ITI 1121. Introduction to Computing II[†]

Marcel Turcotte
(with contributions from R. Holte)

School of Electrical Engineering and Computer Science
University of Ottawa

Version of January 19, 2015

[†]Please don't print these lecture notes unless you really need to!

Review

Objectives:

1. Knowing the expectations regarding Java
2. Introducing basic concepts of computer architecture and program execution

Lectures:

▶ Pages 597–631 of E. Koffman and P. Wolfgang.

Prerequisite

Familiarity with the following concepts is assumed:

- ▶ Using Java's pre-defined data types:
  including arrays and Strings;

Prerequisite

Familiarity with the following concepts is assumed:

▶ Using Java's pre-defined data types:
  including arrays and Strings;

▶ Control structures:
  such as if, for, while...;

Prerequisite

Familiarity with the following concepts is assumed:

- ▶ Using Java's pre-defined data types:
  including arrays and Strings;

- ▶ Control structures:
  such as if, for, while...;

- ▶ Procedural abstractions (structured programming):
  i.e. how to define and use (static) methods;

Prerequisite

Familiarity with the following concepts is assumed:

- ▶ Using Java's pre-defined data types:
  including arrays and Strings;
- ▶ Control structures:
  such as if, for, while...;
- ▶ Procedural abstractions (structured programming):
  i.e. how to define and use (static) methods;
- ▶ How to edit, compile and run a Java program.

Why Java?

Why Java?

1	C	17%
2	Java	16%
3	Objective-C	7%
4	C++	7%
5	C#	5%
6	PHP	4%
7	JavaScript	3%
8	Python	3%
9	Perl	2%
10	PL/SQL	2%

$\Rightarrow$ TIOBE Programming Community Index

Why Java?

Java shares the first rank in popularity with C, but where is Java
used? I don't seem to know any applications built using Java.

Why Java?

Java shares the first rank in popularity with C, but where is Java used? I don't seem to know any applications built using Java.

- ▶ Server-side Web applications and services
- ▶ Mobile (phones) applications

Why Java?

" According to a report from NetApplications, which has measured browser usage data since 2004, Oracle's Java Mobile Edition has surpassed Android as the #2 mobile OS on the internet at 26.80%, with iOS at 46.57% and Android at 13.44%. And the trend appears to be growing. Java ME powers hundreds of millions of low-end 'feature phones' for budget buyers. In 2011, feature phones made up 60% of the install base in the U.S. "

Slashdot
January 3, 2012
http://bit.ly/xSk5pN

Why Java?

- C requires discipline
  (memory management, pointers. . .)

Why Java?

- ▶ C requires discipline
  (memory management, pointers. . .)
- ▶ Java is good vehicle for teaching
  (interface, single inheritance. . .)

Why Java?

- ▶ C requires discipline
  (memory management, pointers. . .)
- ▶ Java is good vehicle for teaching
  (interface, single inheritance. . .)
- ▶ Once you know Java, learning other
  imperative/object-oriented programming languages is easy

Why Java?

"If you look at job requirements across the world, the demand has skyrocketed for Java (holds number 1 place), Objective-C and Swift now, C#."

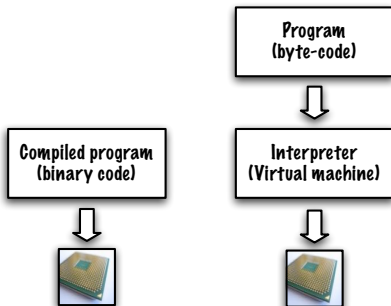**What Is The Most Valuable Programming Language To Know For The Future And Why?**
www.forbes.com/sites/quora/2014/07/14/what-is-the-most-valuable-programming-language-to-know-for-the-future-and-why

Program execution

What are the two main modes of execution?

