Lecture Notes – FITE2000

Notes by José A. Espiño P.

Semester 2 2022–2023

Contents

1 Introduction to Java

Java is a simple, object–oriented (focuses on data and methods to manipulate the data, as well as supports extensive sets of classes), distributed, high performance programming language. One can also say that Java is robust: it has no pointers, does garbage collection automatically, run–time checking, and exception handling

It is a compiled language, but rather than being compiled straight to executable machine code by the interpreter, it is compiled to an intermediate binary form called JVM (Java Virtual Machine) byte code. The byte code is then compiled and/or interpreted to run the program. This adds portability, since the implementation of JVM varies according to the computer.

The Java Platform has two main components, the Java Virtual Machine — usually seen as a `.class` file — and the Java API (Application Programming Interface) — usually seen as a `.java` file. An API is a large collection of functions and components grouped into libraries (also called packages). The API

and VM divorce the Java program from hardware and operating system dependencies.

Object Oriented Programming has four core principles:

1. Abstraction
   It refers to taking away unimportant details; only keeping characteristics relevant to the application in question.
   An example of this is a wallet app: what features are available to card providers but hidden to regular users?

2. Encapsulation
   It consists in treating the ensemble of data (including implementation) and operations as a whole unit. Essentially, it is hiding the implementation, adopting a "black box" model in its stead. Another way to see encapsulation is by seeing a group of methods and variables as a class.
   The presence of encapsulation in Java means that we need to provide methods to get and set the data as well. Some benefits of this principle are better access to control and security and a more portable code.
   For instance, in a wallet app, how are cards and their information stored? It could be via an array, a stack, a queue, hash table, ... Encapsulation leaves this as an unknown to us!

3. Inheritance
   Allows us to create a new class (child class, subclass) from an existing one (parent class, superclass.)

4. Polymorphism
   When we have many classes related to each other via inheritance, but where each uses the methods of the superclass in a different way.

A Java program can essentially be regarded as a collection of classes. In order to use a class, we need to create an object using a constructor and access the method of the class through the created object. For example, if we have this class:

```
class XX{
private
     int i;
     void f1(   .);

public
     int j;
     void f2(   .);
}
```

then,

```
XX obj1 = new XX( );
XX obj2 = new XX( );

obj1.j = 10;
```

```
ojb2.j = 20;
/* notice that each object has its own copy of i,j */
```

Important details to keep in mind about a Java program:

- Java requires declaration of variables and explicit creation of objects
- There must be a `main` method in the program; it will be the entry point for the execution
- Operators:
  Arithmetic: `+, -, *, /, %`
  Comparison: `>, >=, <, <=, ==, !=`
  Logical: `&&, !,` ——
  OOP: `new, instanceof`
- All Java statements end with ;
- Java uses { and } to denote blocks of code (as opposed to indentation in languages like Python.)
- For–loop syntax: `for (initialization; termination_cond; inc) {`statements(s) —

Data types are the classification of data items; they allow the computer to know which operations can be validly performed on them.
Java has two big data types:

- Primitive
  - Integers, which can be byte (8 bits), short (16 bits), int (32 bits), or long (64 bits)
  - Real, which can be float (32 bits) or double (64 bits)
  - Others, such as char (16 bits) and boolean (true or false)
- Reference They are like a pointer.
  - array
  - class

The scope of a variable is the extent of the program code within which the variable can be accessed, declared, or worked with. In Java, there are no global variables: we have **member variables**, whose scope is its class, **method parameters**, whose scope is the method where they are being used, and **local method variables**, whose scope is within the block of code where it is declared.

2 Java Array and String

An array is a sequence of items of the same type (class). In Java, they are declared using the template `type[] array_name = new type[array_size];`. Note that arrays in Java are undynamic: a certain amount of memory is allocated to it in the heap upon declaration, and this value won't change throughout the program execution. When it comes to command line arguments, however, JVM will allocate space for the arguments dynamically — there is no need to instantiate the array. This can be added in the function declaration, for example `public static void main (String[] args){}`, and then this can be accessed through `args[i]`.
Another important fact is that when we create an array of reference type, each of the entries within the array will be `NULL` upon initialisation. We need to create an object in each entry. Imagine the created array as a notebook with n pages. In order to actually read anything from the notebook, we must write or stick something onto the pages first. Given a user–created class `Student`, a way of initialising such an array is:

```
Student[] stud_array = new Student[10];
stud_array[0] = new Student();
stud_array[0].name =  T .M. Chan ;
```

Alternatively, we could also initialise it through:

```
Student stud_array[] = new Student[] {new Student();
    new Student( T .M. Chan );
    new Student( T .M. Chan , 60);
    null };
```

The array class has a defined data field `length`, e.g. `temp.length`. In any array, length is a data field that can always be accessed through `array_name.length`.

