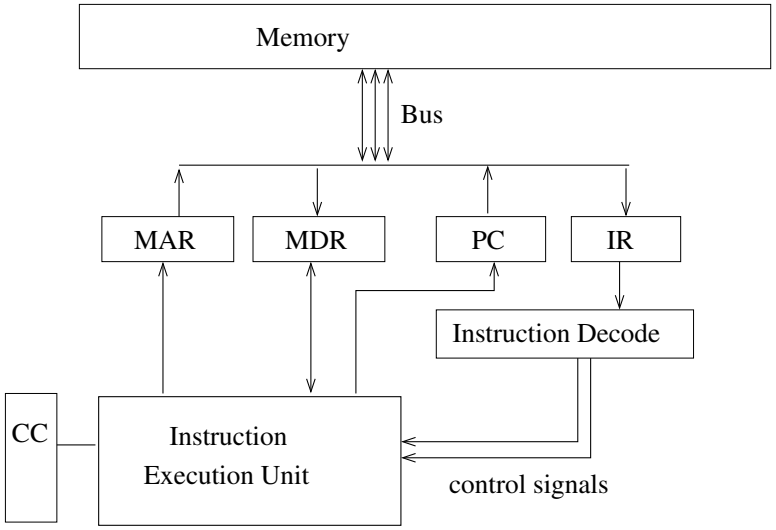
Programming Concepts

Simplified Computer Architecture



MAR Memory Address Register – Address for load/store

MDR Memory Data Register – destination / source for data

PC Program Counter – Address of the next instruction

IR Instruction Register – Holds the current instruction

CC Condition Code Register – Retains information from the `compare` instruction

Fetch-Decode-Execute Cycle

1. Instruction fetch (copy instruction from memory)
2. Instruction decode (create control signals)
3. Operand (data) fetch (copy data from memory)
4. Instruction execution (add, multiply, branch, etc.)
5. Data store (copy data to memory)
6. Update program counter (PC) to next instruction.

Memory and Addresses

An **address** is simply a numbered location in memory, analogous to a numbered parking space in a large single-row parking lot. For most modern computers, the smallest addressable unit is one byte (8 bits). Traditionally, 8 bits is the amount of storage used for one character (letter, punctuation, etc.) in the ASCII representation[1] Internationalized character sets usually use two bytes (16 bits) for each character.

[1] ASCII is an acronym for "American Standard Code for Information Interchange".

Machine Language Program

- A machine language program is a sequence of instructions.
- Consecutive instructions are executed in order, except ...
- a branching instruction changes the program counter to a different point in the machine language program.

Programming Languages

- Assembly Language: Human-readable form of machine language. Examples (SPARC):

```
        mov     %g0, %g1
L0:
        cmp     %g1, 5
        bge     L1
        nop
        add     %g1, 1, %g1
        ba      L0
        nop
L1:
```

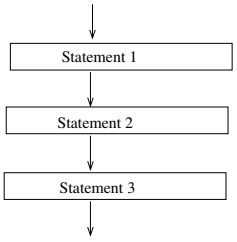
- Compiled Languages: E.g. C, C++, FORTRAN.
  - A program in a compiled language is translated to machine code. The resulting binary code runs directly on the hardware.
- Interpreted Languages: E.g. Python, Perl, Matlab.
  - A program in an interpreted is translated and executed line-by-line. A software layer between the program and the hardware provides a run-time environment for the program.
- Hybrid Language(s): Java
  - A Java program is translated into *byte code*. The *Java Virtual Machine* provides a run-time environment to run the byte code.

Structured Programming

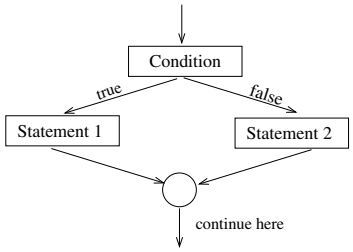
A **goto** statement is a statement in a program that transfers control (i.e., the next statement to be performed) to any chosen place within the program. A **conditional goto** statement might or might not transfer control to a (labeled) destination statement, depending on some condition. In assembly language, the only mechanism for implementing a program's logical flow is a (conditional) goto statement.