Program execution

What are the two main modes of execution?

```
                                   ┌──────────────┐
                                   │   Program    │
                                   │  (byte-code) │
                                   └──────────────┘
                                          ⇩
    ┌──────────────────┐          ┌──────────────────┐
    │ Compiled program │          │   Interpreter    │
    │  (binary code)   │          │ (Virtual machine)│
    └──────────────────┘          └──────────────────┘
             ⇩                             ⇩
```

Compiling and executing a Java program

> javac MyProgram.java

Produces MyProgram.class (the byte-code)

Compiling and executing a Java program

> javac MyProgram.java

Produces MyProgram.class (the byte-code)

> java MyProgram

Here, **java** the Java Virtual Machine (JVM).

Compiling and executing a Java program

> javac MyProgram.java

Produces MyProgram.class (the byte-code)

> java MyProgram

Here, **java** the Java Virtual Machine (JVM).
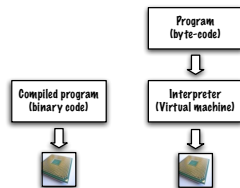
Motivation

▶ Under the old academic program, ITI 1121 (CSI 1101) used to
  have a section on computer architecture: with topics such as
  Boolean algebra, switching logic, number representation,
  assembly programming, program compilation and
  interpretation.

Motivation

► Under the old academic program, ITI 1121 (CSI 1101) used to have a section on computer architecture: with topics such as Boolean algebra, switching logic, number representation, assembly programming, program compilation and interpretation.

► **Today's lecture presents a simplified model of the execution of computer programs at the hardware level.**

Motivation

- ▶ Under the old academic program, ITI 1121 (CSI 1101) used to have a section on computer architecture: with topics such as Boolean algebra, switching logic, number representation, assembly programming, program compilation and interpretation.

- ▶ **Today's lecture presents a simplified model of the execution of computer programs at the hardware level.**

- ▶ This helps understanding the distinction between primitive and reference types, the execution of computer programs in general.

TC1101

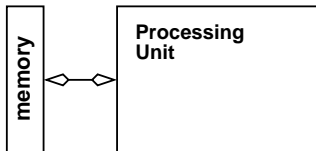A simplified microprocessor and its assembly language.
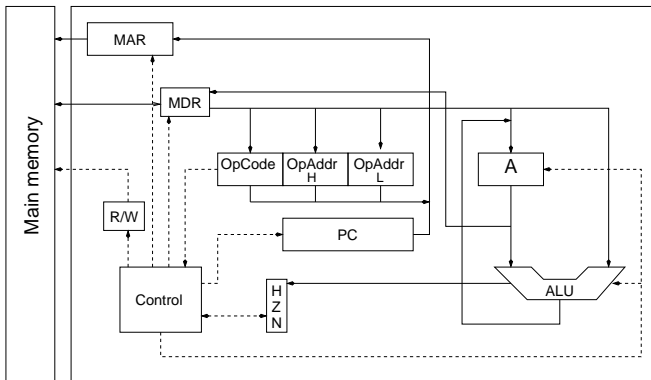
von Neumann model

The design of modern computers is based on a model proposed by
John von Neumann in 1945.

Memory: contains the instructions and the data;

Processing unit: performs arithmetic and logic operations;

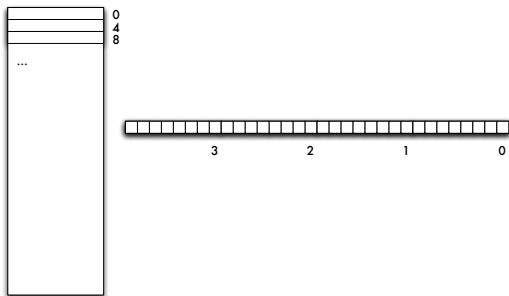Control unit: interprets the instructions.