A string is a sequence of characters. In Java, it is defined in the `java.lang` package. It behaves in a way similar to an array of characters. It is important to notice that the String class is immutable for constant strings.
When you want to compare two Strings (or objects of the reference type) you should not use `==`. This operator will compare the String reference addresses. The way to compare the content of the strings is with the `string_one.equals(string_two)` method.
Strings can be concatenated with the `+` operator. Other String methods are `substring()` and `toLowerCase()`.

An ArrayList is a Java class that can be found in the `java.util` package. Functionally, it is a resizable array. Its size can be modified dynamically. When declaring a new ArrayList object, the type of its contents must be specified parametrically (`ArrayList<Type>`). Take a look at the following code excerpt:

```
import java.util.ArrayList;
public class Main {
  public static void main(String[] args) {
    ArrayList<String> cars = new ArrayList<String>();
```

```
    cars.add("Volvo");
    cars.add("BMW");
    cars.add("Ford");
    cars.add("Mazda");
    System.out.println(cars);
  }
}
```

Some frequently–used methods for ArrayList are the following:

- Declaration
  `ArrayList<Type> array_name = new ArrayList<Type>(size);`
- Adding a new element
  `array_name.add(new Type(element1, element2));`
- Obtain array size
  `arrayname.size()`
- Retrieving the element at a given index
  `arrayname.get(index)`

3 Java Objects and Classes

Java classes contain the two basic elements of any Java program, data (member variables) and operations (methods). Classes are used by creating an object using a constructor and then accessing the method of the class through the created object. We use the operator `new` to create a new object.
A class is a template that will tell you the variables and methods that are common to all objects of this class. An object is a set of related variables and methods that aim to model a real–world object.
An example of a Java class:

```
public class Student{
  private String name; /* variable */
  private int mark;
  static private int tot_stud;

  public int getMark(); /* method */
  public String getName();
  static public void updateTot(inti);
  }

Student s1 = new Student();
s1.name = "Jose";
Student s2 = new Student();
s2.name = "Alex";

int total_students;
```

There are two types of classes: public and private. It is standard Programming practice to define all variables private, because you don't want any external functions to manipulate those objects. Public classes can be accessed anywhere,

whereas private ones can only be accessed within its own class.
Static refers to a part of a class that won't change when we create new objects within that class. It is a set value and will be shared amongst all objects of the same class. In the example above, all the `Student` objects will share the same value for `tot_stud`. If you wanted to access or update the value of `tot_stud`, you would have to call it by `Student.tot_stud`. Using `s1.tot_stud` would not work. In terms of storage, static variables are efficient because no matter how many objects in that class you will create, Java will not create more copies of the static variable in particular.
A variable declared `private` can be accessed from the main program, but cannot be accessed from a different class.

The typical life cycle of an object is the following: creation, use of the object, and lastly, destruction of the object. Creation has three stages:

- Declaration
  `class_name variable_name;`
- Instantiation
  `variable_name = new class_name();`
  The constructor has the same name as the class and is invoked by `new`. Java will provide a default constructor, `class_name()`, which does nothing
- Initialisation
  You can use a constructor to initialise objects. More than one constructor can be defined in a given class if the parameter lists are different:

```
1  public class Student{
2         private String name;
3         private int mark;
4
5
6         public Student(String s){
7               name = s;
8         }
9         public Student(int x){
10              mark = x;
11        }
12
13 }
```

  Constructors are also accompanied by specifiers. These will control which class can call the constructor: Java classes are also accompanied by a specifier upon being defined:

4 Inheritance

Inheritance is the process of creating classes from pre–existing classes. This allows us to reutilise methods and variables, as well as to establish relationships between different iterations of the same superclass.

A superclass is a class's direct ancestor **and** all its ancestors. Subclasses will automatically inherit all variables and methods contained in the superclass, but they can only access the members in the superclasses that are `public`, `protected`, and `package` (if they are in the same package.) This leads us to the question, what is a package? In Java, it is a collection of classes, interfaces, and other reference types. They serve to group related classes and define a namespace for the classes in it. Each Java class is part of a package; the first statement in a source file usually specifies which package the class is part of. If there is no source package, the class is part of an unnamed default package.
Usually the package name relates to the directory structure where the class is at. One usually stores a Java class in a directory with a relative directory path that matches the package name for that class. For example, consider

```
1 package classes.geometry;
2  import java.awt.Dimension;
3  public class Shape {
4   // The implementation for the Shape class would be coded here ...
5  }
```

