alex-levesque.com

CMPE 212 — Introduction to Computer Science II

WINTER 2026

Based on lectures by S. Mohammad – Queen's University

Notes written by Alex Lévesque

Contents

1 Preface

Grading Scheme:

Textbook:

Comments:

-

2 Assembly Guide

In Nios II, every instruction is 32 bits wide. Since memory addresses are assigned to individual bytes (8 bits), we know that each instruction occupies a 4-byte block. Therefore, in a routine, the starting address of the next instruction must be exactly 4 bytes higher than the previous one.

In register transfer notation, use [. . .] to denote contents of a location. Use ← to denote transfer to a destination. Example: $R2 \leftarrow [LOC]$. We can also do $R4 \leftarrow [R2] + [R3]$

2.1 Data Movement & Memory Access

These instructions are used to move data between registers or between registers and memory

`mov` (move register): `mov dest., src.`

- Definition: copies the contents of one register into another register
- Purpose: Used when you need to duplicate a value that is currently in a register to use it elsewhere without modifying the original

`movi` (move immediate): `mov dest., IMM16`

- Definition: Loads a 16-bit constant (immediate) value into a register. The value is sign-extended to 32 bits.
- Purpose: Used to initialize a register with a small number (like 0, 1, or small offsets)

`movia` (move immediate address): `mov dest., label/32-bit address`

- Definition: A macro (pseudo-instruction) that loads a full 32-bit value (typically a memory address) into a register
- Purpose: Since standard instructions can only handle 16-bit numbers at a time, movia is essential for loading the addresses of labels so the program knows where data is located in memory

`ldw` (load word): `ldw dest., byte_offset(base address)`

- Definition: Reads a 32-bit word from a specific memory address and loads it into a destination register
- Purpose: Used to retrieve data stored in RAM so the processor can operate on it

`stw` (store word): `stw source, byte_offset(base address`

- Definition: Writes the 32-bit value currently in a register to a specific memory address
- Purpose: Used to save the results of calculations back into RAM for later use

Byte offsets are used to provide a flexible and efficient way to access specific memory locations relative to a known starting point

2.2 Arithmetic Operations

These instructions perform mathematical calculations.

`add` (add): `add dest., src., src.`

- Definition: adds the contents of two source registers together and stores the result in a destination register
- Purpose: Used for standard addition of variables

`addi` (add immediate): `addi dest., src., IMM16`

- Definition: adds the value of a source and a 16-bit constant, storing the result in the destination
- Purpose: commonly used to increment counters or move points to the next item in a list

`subi` (subtract immediate): `subi dest., src., IMM16`

- Definition: Subtracts a 16-bit constant value from source and stores the result in the destination
- Purpose: used to decrement counters or adjust values downwards by a fixed amount

2.3 Control Flow (Branching)

These instructions change the order in which the code executes (loops and if-statements). Branch instruction causes repetition of body. Many branches are conditional, as seen below.

`br` (branch): `br LABEL`

- Definition: unconditionally jumps to the instruction located at LABEL
- Purpose: used to force a loop to repeat or to skip over a section of code entirely

`beq` (branch if equal): `beq src., src., LABEL`

- Definition: compares registers; if they are equal, the program jumps to LABEL
- Purpose: often used to check loop termination conditions

`bne` (branch if not equal): `bne src., src., LABEL`

`bge`, `ble`, `bgt`, `blt` (branch if $\geq, \leq, >, <$): `opcode src., src., LABEL`

- Definition: compares registers; if they are $\geq, \leq, >, <$, the program jumps to LABEL
- Purpose: often used to check loop termination conditions