Memory model

▶ Can be seen as a large array, where each cell holds one bit of
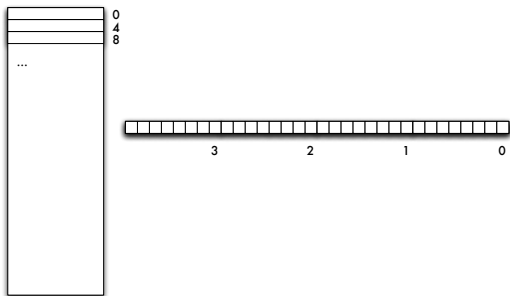 information (*binary digit*), 0 or 1;

Memory model

- ▶ Each byte has a unique/distinct address
- ▶ Bytes are grouped together to form words
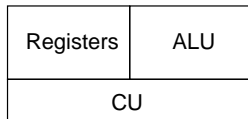- ▶ Some data types require using more than one byte

Memory

- ▶ This type of memory is called RAM
  (*Random Access Memory*)
- ▶ **The access time is the same (is constant) for all cells,**
  Typically, 5 to 70 nano seconds (nano $= 10^{-9}$)

Central Processing Unit (CPU), processor or μ-processor

▶ Executes one instruction at a time (in the case of sequential computers — not parallel ones)

Registers	ALU
CU	

ALU *Arithmetics/Logic Unit*, contains the necessary circuits to execute all the instructions supported by the hardware, e.g. addition

CU *Control Unit*, transfers the instructions from memory and determine their type

Registers are units inside the processor that serve to store data

Mnemonic, opCode and description

LDA	91	load x
STA	39	store x
CLA	08	clear (a=0, z=true, n=false)
INC	10	increment accumulator (modifies z and n)
ADD	99	add x to the accumulator (modifies z and n)
SUB	61	subtract x to the accumulator (modifies z and n)
JMP	15	unconditional branch to x
JZ	17	go to x if z==true
JN	19	go to x if n==true
DSP	01	display the content of the memory location x
HLT	64	halt

Compilation

Programs, statements in a high level programming language, are translated (**compiled**), into a lower level representation (assembly, **machine code**), that can be directly interpreted by the hardware.

Compilation

Programs, statements in a high level programming language, are
translated (**compiled**), into a lower level representation (assembly,
machine code), that can be directly interpreted by the hardware.
The expression $y = x + 1$ is translated to assembly code:

LDA X
INC
STA Y
HLT

which is then translated to machine code:
| 91 00 08 | | 10 | | 39 00 09 | | 64 | | 10 | | 99 |

Division, successive subtractions: assembly

```
[1]    CLA
       STA Quot
[2]    LDA X
[3]    SUB Y
[4]    JN  [7]
[5]    STA Temp
       LDA Quot
       INC
       STA Quot
       LDA Temp
[6]    JMP [3]
[7]    ADD Y
[8]    STA Rem
[9]    DSP Quot
[10]   DSP Rem
[11]   HLT
X      BYTE 25
Y      BYTE 07
Quot   BYTE 00
Rem    BYTE 00
Temp   BYTE 00
```

Division: machine code

```
[1]    CLA                      08
       STA Quot                 39      00 44
[2]    LDA X                    91      00 42
[3]    SUB Y                    61      00 43
[4]    JN  [7]                  19      00 29
[5]    STA Temp                 39      00 46
       LDA Quot                 91      00 44
       INC                      10
       STA Quot                 39      00 44
       LDA Temp                 91      00 46
[6]    JMP [3]          ⟹      15      00 07
[7]    ADD Y                    99      00 43
[8]    STA Rem                  39      00 45
[9]    DSP Quot                 01      00 44
[10]   DSP Rem                  01      00 45
[11]   HLT                      64
X      BYTE 25                  25
Y      BYTE 07                  07
Quot   BYTE 00                  00
Rem    BYTE 00                  00
Temp   BYTE 00                  00
```

Division: machine code

08 39 00 44 91 00 42 61 00 43 19 00 29 39 00 46 91 00 44 10 39
00 44 91 00 46 15 00 07 99 00 43 39 00 45 01 00 44 01 00 45 64
25 07 00 00 00

Registers

▶ Memory units that reside inside the processor, access time is
   very fast

Registers

- ▶ Memory units that reside inside the processor, access time is very fast
- ▶ Registers are not identified by address but by name (MAR, MDR, A, etc.)

