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 Contrasting Between C and Java

2.1 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.

3 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.

4 Arrays, Loops, and Lists

4.1 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.

4.2 Lists

A java list almost always hold elements of one type and have dynamic size.

Operations:

- `.add(#)` adds the value # to the list
- `.get(#)` gets the element by index # of the list
- `.set(#, "string")` sets the element by index # to a string
- `.remove(#)` removes the element of index # of the list
- `.remove(Integer.valueOf(#))` removes the value # of the list
- `Collections.reverse(list)` reverses the order of the elements in a list

5 Maps

A map is an abstract data type (ADT). An ADT is a formal specification of a type that defines logical behaviour, whereas a data structure is the specific implementation of that type intended for information storage and retrieval.

A map models a group of elements that are accessed by keys. They have:

- Unique Keys: The keys within a map must be unique, it is a set
- Mapped Values: Each key maps to exactly one value
- Non-Unique Values: Multiple different keys can map to the same value, therefore the values form a collection but not necessarily a set

5.1 Implementation

Java provides several classes that define how it is implemented. The most common implementation is with `HashMap` in `java.util.HashMap`. It has $O(1)$ complexity and the keys are unordered.

A `TreeMap` in `java.util.TreeMap` is a dictionary that sorts its keys, with $O(\log n)$ time complexity.

The `LinkedHashMap` in `java.util.LinkedHashMap` maintains the key sin order, has $O(1)$ complexity, but is slightly slowly due to maintenance of linked list ordering.

5.2 Working with Maps

Add entries: `map.put(key, value)`

Retrieve a value: `get(key)`

Check if contains key: `!map.containsKey(key)`

A common application of Maps is counting the occurrences of items in a list.

```
// 't' is an existing List of Strings
HashMap<String, Integer> count = new HashMap<>();

for (String s : t) {
    if (count.containsKey(s)) {
        // If string exists, increment current count
        Integer n = count.get(s);
        count.put(s, n + 1);
    } else {
        // If string is new, initialize count at 1
        count.put(s, 1);
    }
}
```

While keys are a set and keySet method returns a set, the values method returns a collection because values are not necessarily unique.

5.3 The Java Collection Framework

The Java collection framework is a unified architecture for representing and manipulating collections. It consists of:

- Interfaces (specifications of method signatures)
- Classes (concrete implementations)
- Algorithms (methods like sorting and searching)

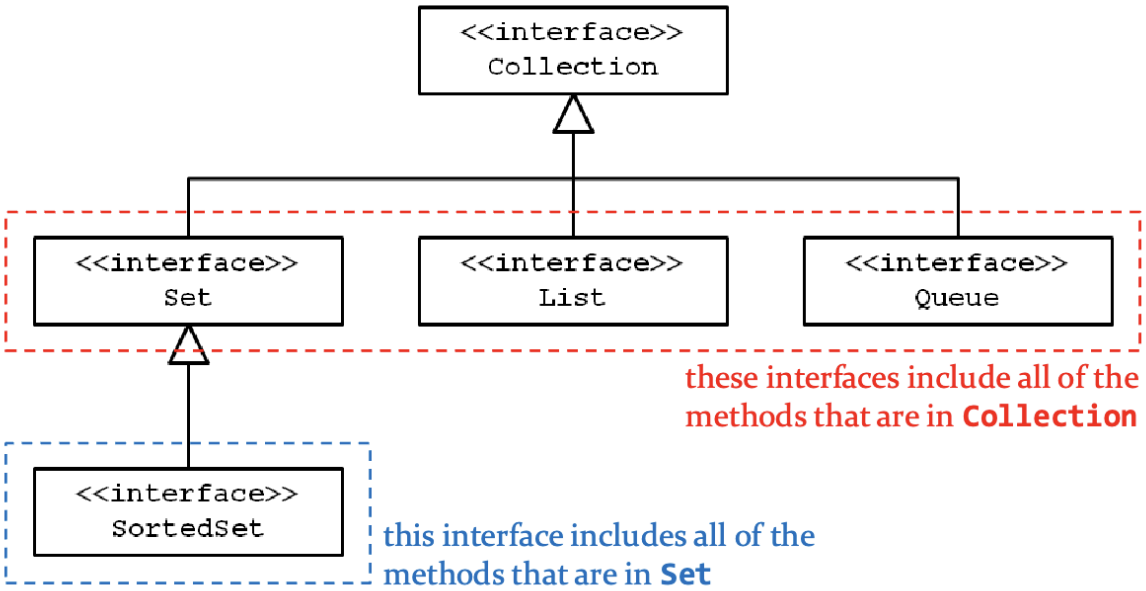
Interfaces are reference data types, meaning you can make a variable or parameter whose type is an interface

```
List<String> t = new ArrayList<String>();
```

interface — implements the interface

```
Map<String, Integer> count = new HashMap<>();
```

The Collection interface defines general operations like `size`, `isEmpty`, `add`, and `remove`. Its children interfaces include Set, List, and Queue.