In that code excerpt, Java will try to find the Shape class and store it in the relative directory structure `classes\geometry`.
Some common Java packages are `java.lang` (includes fundamental classes) and `java.io` (includes input/output classes).
All classes declared in a program are subclasses of the built–in root class **object**. This means that everything in Java, but elementary types, are considered objects.
There are two important keywords related to this idea: `this` refers to the current object or member variables of it. `super` refers to a hidden member of a superclass or its constructor. Java inheritance is characterised by the keyword `extends`. Superclasses tend to be more general, subclasses more specific.
When designing an *inheritance tree*, we need to design the parent class: what are the common variables to be shared amongst all subclasses? What methods do they all share?
Then, we need to decide if a subclass needs method implementations that are specific to that particular subclass type. As we design the tree, we must look for opportunities to use abstraction: find subclasses that might need common methods.

5 Polymorphism

Normally, we declare an object as follows:
Class Dog ... Dog myDog = new Dog(); —
We firstly declare a reference variable (`Dog myDog`); then, we create an object (`new Dog()`), and lastly, we link the object and the reference using `=`. Here, the reference type and the object are the same.
This does not have to be the case all the time. With polymorphism, the refer-

ence type and the object type can be difference: `Animal myDog = new Dog();`. Here, the reference type is a superclass of the actual object type. Why is this useful? We can build polymorphic arrays, arrays whose members are all subclasses of a common superclass. Imagine having an array of type `Animal` representing a zoo, and every element of this array being a type of animal defined as a subclass of `Animal`.
Polymorphism also presents us with an "enhanced" for–loop: for (declaration : expression) statement /* for example */ for (BankingAccount acnt:accounts) total$_i nterests+ = acnt.calInterest(); /*accountsisanArrayListofBankingAccount*$ /|

6 Abstract Class

An abstract class is one nobody can make a new instance of. It can be declared reference type for the purpose of polymorphism. For example, imagine we have the following superclass:

```
abstract public class Animal{ ... }
abstract public class Canine extends Animal {
    public void roam() {}
}
```

Then,

```
public class MakeCanine{
    public void go(){
        Canine c;
        c = new Dog();
        c = new Canine(); /* this line will generate a compile
    error */
        c.roam();
    }

}
```

From this example, you can see it is not possible to create an instance of just `Canine`. There are also abstract methods: they must be defined inside an abstract class and have no body. These methods are implemented by defining the body for the method: the first concrete class in the inheritance tree **must** implement all abstract methods.
As you might have noticed, every class extends from `Object`. What is class `Object` then? This class has certain methods defined in it:

- equals(Object o): checks if 2 objects are considered equal.
- getClass(): returns the class that the object was instantiated from.
- hashCode(): returns a hashcode for the object, a unique ID
- toString(): returns a String message with the name of the class and some other numbers

Objects contain everything they inherit from each of their superclasses
Method overriding principles:

- Arguments must be the same, return types must be compatible
- The method in the subclass cannot be less accessible

7 Java Interface

Java Interface is a way of grouping classes that have different parent classes under the same umbrella and guarantees that they all have the same methods. However, every method in an interface will be abstract; each of its subclasses must implement the methods. An example of this is:

```
public interface Pet{
    public abstract void beFriendly();
    public abstract void play();
}
public class Dog extends Canine implements Pet {
    public void beFriendly() { ... }
    public void play() { ... }

    public void roam() { ... }
    public void eat(){ ... }
}
```

Classes from different inheritance trees can implement the same interface; conversely, a class can implement multiple interfaces.

8 Java Exception and IO

An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions. There are two types of exceptions, runtime exceptions and checked exceptions. Runtime exceptions occur during the Java runtime system, whereas checked ones occur during the control of the Java runtime system — the compile time. Whichever the type, an exception will be an object of type `Exception`. These can be handled with try–catch blocks:

```
try{
    //do risky thing
}
catch (Exception ex){
    //try to recover
}
finally{
    //code that will always run
}
```

You can also add a finally block: code that must run regardless of an exception. The compiler checks for everything except RuntimeExceptions; this guarantees that if you throw an exception in your code, you must declare it using the throws keyword in your method declaration and if you call a method that throws an

exception, you must acknowledge that you are aware of the exception.
Slides 8

```
Sample!
```

Specifier	Explanation
`private`	No other class can create instances of this class
`public`	Any class can create instances of this class
`protected`	Only subclasses can create instances of this class
default	Only classes within the same Package

Specifier	Explanation
`default`	Can be used in classes within same package
`public`	Can be used by any classes
`abstract`	This class cannot be instantiated
`final`	Cannot be inherited