2.4 Subroutines (Functions)

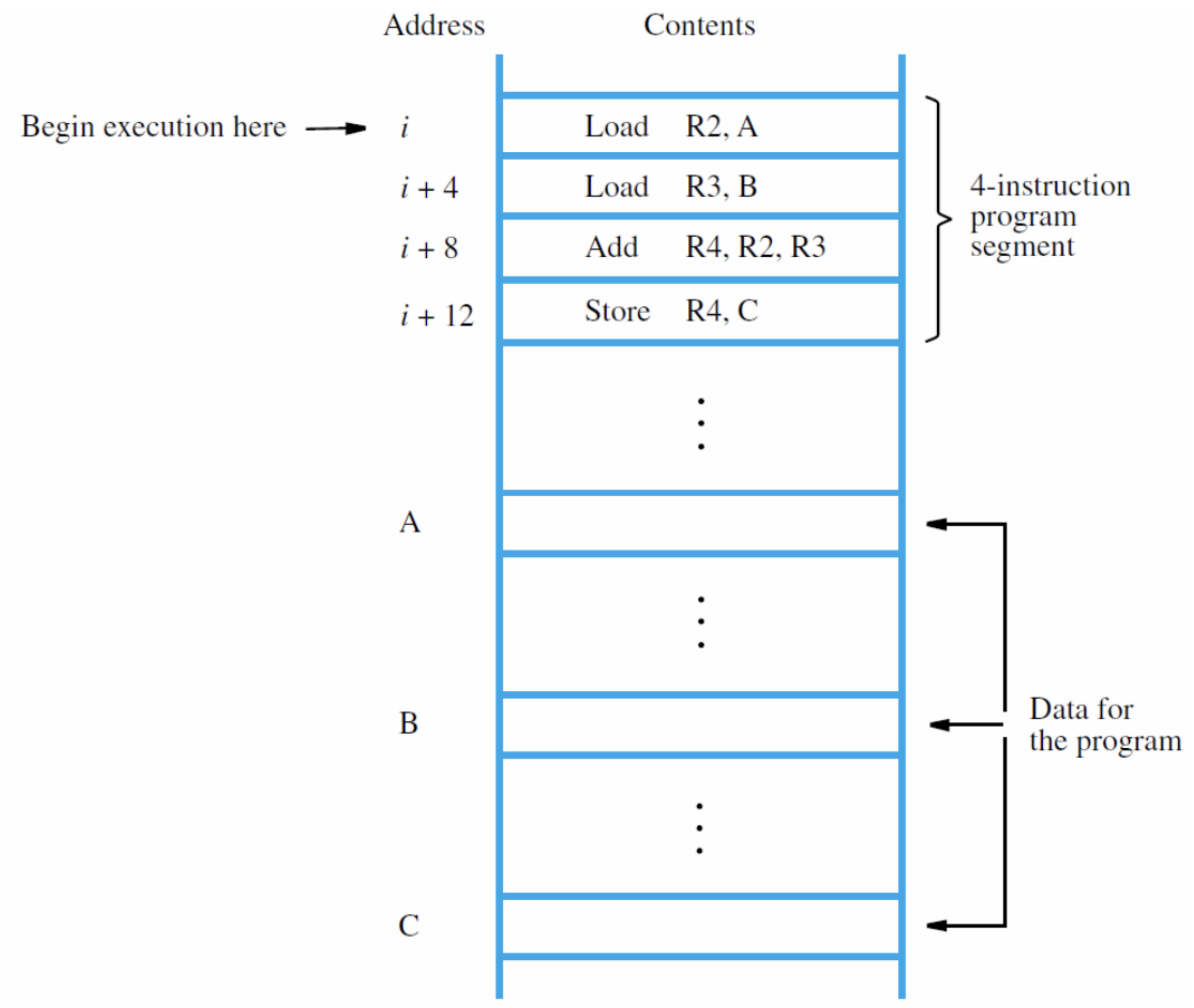
These instructions are used to call and return from functions

`call` (call subroutine): `call LABEL`

- Definition: jumps to LABEL and automatically saves the address of the next instruction into the return address register r31
- Purpose: allows the program to execute a separate block of code and remember where to come back to when it's done

`ret` (return):

- Definition: jumps to the address stored in the return address register
- Purpose: used at the end of function to send the processor back to the point in the main code immediately following the call instructions


Address
Contents
Begin execution here
i
i + 4
i + 8
i + 12
Load R2, A
Load R3, B
Add R4, R2, R3
Store R4, C
4-instruction program segment
A
B
C
Data for the program

3 Contrasting Between C and Java

3.1 Primitivity and Variables

Strings: Java has a separate type that represents strings.

```
String s = "Hello"
```

Instead of using strcat in C to concatenate strings, we can concatenate strings using the + operator.

Instead of doing `string1 == string2`, we need to do `string1.equals(string2)`. It returns a boolean.

To compare the two, we use `.compareTo`. A negative return value represents < and a positive value for >.

In C, variables have a type, it is declared along with the variable name, it cannot be changed, and the variable can hold values only of the declared type. This is the same in java:

```
String s = "Hello";
int t = 123;
int[] anArray = new int[3]; // new allocates memory for an object on the heap and
anArray[0] = 0;
anArray[1] = 1;
```

`double` is the usual choice for routine calculations involving floating-point values

In C, integer types have minimum sizes, not fixed sizes. In Java, integer types are also all signed (except char) and sizes are fixed.

Ulp: To handle numerical precision mistakes from `float` and `double`, we can use `ulp(...)` to find the difference between x and the next representable floating-point number greater than x. Therefore, `ulp` is commonly used to reason about floating-point error bounds.