The Map interface does not extend Collection, but it provides "Collection Views" that allow map data to be manipulated using standard Collection logic

6 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 behaviour.

A type is a set of values and the operations that can be performed with those values.

Examples:

- `java.lang.String` needs a sequence of zero or more Unicode characters supporting ==, equals, charAt, indexOf, substring, etc. Strings do not have 'get' because they are immutable
- `java.util.List` needs a sequence of zero or more elements of the same type supporting ==, equals, get, set, subList, etc.

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"

6.1 The Object Model

Object-oriented programming focuses on decomposing programs into interacting objects that contain data and can operate on their own data

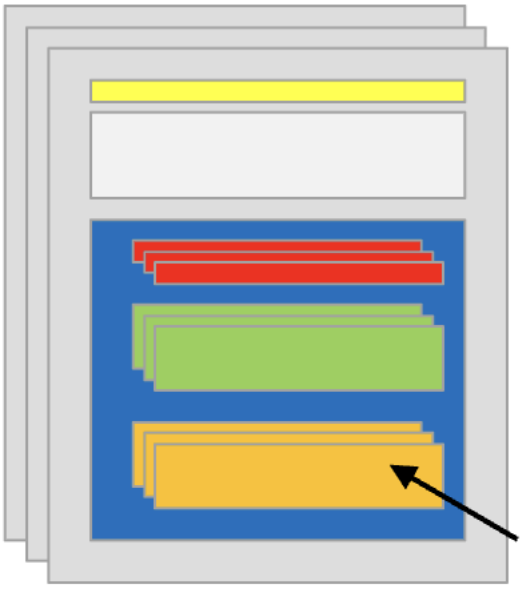
Abstraction: An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer

Encapsulation: Encapsulation is the process of compartmentalizing the elements of an abstraction that constitute its structure and behaviour; encapsulation serves to separate the contractual interface of an abstraction
and its implementation

Modularity: Modularity in object-oriented programming (OOP) is the practice of dividing a software system into smaller, independent, and interchangeable modules (classes or packages) that interact through well-defined interfaces

See below the high level organization of a typical java program

Organization of a Typical Java Program



- one or more files
- zero or one package name
- zero or more import statements
- one class
- zero or more fields (class variables)
- zero or more more constructors
- zero or more methods

zero or one package declaration
zero or more import statements
one or more class declarations

```
package ca.queensu.cs.cisc124.notes.basics;
import java.util.ArrayList;
public class Stack {
    private ArrayList<String> stack;
    public Stack() {
        this.stack = new ArrayList<>();
    }
    public int size() {
        return this.stack.size();
    }
    public void push(String elem) {
        this.stack.add(elem);
    }
    public String pop() {
        String elem = this.stack.remove(this.size() - 1);
        return elem;
    }
    public String toString() {
        StringBuilder b = new StringBuilder("ListStack:");
        if (this.size() != 0) {
            for (int i = this.size() - 1; i >= 0; i--) {
                b.append('\n');
                b.append(this.stack.get(i));
            }
        }
        return b.toString();
    }
}
```

zero or more fields
zero or more constructors
zero or more methods

6.2 Class

Definition: A **class** is a formal template used to create objects and define their data types and behaviours. A class is a blueprint that is used to make objects. A reference is an identifier that can be used to find a particular object.