It was recognized early on in the development of computer programs that allowing a programmer to use a **goto** statement contributed to a lot of (human) programming errors. Many modern computer languages do not support a **goto** statement. Instead, three types of program structure are supported:

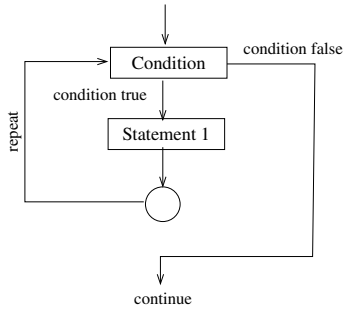
- Sequence



- Selection

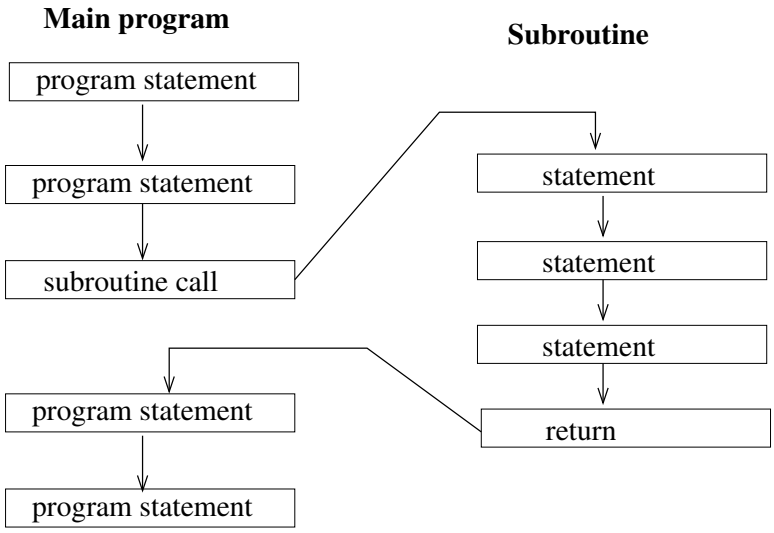


- Iteration



- The *Structure Theorem* proves that any program using (conditional) **goto** statements can be re-written using the three structured forms: sequence, selection, and iteration.
- The three structured forms must be properly nested.

Subroutines In addition to the structures described above, all modern languages support the concept of sub-program units, often called **subroutines**. Each subroutine must have well-defined inputs and outputs. Control is transferred from the main program to the subroutine. We refer to the transfer of control as a **subroutine call**. On completion, the subroutine returns control flow to the point in the caller immediately after the call. This arrangement of control flow is illustrated below:



Later, we will make a distinction between different types of subroutines including **functions**, **procedures**, and **methods**.

A Good Programming Practice: Each subroutine should do one thing and do it well.

Java Concepts

Objects

Java is an object-oriented language. A Java **object** is an organizational unit consisting of both data and methods. You can think of a *method* as a subroutine belonging to an object. Both data and methods may be either *private* or **public**.

Frequently, objects are designed so that the data is private to the object. The only operations allowed on the data are the ones provided by public methods. The combination of private data and public methods provides encapsulation of the data.

Encapsulation

The purpose of encapsulation is to reduce (human) programming errors. If we allow the data to be changed in an ad-hoc way at numerous places throughout the program, it becomes more likely that a (human) programmer will make a mistake in one of those places. Such programming errors can be exceptionally difficult to find and correct. Worse still, a minor design change can require corrections to all of the places where the data is updated. If the places where the data is updated are scattered throughout thousands (or millions) of lines of code, the process of making corrections begins to exceed the human capacity to keep track of it all. A much better approach is to limit the number of places where the data is changed to a small number of methods.

Classes

A **Java class** can be thought of as a pattern from which objects are made. A class declaration defines what data members and methods belong in the class.