## Registers

- ▶ Memory units that reside inside the processor, access time is very fast
- ▶ Registers are not identified by address but by name (MAR, MDR, A, etc.)
- ▶ While central memory is general, registers are specific, **each of them has a distinct function/role**

Registers

- ▶ Memory units that reside inside the processor, access time is very fast
- ▶ Registers are not identified by address but by name (MAR, MDR, A, etc.)
- ▶ While central memory is general, registers are specific, **each of them has a distinct function/role**
- ▶ While the size of the elements of the central memory is the same, **the size of the registers vary according to their function**
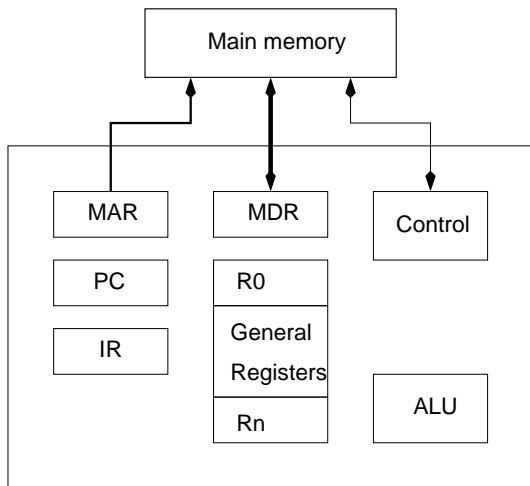
Interface between the memory and the CPU

- ▶ Bits are not transferred one at a time, but in parallel
- ▶ We call **bus** the set of wires (lines) that enable communications between units
- ▶ **There are 3 types of buses: data, address and control**
- ▶ The number of lines (wires) determines the width of the bus
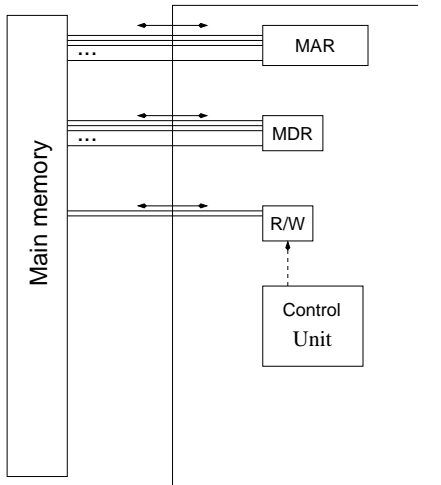
Address bus

- ▶ The width of the bus determines the maximum size of the memory
- ▶ If the width of the bus is 16 lines, addresses are made of 16 bits, there are therefore $2^{16} = 65,536$ distinct addresses, for 32 lines, addresses are made of 32 bits, there are therefore $2^{32} \simeq 4 \times 10^9$ distinct addresses
- ▶ The memory register will also have 32 bits to store an address

Data bus

The width of the data bus determines the number of bits
transfered in one access (to/from memory).

Control bus (lines)

In our simplified model, indicates the direction of a transfer

R (read) transfer should be made from the memory to the
processor

W (write) transfer should be made from the processor to the
memory

Transfer from the memory

In order to transfer a value from the memory location/address x to the processor,

1. put the value x into the memory address register
2. set the status bit RW to true
3. activate the control line "access_memory"
4. the memory data register (MDR) now contains a **copy** of the value found at the address x of the memory

Transfer to the memory

In order to transfer a value v from the processor to the address location x of the memory:

1. put v into the memory data register (MDR);
2. put x into the memory address register (MAR);
3. set the status bit RW to false;
4. activate the control line "access_memory".

"FETCH-EXCECUTE" Cycle

1. fetch:
    1.1 transfer the opcode,
    1.2 increment PC,
2. depending of the opcode transfer the operand:
    2.1 transfer the first byte,
    2.2 increment PC,
    2.3 transfer the second byte,
    2.4 increment PC
3. execute.