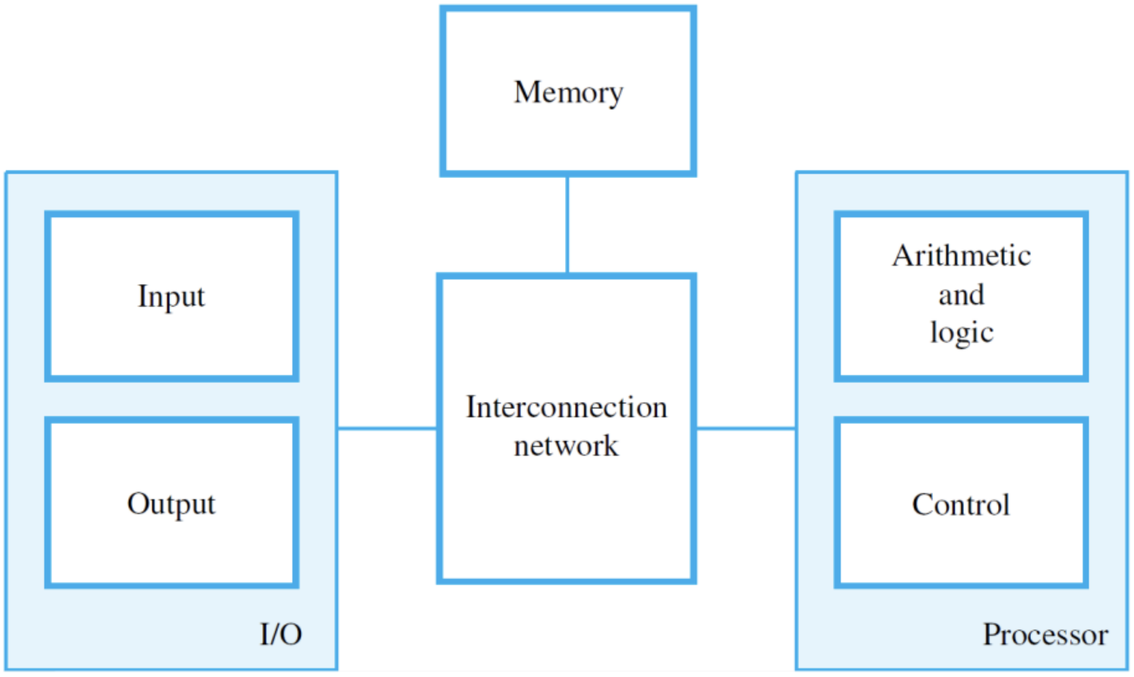
Boolean: In Java, we have a `boolean` type that can only be true or false.

In regards to integer overflow, adding 1 past the integer maximum values wraps it around to the minimum value, causing massive real-world failures.

4 Basic Structure of Computers

Definition: Computer architecture is the specification of a set of instructions and behaviour of hardware units

Functional Units: Computers consist of 5 basic units: input, memory, arithmetic and logic, output, and control. The interconnection network supports transfer of information between units. The **processor** includes arithmetic and logic with control. The **I/O System** includes input and output units together.



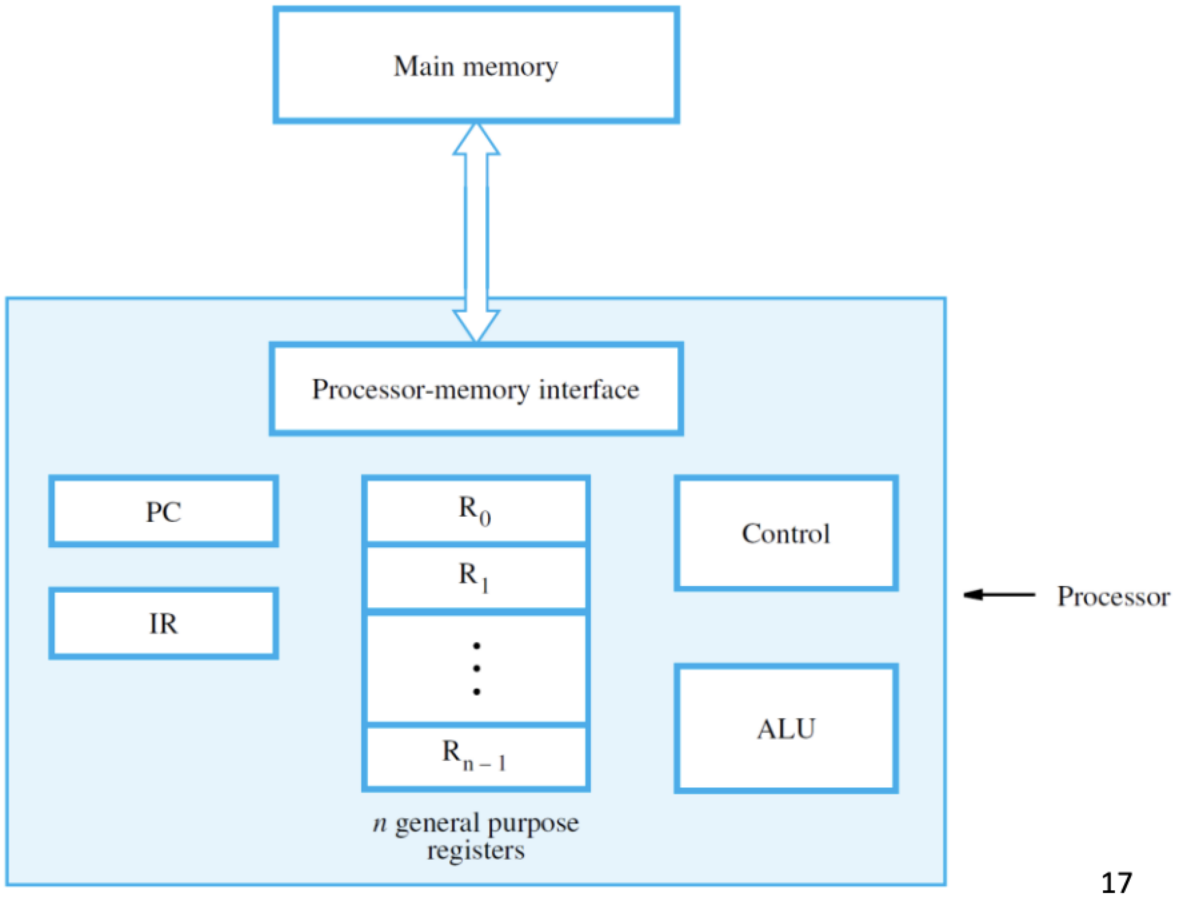
Input: Accepts coded information (in binary) for processing. Ex: mouse, keyboard, etc.