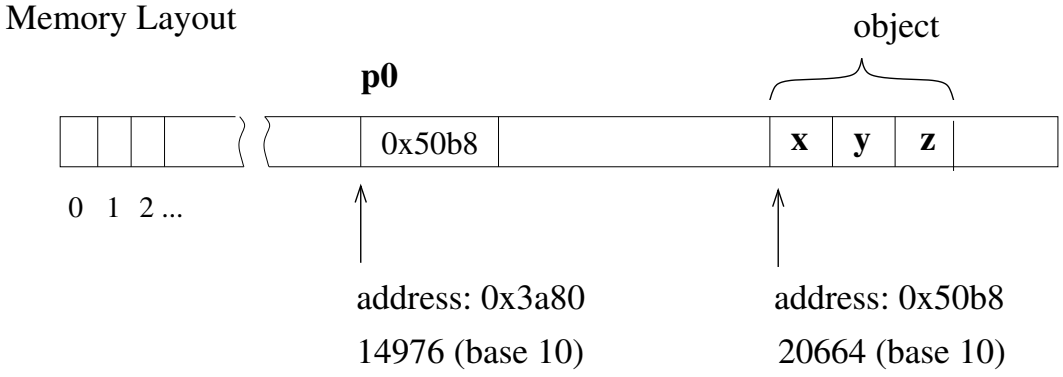
References (a.k.a. pointers)

A **reference** (to an object) consists of two things:

- a memory address where the object is stored, and
- the data type (class) of the object.

One of the most frequent sources of program errors is the mis-use of references. To avoid such errors, we need to have a clear understanding of the layout of memory. We illustrate memory layout with an example.

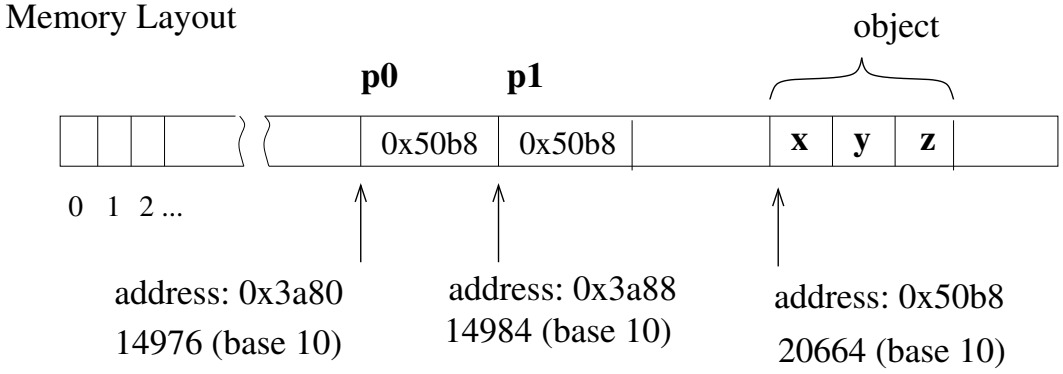
Suppose we wish to create an object named `p0` to represent a 3-D point. The object contains three (floating point) data members named `x`, `y`, and `z`. All objects in Java are represented by references. The following diagram illustrates an example memory layout for our object.



So what?: The fact that objects are represented by references, has important implications whenever an object is copied. Suppose we have created object `p0` as shown above, and we consider the statement copying `p0` to `p1`:

```
p1 = p0 ;
```

This results in the following memory layout:



The situation illustrated above is known as a **shallow copy**. It is easy to forget that `p0` and `p1` refer to the same memory location. If we change the data member `x` belonging to object `p1` it has the (perhaps unexpected) effect of changing the `x` in object `p0`.

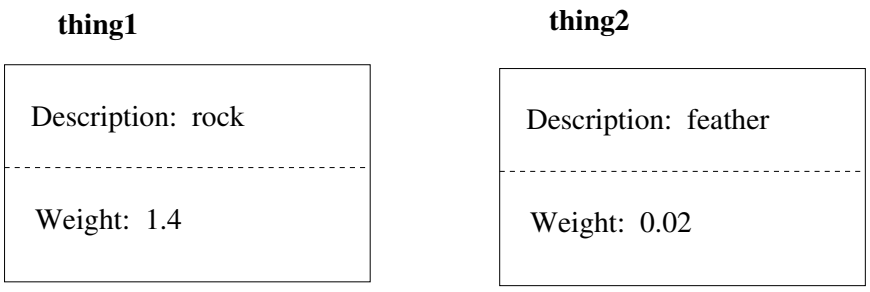
The alternative to a shallow copy is a **deep copy**. To make a deep copy, we need to allocate a new block of memory and copy the data members `x`, `y`, and `z` to the new memory block. The usual approach is to include a `deep_copy` method in the class to make the needed copies.