Example

In order to add 1 to the value x and save the result to y
($y = x + 1$)

- ▶ Load the value x into the accumulator, register A
- ▶ Increment the value of the accumulator
- ▶ Save the value of the accumulator at the address y

Example

In order to add 1 to the value x and save the result to y
($y = x + 1$)

- ▶ Load the value x into the accumulator, register A
- ▶ Increment the value of the accumulator
- ▶ Save the value of the accumulator at the address y

Which requires three machine instructions:

91: load

10: increment

39: store

Example

In order to add 1 to the value x and save the result to y
($y = x + 1$)

- Load the value x into the accumulator, register A
- Increment the value of the accumulator
- Save the value of the accumulator at the address y

Which requires three machine instructions:

> 91: load
>
> 10: increment
>
> 39: store

If x designates the address 00 08 and y designates the address 00 09, then $y = x + 1$ can be written in machine language as follows:

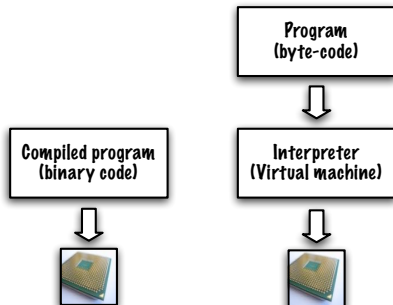| 91 00 08 | | 10 | | 39 00 09 | | 64 | | 10 | | 99 |

TC1101 Java Simulator

These slides provide complementary information on the TC1101
microprocessor and its assembly language.
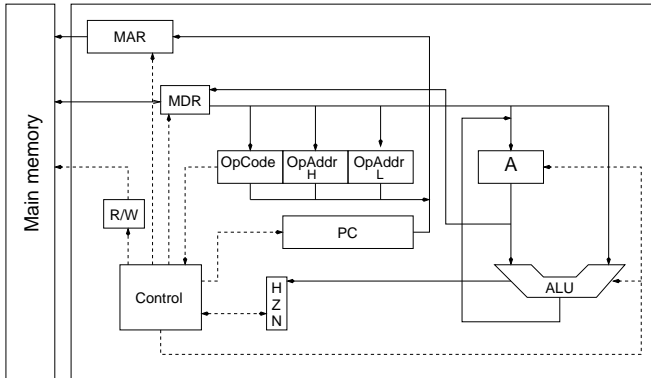
TC1101 Java Simulator

Our simulator plays the same role as the Java Virtual Machine
(the interpreter on the right hand side)
and shares many characteristics.

Mnemonic, opCode and description

LDA	91	load x
STA	39	store x
CLA	08	clear (a=0, z=true, n=false)
INC	10	increment accumulator (modifies z and n)
ADD	99	add x to the accumulator (modifies z and n)
SUB	61	subtract x to the accumulator (modifies z and n)
JMP	15	unconditional branch to x
JZ	17	go to x if z==true
JN	19	go to x if n==true
DSP	01	display the content of the memory location x
HLT	64	halt

Functional Units of the TC-1101

PC (2 bytes): *Program Counter*, one 2 bytes register that contains
the address of the next instruction to be executed;

opCode (byte): instruction register (sometimes called IR), contains
the OPCODE of the current instruction;

opAddr (2 bytes): the operand of the current instruction. The
operand is always an address. Some instructions
necessitate the value found at the address designated
by the operand — this value is not transfered by the
basic cycle, but needs to be transfered during the
execution of the instruction (see step 3 of the cycle
and the description of each instruction below);

Functional Units of the TC-1101

MDR (byte): *Memory Data Register*. A value transfered
(read/written) from the memory to the processor (or
vice-versa) is always stored in this registered;

MAR (2 bytes) : *Memory Address Register*. This register contains
the memory address of a value to be read or to be
written;

A (byte): Accumulator. All the arithmetic operations use this
register as an operand and also to store their result;

Functional Units of the TC-1101

H (bit): status bit "Halt". This bit is set by the instruction halt (hlt). If the bit is true the processor stops at the end of this cycle;

N (bit): status bit "Negative". Arithmetic operations set this bit to true whenever they produce a negative result. Some operations are not affecting the value of this bit, therefore its value does not always reflect the content of the accumulator;

Z (bit): status bit "Zero". Arithmetic operations set the value of this bit to true whenever the result is zero. Some operations do not affect the content of this bit, therefore, its value does not always reflect the content of the accumulator;

Functional Units of the TC-1101

RW (bit): status bit "READ/WRITE". A value true means a
value must be read (fetched) from the memory and
transfered to MDR. A value false signifies that a
value must be transfered from MDR to the memory.