Memory: Stores programs (instruction lists) and data. Ex: RAM, SSD/HDD, cache

ALU: Performs arithmetic $(+, -, \cdot, /)$ and logic $(\wedge, \vee, \neg)$. Ex: high-speed registers for holding operands

Output: Presents processed results. Ex: displays, printers, and audio devices

Control: Coordinates all other units by sending timing and state signals. Ex: control circuits and control lines



How They Interact

1. **Instruction Fetching:** The **Control Unit** uses the **Program Counter (PC)** to find the address of the next instruction in **Memory**. This instruction is moved into the **Instruction Register (IR)** within the processor. During execution of each instruction, PC register is incremented by 4.
2. **Decoding:** The Control Unit interprets (decodes) the instruction in the IR to determine what action is required.
3. **Data Transfer:** Depending on the instruction, data may be moved from an **Input Unit** to Memory, or from Memory to **Processor Registers** (Load instruction).
4. **Processing:** If the instruction involves math or logic, the Control Unit directs the **ALU** to perform the operation using operands stored in registers.
5. **Storing and Outputting:** The result of a calculation is either kept in a register or written back to **Memory** (Store instruction). Finally, the **Output Unit** may transfer these results to a user or external device.

Instructions Running Cycle:

1. Fetch: The CPU fetches the instructions from memory. It uses the PC to know the memory address of the next instruction which is then loaded into the IR
2. Execute: The CPU carries out what the instruction says, could be ALU, jump/branch, moving data, etc.

5 Console Input and Output

Printing: In C, we use printf but must differentiate between types. In Java, we use print to print, and printIn to print and go to the next line.

```
int num = 212;
String subj = "CMPE";
System.out.print(subj);
System.out.print(" ");
System.out.printIn(num);

// or use String concatenation
System.out.printIn(subj + " " + num);
```

Scanner: importing `java.util.Scanner` and using the an object of the class Scanner enables reading data that the user types on the keyboard.

6 Arrays and Loops

Declaration:

```
int [] data = new int[10];

// or

int[] data;
data = new int[10];
```

To fill an existing array so that all elements have the same value, use the method `Arrays.fill`

```
import java.util.Arrays;
public class Lecture4 {
    public static void main(String[] args) {
        double[] someDbls = new double[10];
        System.out.println(Arrays.toString(someDbls));
        // fill array with 1.0
        Arrays.fill(someDbls, 1.0);
        System.out.println(Arrays.toString(someDbls));
    }
}
```

A regular for loop has four main parts:

1. an initialization expression
2. a termination condition
3. an update expression
4. a loop body

Syntax:

```
for (initialization; boolean_expression; update) {
    block_of_code;
}
```

Often, you will want to visit every element in a collection, not just a part. We can use a for each loop:

```
for (type variable: collection){
    // statements
}
```

Other keywords include `continue` and/or `break` statement to interrupt the execution of a loop, where the former returns control to the top of the loop and the latter transfers control to the first statement after the loop

Similarly to C, we can have $n-$dimensional arrays:

```
//single

int[] testArray;
testArray = new int[10];

//multi

int[][] twoDArray;
twoDArray = new int[10][];
```

We can Alias two objects, where they then point to the same data memory and changes to this data is represented in both aliases:

```
int[] first = {1,2,3,4,5};
int[] second = {10,20,30,40,50,60,70};
second = first;
```

In this case, the last two elements of `second` array are automatically sent to Garbage Collection, because we have moved outside of their scope.

