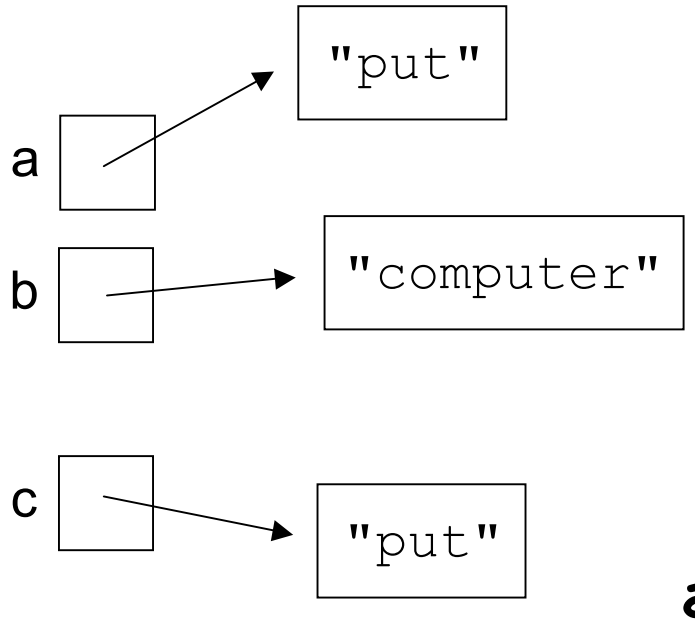



```
String a = "put";
String b = "computer";
String c = b.substring(3,6);
```

`a == c` is false

Comparing Strings

To compare two String variables character by character, we must use the **`.equals()`** instance method



`a == c` **is false**

`a.equals(c)` **is true**

Assignment 3 Preview

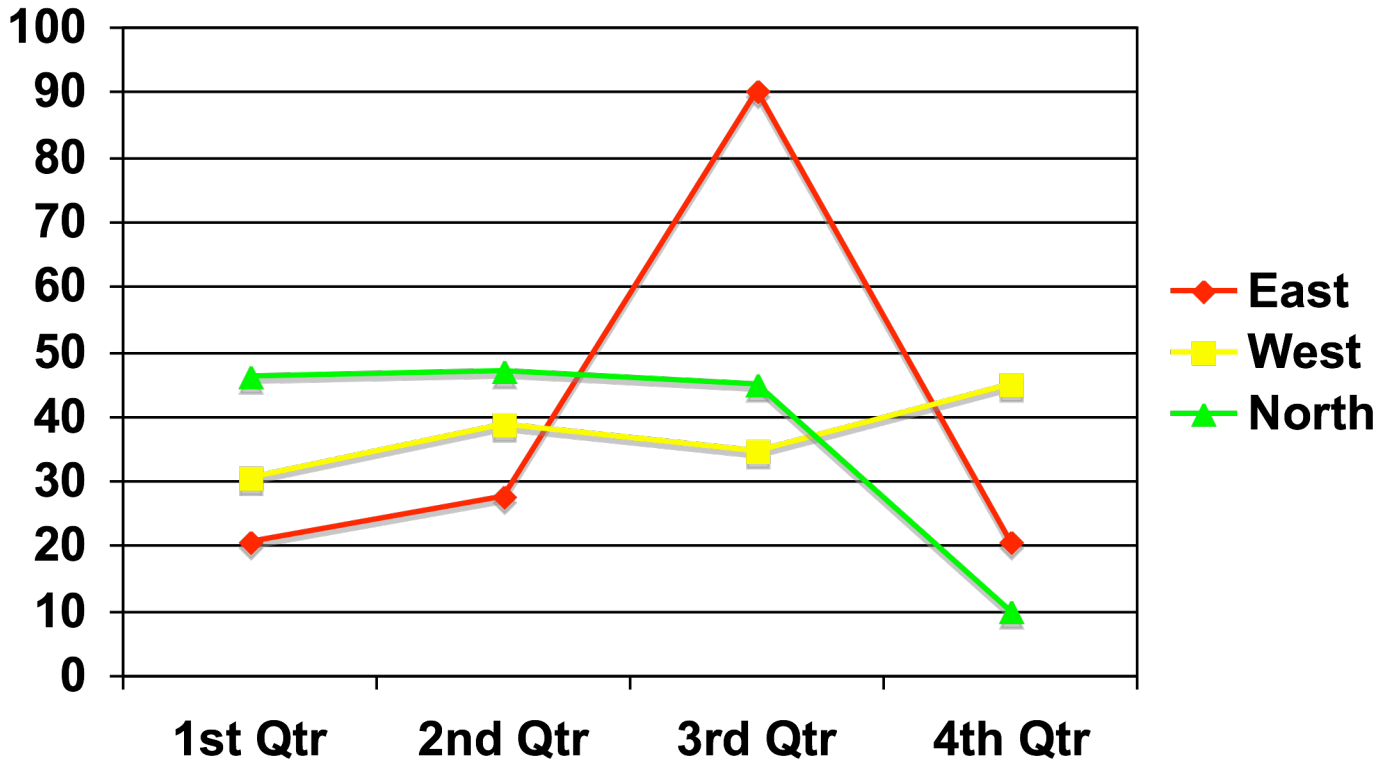
In assignment 3, you will be writing classes for graphs, nodes, and edges. You will be reading in a simplified notation for graphs and writing out a description of the graph in a language called "dot". There are a number of freeware programs that reads in "dot" descriptions of graphs and displays the described graphs.