Note: When an object is passed to a method, it is passed via a **shallow copy**.

Static v.s. Non-static Data and Methods

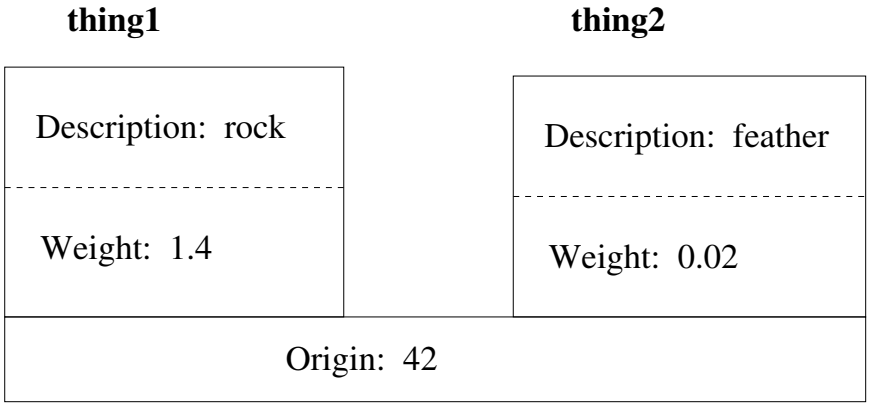
Most of the time, it is in our interests to use non-static data members in a class. For a non-static data member, there is an instance of that data member for every object of that class.

Suppose we have a class named `shipping_box` containing two non-static data members: `Description` and `Weight`. Suppose also we have two objects named `thing1` and `thing2`. The objects with non-static data members are illustrated in the following diagram:



As a second example, let us consider a class with a static data member. There is only **one instance** of a static data member for every object of that class.

Suppose we are in the business of shipping boxes; we have a retail store that is part of a franchise. Suppose we are store number 42. All boxes shipped out of our store will have the same store number as the shipping origin. A natural way to implement this is to make `Origin` a static data member. All boxes (that we handle) will share the same store number. This situation is illustrated in the following diagram:



Notice that the two objects contain distinct non-static data: `Description` and `Weight`. However the objects share the data `Origin`.

Note: A consequence of static v.s. non-static storage is that a static method can not refer to non-static data.

Example Java Program

```
//
//  File:  demo.java
//
import java.util.Scanner ;

class point {

   //  Data members
   private float x ;
   private float y ;

   //  Methods
   public void read_point() {
      Scanner sam ;
      sam = new Scanner(System.in) ; // Dynamically allocate a scanner object.
      x = sam.nextFloat() ;
      y = sam.nextFloat() ;
   }

   public void mid_point( point p1, point p2 ) {
     x = 0.5f * ( p1.x + p2.x ) ;
     y = 0.5f * ( p1.y + p2.y ) ;
   }

   public void print_point( String label) {
       System.out.println( label + ": [ " + x + ", " + y + "]" ) ;
   }

}  // end class point.

class demo{
   public static void main( String [] args )
   {
      point p1, p2, p3 ; //   Declare the variables.
      p1 = new point() ;  // Create a point object.
      p2 = new point() ;  // Create a point object.
      p3 = new point() ;  // Create a point object.

      p1.read_point() ;
      p2.read_point() ;
      p3.mid_point( p1, p2 ) ;
      p1.print_point("p1") ;
      p2.print_point("p2") ;
      p3.print_point("midpoint" ) ;
   }

}  // end class demo.
```

- - - - - - - - - - Sample Session - - - - - - - - - -

```
hawk%
hawk% javac demo.java
hawk% java demo
3 5
11 17
p1: [ 3.0, 5.0]
p2: [ 11.0, 17.0]
midpoint: [ 7.0, 11.0]
```