Example: `List<String> t = new ArrayList<>()`

ArrayList is a class, and t is the reference to the object.

```
// template

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

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`

Access modifiers on fields:

- **Private:** field is visible only to the class that the field is in
- **no access modifier:** same as private and field is also visible to all other classes in the same package
- **Protected:** same as package private and field is also visible to all subclasses in the same package
- **Public:** field is visible to all classes everywhere. A field that represented a constant value can be public. All other fields should be private

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

In Java, the return value 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:

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

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

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**.

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.

6.3 Constructors

The purpose of a constructor is to initialize the state of an object

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.

Copy: A copy constructor initializes the state of an object by copying the state of another object (having the same type)

Simple Example:

```
/* initializes this object by copying from 'other' object
*/
public Point2(Point2 other) {
    this.x = other.x;
    this.y = other.y;
}
```

6.4 Full Stopwatch Example

```java
package ca.queensu.cs.cisc124.notes.basics.stopwatch;

/**
 * A simple stopwatch for recording elapsed time.
 */
public class Stopwatch {

    /**
     * Raw time (from nanoTime) when the stopwatch is started
     */
    private long rawStartTime;

    /**
     * Raw time (from nanoTime) when the stopwatch is stopped
     */
    private long rawStopTime;

    /**
     * True if stopwatch is started and not stopped, false otherwise
     */
    private boolean isRunning;

    /**
     * Number of nanoseconds in one second
     */
    private final static long BILLION = 1000000000L;

    /**
     * Initializes this stopwatch so that the elapsed time
     * is zero seconds.
     */
    public Stopwatch() {
        this.rawStartTime = 0L;
        this.rawStopTime = 0L;
        this.isRunning = false;
    }

    /**
     * Starts this stopwatch running. Does nothing if this
     * stopwatch is already running.
     */
    public void start() {
        if (!this.isRunning) {
            this.rawStartTime = System.nanoTime();
        }
        this.isRunning = true;
    }

    /**
     * Stops this stopwatch and returns the elapsed time in
     * seconds since the watch was started. If the stopwatch is
     * already stopped then the elapsed time in seconds (as
```

6.5 Javadoc Documentation

A doc comment is written in HTML and includes a description and a block of tags. The first sentence of the description is coped to the summary section of the generated document. Block tags begin with the @ symbol to provide specific details. It is common to use the paragraph tag <p> to add paragraphs to the generated documentation

- @*param* : Used to describe a parameter in a constructor or method
- @*return* : Used to describe the value returned by a method
- @*throws* : Used to describe an exception and the specific conditions under which it is thrown

Method precondition: This is used to show that a method requires a condition to be true before the method is run.

Example:

```
/**
* Divide this point by a scalar value changing the coordinates of this point (desc
*
* <p>
* Use this method when you want to write something like
* {@code p = p/s} where {@code p} is a point and {@code s}
* is a scalar
*
* @param s the scalar value to divide this point by; must
* not be equal to zero
* @return a reference to this point
*/
```

If a method is called when a precondition is not satisfied, the consequences may include the method returning the wrong value, encountering a runtime error, or putting the object into an invalid state.

Preconditions on public methods should be enforced by explicitly checking the argument values. A method should throw an exception to indicate that a precondition has not been satisfied.

Exceptions: This is shorthand for exceptional event, meaning the method creates an exception object containing information about the error type and the state of the program at the time of error

Common Examples:

- IllegalArgumentException
- IndexOutOfBoundsException
- NullPointerException

Postcondition: This is a condition that the method must ensure is true immediately after it finishes running. We can add an `assert` statement to test the condition, and if false, then throw error.

Class Invariants: An invariant is a state of the object that is always true. For example,

a Counter class must always have nonnegative integer value. We can run checkInvariant to check

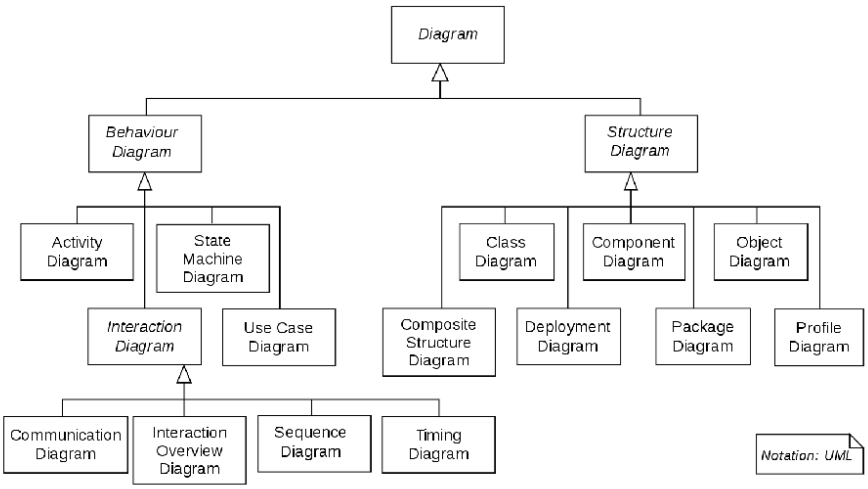
Full Example:

```java
package ca.queensu.cs.cisc124.notes.basics.geometry;

/**
 * A Cartesian point in 2-dimensions having real coordinates.
 */
public class Point2 {

    /**
     * The coordinates of this point.
     */
    private double x;
    private double y;

    /**
     * Initializes the coordinates of this point to {@code (0.0, 0.0)}.
     */
    public Point2() {
        this.x = 0.0;
        this.y = 0.0;
    }

    /**
     * Initializes the elements of this point to {@code (x, y)} where
     * {@code x} and {@code y} are specified by the caller.
     * * @param x the x value of this point
     * @param y the y value of this point
     */
    public Point2(double x, double y) {
        this.x = x;
        this.y = y;
    }

    /**
     * Sets the x and y coordinate to the specified values.
     * * @param newX the new x coordinate
     * @param newY the new y coordinate
     * @return a reference to this point
     */
    public Point2 set(double newX, double newY) {
        this.x = newX;
        this.y = newY;
        return this;
    }

    /**
     * Divide this point by a scalar value changing the coordinates
     * of this point.
     * * <p>Use this method when you want to write something like
     * {@code p = p / s} where {@code p} is a point and {@code s}
     * is a scalar.
     * * @param s the scalar value to divide this point by; must not be
     * equal to zero
     * @return a reference to this point
     * @throws IllegalArgumentException if s == 0.0 is true
```

6.6 UML Class Diagram

Unified Modelling Language (UML)

- a visual modelling language used in software engineering for visualizing the design of a software system

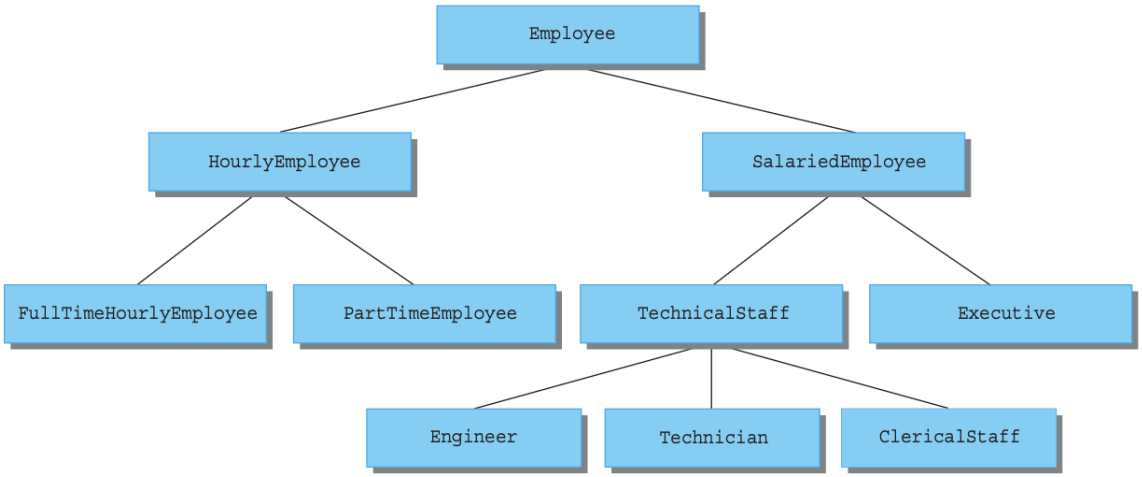


7 Inheritance and Encapsulation

7.1 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:



7.1.1 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.

7.1.2 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
    }
}
```

8 Polymorphism

9 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
} ]
```

10 Sets

There is no indexing or slicing using an index in sets. They support mathematical operations such as intersection, union, and difference.

A special type of set is a Hashset in the `java.util.HashSet` library with $O(1)$ expected complexity

```
HashSet<String> t = new Hashset<>();
```

We could also use a `TreeSet` with $O(\log n)$ complexity with natural ordering but slowly operations than HashSet

A third type is `LinkedHashSet` which has guaranteed order by linkage.

10.1 Operations

```
// union
HashSet<String> s1 = new HashSet<String>(s1);
s1.addAll(s2); // s1 U s2
```