MAKE NOTES ABOUT PASSING PARAMETERS AND ARRAYS BY REFERENCE

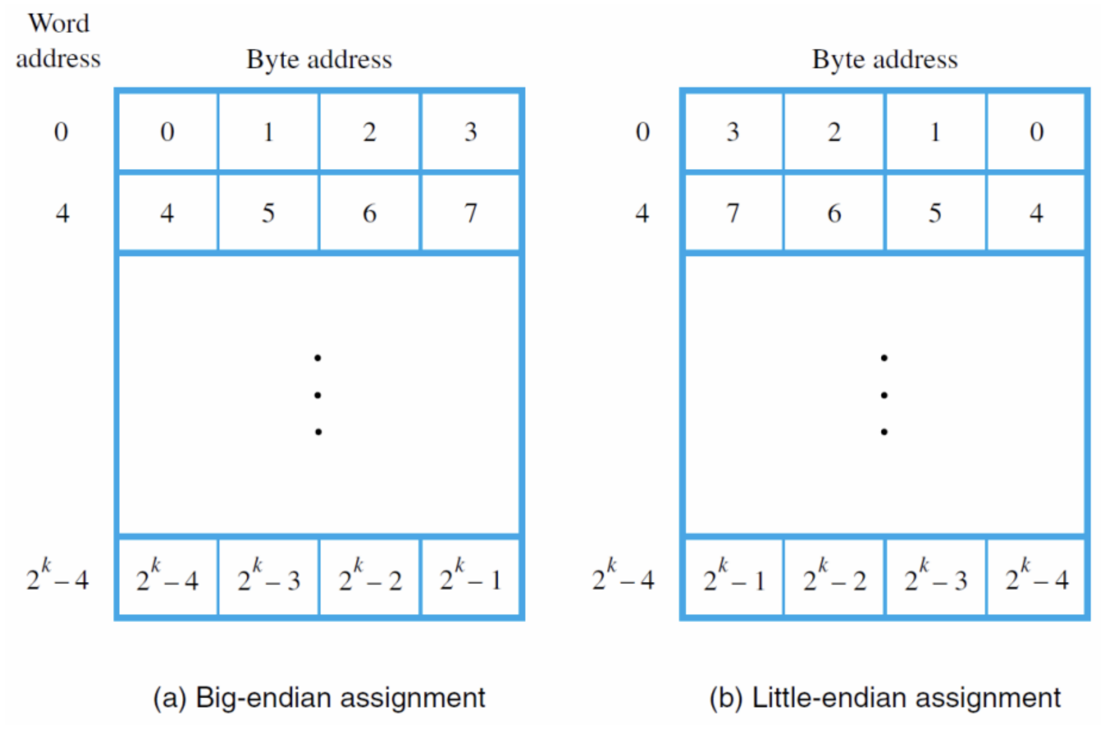
7 Instruction Set Architecture

A common word length is 32 bits. Numbers 0 to $2^k - 1$ are used as addresses for successive locations in the memory. Byte size is always 8 bits, but word length may range from 16 to 64 bits.

Byte locations have addresses $0, 1, 2, \ldots$, and word locations have addresses $0, 4, 8, \ldots$. We provide a byte-addressable memory that assigns an address to each byte. We have two ways to assign byte address across words.

Big-endian: Assigns lower addresses to more significant (leftmost) bytes of word

Little-endian: Uses opposite order.


Word address
Byte address
0
4
2^k-4
0 1 2 3
4 5 6 7
2^k-4 2^k-3 2^k-2 2^k-1
(a) Big-endian assignment
Byte address
0
4
2^k-4
3 2 1 0
7 6 5 4
2^k-1 2^k-2 2^k-3 2^k-4
(b) Little-endian assignment

7.1 Memory Organization

Hierarchy and Storage Types

Memory is organized into different levels to balance speed, capacity, and cost.

Primary Memory: This is fast, electronic memory composed of semiconductor storage cells. It is essential for storing programs and data currently in use.

Cache Memory: A smaller, faster electronic memory located on the same chip as the processor. It holds copies of instructions and data from the main memory that were recently used or are likely to be used soon, significantly speeding up access.

Secondary Storage: This provides large-capacity storage that retains information even when the power if off. While it is less expensive per bit, i tis generally slower than primary memory and has traditionally been based on magnetic or optical devices, though it now includes flash memory.

Physical and Logical Organization

The physical structure of memory dictates how the processor interacts with it.