TC1101 Simulator

- ▶ Uses constants to represent opcodes
- ▶ Class variables represent memory and registers
- ▶ The class method **accessMemory()** simulates the transfer of data in between the processor and memory
- ▶ The class method **run()** simulates the "FETCH-EXECUTE" cycle: read opCode, transfer operand and execute the current instruction

Constants to represent the opCodes

```java
import java.io.*;
import SimIO; // read data from a file
class Sim {
    // addresses are 2 bytes
    public static final int  MAX_ADDRESS = 9999;

    // load from memory to the accumulator
    public static final int  LDA = 91;
    // save accumulator to memory
    public static final int  STA = 39;
    // set accumulator to 0
    public static final int  CLA =  8;
    // increment accumulator by 1
    public static final int  INC = 10;
    // add to the accumulator
    public static final int  ADD = 99;
    // subtract from the accumulator
    public static final int  SUB = 61;
    // unconditional branch "go to"
    public static final int  JMP = 15;
    // branch to address if Z
    public static final int  JZ  = 17;
    // branch to address if N
    public static final int  JN  = 19;
    // display to screen
    public static final int  DSP =  1;
    // "halt"
    public static final int  HLT = 64;

    // ...
```

Modeling the state of the memory and registers

```java
private static final int[] memory = new int[MAX_ADDRESS + 1];

// program counter
private static int pc;

// accumulator
private static int a;

// opcode of the current instruction
private static int opCode;

// address of the operand
private static int opAddr;

// status bit "Zero"
private static boolean z;

// status bit "Negative"
private static boolean n;

// status bit "Halt"
private static boolean h;

// memory address register
private static int mar;

// memory data register
private static int mdr;

// bit Read/Write.  Read = True; Write = False
private static boolean rw;
```

Loading a machine code program

```java
public static void load(String filename)
throws IOException {
    int[] values;
    int i;
    int address = 0;
    SimIO.setInputFile(filename);
    while (!SimIO.eof()) {
        values = SimIO.readCommentedIntegerLine();
        for (i = 0; i < values.length; i++) {
            memory[address] = values[i];
            address = address + 1;
        }
    }
}
```

Memory access

```java
// the method simulates the effect of activating the access
// control line.

private static void accessMemory() {

    if (rw) {

        // rw=True signifies "read"
        // copy the value from memory to processor

        mdr = memory[mar];

    } else {

        // rw=False signifies "write"
        // copy a value from the processor to the memory

        memory[mar] = mdr;

    }
}

// ...
```

FETCH-EXECUTE

```java
// ''FETCH-EXECUTE'' cycle simulation starts
// at the address 00 00

public static void run() {

pc = 0;        // always starts at zero
h  = false;    // re-initialize the status bit halt

while (h == false) {

    // load opCode

    mar = pc;
    pc  = pc + 1;     // pc is incremented
    rw  = true;
    accessMemory ();
    opCode = mdr;
```

```
// if the opCode is odd, this instruction
// necessitates an operand

if ((opCode % 2) == 1) {
    mar = pc;
    pc  = pc + 1;     // increment pc
    rw  = true;
    accessMemory ();  // reading the high part of
    opAddr = mdr;     // this address
    mar = pc;
    pc  = pc + 1;     // increment pc
    rw  = true;
    accessMemory ();  // read low part of this address
    opAddr = 100 * opAddr + mdr;  // put high+low together
}
```

```
// execute the instruction

switch (opCode) {

case LDA: {
    mar = opAddr;        // read value designated by operand
    rw  = true;
    accessMemory ();
    a   = mdr;           // put this value into accumulator
    break;
}
```

```
case STA: {
    mdr = a;            // put content of the accumulator
    mar = opAddr;       // at address designated by opAddr
    rw  = false;
    accessMemory ();
    break ;
}

case CLA: {
    a   = 0;            // clear = set accumulator to zero
    z   = true;         // also sets status bit Z and N
    n   = false;
    break ;
}
```