Graphs are an important mathematical structure (a la graph theory) and an important data structure. You will learn a lot more about graphs and their properties in CS 220.

In CS220 you will learn much more efficient ways of representing graphs than the way you will be asked to represent them in this assignment.

Background to Assignment 3

Graphs you are probably familiar with:

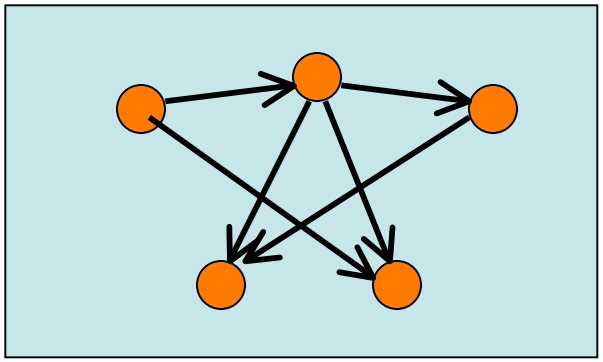


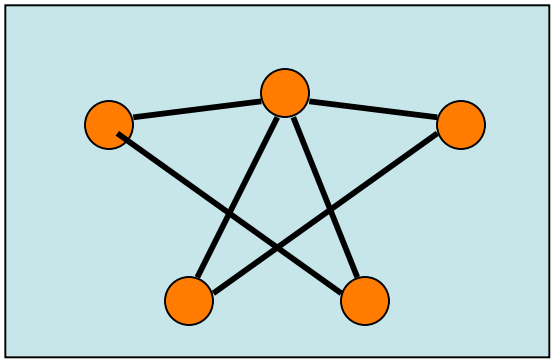
There are data points and lines connecting the points.

Abstract Graphs

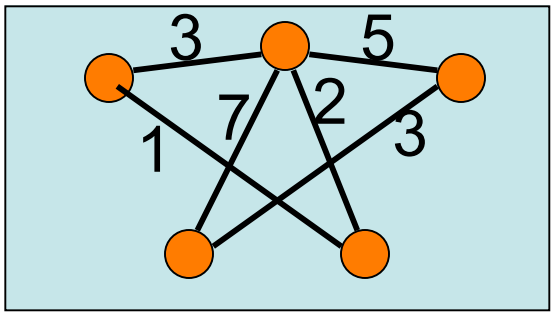
In math and computer science, a graph is a collection of nodes and a collection of edges which connect pairs of those nodes.

Graphs can either be directed or undirected. In directed graphs, the edges have a directionality, in undirected graphs, they don't.





Graphs can either be weighted or unweighted. In weighted graphs, the edges have a weight, in unweighted graphs, they don't.



Examples of Graphs

1. A directed graph can be used to represent prerequisite information for papers. Nodes would represent papers and edges would directed from a paper to its prerequisite papers. One way of reducing the number of edges is to remove edges that are redundant. For example, if paper A requires both papers B and C, and paper B requires paper C, then only the edges from A to B and from B to C are necessary as the information that paper A requires paper C can be inferred from the other two edges (A -> B and B -> C).

2. A weighted undirected graph could be used to represent the "air miles" between airports. The nodes would represent the airports, the edges would represent pairs of airports which have direct flights between them. The weights of the edges would be the air miles between those airports. Note that in this graph, you cannot remove edges as you did in the example above. Just because there are direct flights from A to B and from B to C, that does not mean there are direct flights from A to C.

Examples of Graphs

3. A directed graph could be used to represent method calls. The nodes are methods and edges represent the presence of a method call. The source node represents the calling method, and the target node represents the called method.