- Binary Representation: Information is stored in bits
- Words: Bits are grouped into multi-bit “words” (typically 32 bits) to allow the processor to access multiple bits simultaneously for efficiency
- Addressing: Each word location has a unique address, numbered consecutively starting at 0
- Random Access Memory: Memory is organized so that any location can be accessed in a fixed, short amount of time, regardless of where the data is physically located

8 Defining Classes

A **class** is a template that defines a type. An **object** is a concrete instance of a class. A **method** is a function defined within a class that describes behavior.

8.1 Methods

In Java, a **method** is a block of code that is **associated with an object** or defined inside a class. Every "function" in Java is actually a method because it must reside within a class definition.

Example: In C, the focus is on the function, and the data is separate from the logic that processes it.

In Java, we encapsulate the data and the logic into a **Class**. The "function" becomes a method belonging to that class.

```java
public class Circle {
    // A Method (specifically a static method for utility)
    public static double calculateArea(double radius) {
        return Math.PI * radius * radius;
    }

    public static void main(String[] args) {
        double r = 5.0;
        // Calling the method through its Class name
        double area = Circle.calculateArea(r);
        System.out.println("Area: " + area);
    }
}
```

Definition: In Object-Oriented Programming, we use Encapsulation to bundle data and the methods that operate on that data into a single unit called a **class**.

8.2 Class

Definition: A **class** is a formal template used to create objects and define their data types and behaviours.

```java
// template

public class ShowStructure {
    // instance variables or "attributes" (fields) here
    // methods here
} // end class ShowStructure
```

```java
// method syntax
[private|public] [static] [final] returnType methodName ([parameterList]) {...}
```

Fields declared in a class defines their "scope". We can control their privacy and the way they are stored in memory using `public/private/protected` and `static`

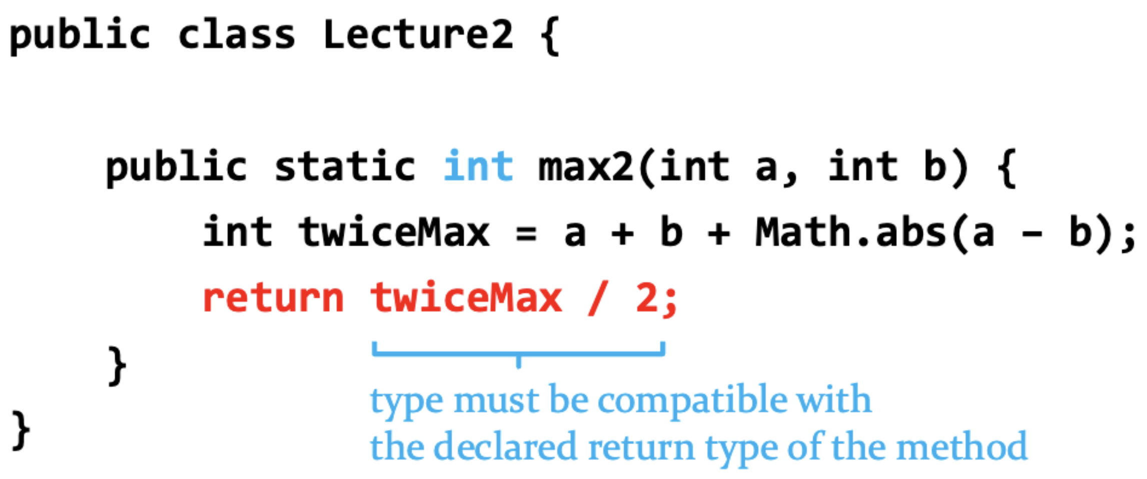
```java
// field syntax
[private|public] [static] [final] type attributeName [= literalValue];
```

```java
// field examples
public static double aVar;
public static int aNum = 100;
private static String hello = "Hello";
```

Definition: The **public** access modifier on a top-level class means that the class is visible to all other classes.

Definition: The **static** modifier on a method means that the method is associated with its class.

In Java, the return value must be compatible with the declared return type of the method.



```java
public class Lecture2 {

    public static int max2(int a, int b) {
        int twiceMax = a + b + Math.abs(a - b);
        return twiceMax / 2;
    }
}
```

type must be compatible with
the declared return type of the method

A void method has no return statement, but an empty return statement is legal.

Example:

```java
public void printHelloName (String yourName) {
    System.out.println("Hello " + yourName);
} // end printHelloName
```

You can use the ellipsis operator (. . .) to create a parameter that can take any number of arguments of that type.

```java
// example
public void seeBling (Bling... blingers) {// code}
```

Within `seeBling`, you get at the individual arguments of type Bling by pretending that blingers is an array

Full Example:

```java
public class Simple {
    public static int aNum = 100;
    public static int sumNums(int num1, int num2) {
        return num1 + num2;
    }
    public static void main(String[] args) {
        int anotherNum = 200;
        System.out.println(sumNums(aNum, anotherNum));
    }
}
```

To reference a method within a class, we must `import` the class, unlike `#include` in C