FETCH-EXECUTE (contd)

```java
case INC: {
    a   = (a + 1) % 100; // increment = add 1 to accumulator
    z   = (a == 0);      // affect the status bits
    n   = (a < 0);
    break ;
}

case ADD: {
    mar = opAddr ;               // read value designated by operand
    rw  = true ;
    accessMemory ();
    a   = (a + mdr) % 100; // add this value to accumulator
    z   = (a == 0);        // update the status bits
    n   = (a < 0);
    break ;
}
```

```
case SUB: {
    mar = opAddr;              // read value designated by operand
    rw  = true;
    accessMemory();
    a   = (a - mdr) % 100;     // subtract from the accumulator
    z   = (a == 0);            // update the status bits
    n   = (a < 0);
    break;
}

case JMP: {
    pc = opAddr;               // the operand contains the address
    break;                     // of next instruction to be executed
}
```

```
case JZ : {
    if (z) {                    // branch if Z
        pc = opAddr;
    }
    break;
}

case JN : {                     // branch if N
    if (n) {
        pc = opAddr;
    }
    break;
}

case HLT: {
    h = true;                   // sets H to true
    break;
}
```

```java
        case DSP: {
            mar = opAddr;                // read value designated by operand
            rw  = true;
            accessMemory();

            // in order to produce a clean output add zeros to the left
            // if necessary

            String smar = "" + mar ;
            while (smar.length() < 4) {
                smar = "0" + smar;
            }

            String smdr = "" + mdr ;
            if (mdr < 10) {
                smdr = " " + smdr ;
            }

            System.out.println("memory location " + smar +
                               " contains " + smdr);

            break;
        }
    default: System.out.println ("Error - unknown opCode: " + opCode) ;
        }
    }
  }
}
```

Summary

- ▶ You should be familiar with primitive and reference types

Summary

- ▶ You should be familiar with primitive and reference types
- ▶ You should be familiar with control structures

Summary

- ► You should be familiar with primitive and reference types
- ► You should be familiar with control structures
- ► You should be able to compile, run and execute a Java program

Summary

- ▶ You should be familiar with primitive and reference types
- ▶ You should be familiar with control structures
- ▶ You should be able to compile, run and execute a Java program
- ▶ I presented an overview of a simplified computer architecture

Summary

- ▶ You should be familiar with primitive and reference types
- ▶ You should be familiar with control structures
- ▶ You should be able to compile, run and execute a Java program
- ▶ I presented an overview of a simplified computer architecture
- ▶ I simulated the execution of a simple program

Summary

- ▶ You should be familiar with primitive and reference types
- ▶ You should be familiar with control structures
- ▶ You should be able to compile, run and execute a Java program
- ▶ I presented an overview of a simplified computer architecture
- ▶ I simulated the execution of a simple program
- ▶ You should understand the concept of a variable in the context of this simplified model

Next lecture

► Primitive vs reference types

Next lecture

- ▶ Primitive vs reference types
- ▶ Call by value

Next lecture

- ▶ Primitive vs reference types
- ▶ Call by value
- ▶ Scope

References I

📄 E. B. Koffman and Wolfgang P. A. T.
Data Structures: Abstraction and Design Using Java.
John Wiley & Sons, 2e edition, 2010.

📄 P. Sestoft.
Java Precisely.
The MIT Press, second edition edition, August 2005.

📄 D. J. Barnes and M. Kölling.
Objects First with Java: A Practical Introduction Using BlueJ.
Prentice Hall, 4e edition, 2009.

Please don't print these lecture notes unless you really need to!