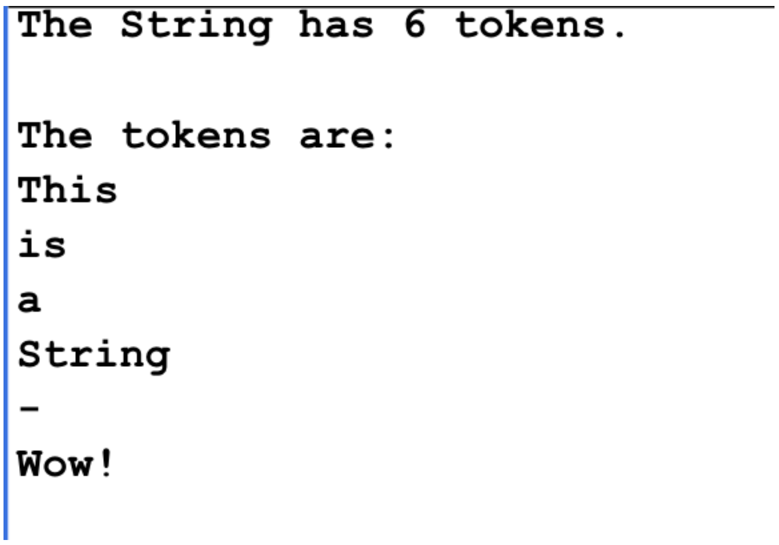
8.3 Methods and Java.lang

A **method** is written to avoid repeating code. They should be short, do only one thing, and do it well. A code in a method should be at the same level of abstraction and use less than three parameters wherever possible.

Java.lang is a core Java package that contains the fundamental classes required for almost any Java program. Conceptually, it sits at the lowest level of abstraction for everyday Java programming.

Fundamental Classes	Function
String	The root superclass of all classes in Java
Integer, Double, Boolean	Wrapper classes for object representations of primitives
Math	Common math functions
System	Standard I/O, environment access, garbage collection hooks
Thread, Runnable	Basic concurrency primitives
Throwable, Exception, RuntimeException, Error	The exception hierarchy
StringTokenizer	Parsing strings to pieces, i.e. “tokens”

```java
// example StringTokenizer
String aString = "This is a String - Wow!";
StringTokenizer st = new StringTokenizer(aString);
System.out.println("The String has " +
    st.countTokens() + " tokens.");
System.out.println("\nThe tokens are:");
while (st.hasMoreTokens()) {
    System.out.println(st.nextToken());
} // end while
```



```
The String has 6 tokens.

The tokens are:
This
is
a
String
-
Wow!
```

Method Overloading: Overloading is when a method name is used more than once in method declarations within the same class. The rule is that no two methods with the same name within a class can have the same number and/or types of parameters in the method declarations.

Method Overriding: Where a subclass provides its own implementation of a pre-defined method from the super class.

8.4 Constructors

Super: A derived class inherits all instance variables from its base class, but it cannot access private base class variables directly. To initialize this inherited data, a derived class constructor uses the `super` keyword to call a constructor of the base class

The syntax is `super(Argument_List);` and must always be the first action taken in a derived class constructor definition. You cannot use a base class name to call its constructor.

This: When defining multiple constructors in the same class (overloading), it is often convenient for one constructor to call another. This is done using the keyword `this`. Using `this(name, date)` inside a constructor to trigger a different constructor in the same file is a good example.

Example:

Base:

```java
public class Employee {
    private String name;
    private Date hireDate;

    // No-argument constructor
    public Employee() {
        name = "No name";
        hireDate = new Date("January", 1, 1000); // Placeholder
    }

    // Parameterized constructor
    public Employee(String theName, Date theDate) {
        name = theName;
        hireDate = new Date(theDate); // Uses Date's copy constructor
    }

    // Base class copy constructor
    public Employee(Employee originalObject) {
        name = originalObject.name;
        hireDate = new Date(originalObject.hireDate);
    }
    // ... Accessors/Mutators/toString ...
}
```

Derived:

```
public class HourlyEmployee extends Employee {
    private double wageRate;
    private double hours;

    // A. Using 'this' to call another constructor in the SAME class
    public HourlyEmployee() {
        // Calls the constructor below (C) with default values
        this("No name", new Date("January", 1, 1000), 0, 0);
    }

    // B. Derived Class Copy Constructor using 'super'
    public HourlyEmployee(HourlyEmployee originalObject) {
        // Calls the base class (Employee) copy constructor to set name/hireDate
        super(originalObject);

        // Sets the specific instance variables for this derived class
        wageRate = originalObject.wageRate;
        hours = originalObject.hours;
    }

    // C. Using 'super' to call a BASE class constructor
    public HourlyEmployee(String theName, Date theDate,
                          double theWageRate, double theHours) {
        // Must be the FIRST action: initializes inherited name and hireDate
        super(theName, theDate);

        // Initializes derived-only variables
        wageRate = theWageRate;
        hours = theHours;
    }
}
```

8.5 Memory

Memory: Java has no mechanism for accessing memory directly. Therefore, there are no pointer types.

Java programmers generally are not concerned with allocating memory for objects. Except for arrays where the size of the array has to be specified. De-allocating memory sued by objects that are no longer needed is done automatically by the garbage collector.

Reference Type: Any type that begins with a capital letter is a reference type, like String, List, etc. Classes are user-defined reference types.

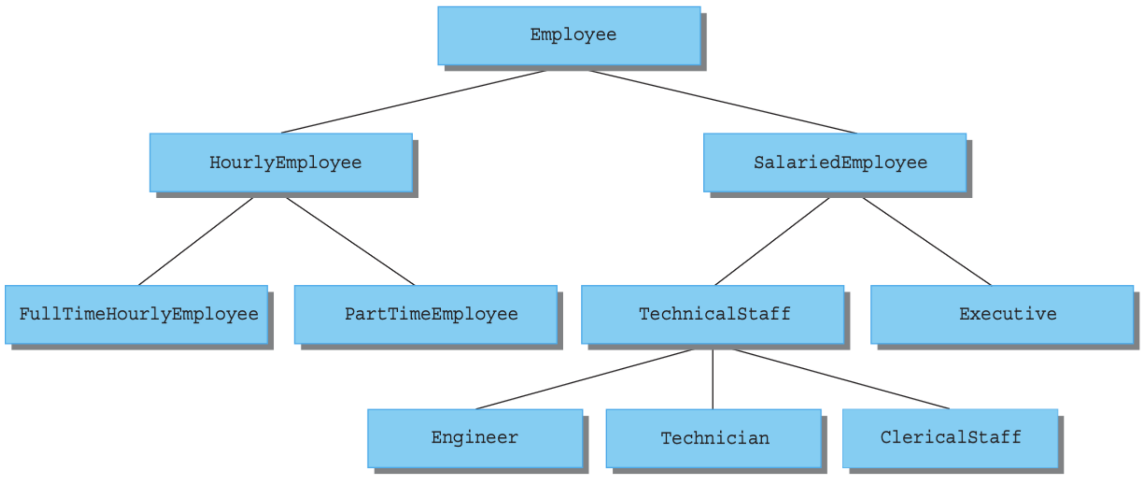
Primitive Type: Any type that begins with a lowercase letter is a primitive type. boolean, byte, char, short, int, long, float, double. All primitive types occupy a defined fixed amount of memory.

9 Inheritance and Encapsulation

9.1 Inheritance

Definition: One of the main techniques of OOP is **inheritance**, which means that a very general form of a class can be defined and compiled. Later, more specialized versions of that class may be defined by starting with predefined definition and adding variables and methods. The specialized classes are said to *inherit* the methods of the predefined class.

For example, we can create a class hierarchy, starting with the class `Employee` and continuing with *derived classes* for different kinds of employees:



9.1.1 Derived Classes

A derived class, or a subclass, is built upon a base class, or a superclass.

Adding the `final` modifier to the definition of a method/class indicates that it may not be redefined in a derived class.

```java
// Base class
class Animal {
    void eat() {
        System.out.println("This animal eats food.");
    }
}

// Derived class
class Dog extends Animal {
    void bark() {
        System.out.println("The dog barks.");
    }
}

// Main class to test
public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();  // Create an object of the derived class
        myDog.eat();            // Inherited method from Animal
        myDog.bark();           // Method of Dog
    }
}
```

Definition: A base class is often called the **parent class**. A derived class is then called a child class, therefore we have ancestor classes and descendent classes.

9.1.2 Overriding a Method

If a derived class requires a different definition for an inherited method, the method may be redefined in the derived class. This is called **overriding** the method definition:

```java
// Base class
class Animal {
    void sound() {
        System.out.println("This animal makes a sound.");
    }
}

// Derived class
class Dog extends Animal {
    // Overriding the sound() method
    @Override
    void sound() {
        System.out.println("The dog barks.");
    }
}

// Main class
public class Main {
    public static void main(String[] args) {
        Animal myAnimal = new Animal();
        myAnimal.sound();   // Calls Animal's method

        Dog myDog = new Dog();
        myDog.sound();      // Calls overridden method in Dog
    }
}
```

10 Polymorphism

11 Exception Handling

Exception Object: If a method throws an exception, then that method is immediately halted and there is no need for any return value, even if the method is non-void. The exception does not do anything about an error or a problem, but it stops the program.

Try/Catch: We use a try/catch block for an exception object.

```
try {
    // block of statements that might generate an exception
} catch (exception_type identifier) {
    // block of statements
} [ catch (exception_type identifier) {
    // block of statements
...
} ] [ finally {
    // block of statements
} ]
```