Object Oriented Programming – I

Review-Part 1

MODULE – I: Basic Computer Structure

A computer is composed of wires and switches. Data is transmitted down wires and each wire can have two different values: on (has current) or off (no current). So a wire transmits two different values: 1 (on) or 0 (off).

- If we put 2 wires together, we get 4 different values: both off (00), both on (11), the first on and the second off (10), the first off and the second on (01).
- If we put 3 wires together, we get 8 different values: 000, 001, 010, 011, 100, 101, 110, 111. (Note that 8 = 2^3.)
- If we put 8 wires together, we get 2^8 = 256 different values.

ALU: The main processing in a computer is done in the ALU (Arithmetic-Logic Unit) that is a part of the CPU (Central Processing Unit). The ALU is made up of separate circuits to perform different tasks like addition, multiplication, etc. The ALU has three sets of input wires and one set of output wires.

- Two of the sets of input wires are for the data to the ALU
- The third input set of wires controls which operation the ALU should perform.

x-bit Machines

We sometimes refer to machines as x-bit machines. For example, your computer was probably advertised as a 64-bit machine (newer ones) or a 32-bit machine (older ones). The number refers to how many wires go into the computer circuits, and that means how much data the computer can process in one step. For example, in a 32-bit machine's each set of input wires to the ALU will consist of 32 wires and the output will also consist of 32 wires. 32 wires together is 32 bits of information, and that means each input can be one of 2^32 different values.

How a computer does a single operation:

The input wires for the ALU are turned on/off depending on the desired input values. The control wires for the ALU are turned on/off depending on the operation being done. As the electricity from these wires passes through the ALU circuits, different switches are activated to perform the calculation, and the resulting output appears as the output wires turn on/off appropriately. The control unit has a crystal based clock (similar to what runs a wrist watch), and when enough time passes for the calculation to complete, the control unit takes the output values and stores them into an appropriate location of memory. The speed of calculation is determined by how long it takes the electric current of the input/control wires to traverse the ALU and reach the output. Modern computers are so fast because these circuits are incredibly tiny.

Key idea to remember:

All data on a computer is transferred by a group of wires. Therefore all data (whether an integer, a fraction, a music file, etc.) is just a collection of 0's and 1's (is the wire off or on?)

MODULE – II: Data Types:

The type of a piece of data is what the piece of data represents. It is specified by the programmer and by rules of the programming language. Java is a "strongly typed" language. That means that every piece of data will have a well defined type and unambiguous type and the compiler will verify that every type is used correctly. The programmer must set the type of every expression (either explicitly or implicitly), and the Java compiler will verify that each type is used correctly before you can run your program. Java is very strict. If a type is used incorrectly, an error is given. For example, if you specify that a piece of data is English text, then Java will only allow you to do actions appropriate for text on it.

There are two kinds of types in Java:

1. primitive types: there are 8 primitive types, and they are all pre-defined in the Java language

2. compound types: there is a potentially infinite number of such types; many are pre-defined in Java, but programmers can make new ones as needed

Primitive Types in Java: The following are the eight primitive types:

int: 32-bits of data. Represents an integer between -2^31 and +2^31 – 1. All whole numbers in your program default to the type int.

double: 64-bits of data. Represents a "floating point" number (a number with a decimal point) in scientific notation. Roughly 52 bits for the mantissa, 11 bits for the exponent, 1 sign bit. All numbers with decimal points or in scientific notation in your program default to the type double. The largest/smallest values that can be represented are +/- 4.9 x 10^-324 and +/- 1.7 x 10^308.

char: 16-bits of data. Represents a character. char values are single characters inside single quotes. **example:** 'a', 'x', ' ', '?', '3' are all char values.

boolean: 1-bit of data (really stored as 8-bits). Represents either true or false.

short: 16-bits of data. Represents an integer between -2^15 and +2^15 - 1.

byte: 8-bits of data. Represents an integer between -2^7 and +2^7 - 1.

long: 64-bits of data. Represents an integer between -2^63 and +2^63 – 1. **example:** 10L, 300000L are all long values

float: 32-bits of data. Binary, scientific notation. Roughly 23 bits for the mantissa, 8 bits for the exponent, 1 sign bit. **example:** 3.14F is a float value

Typecasts (type conversions)

Java allows the programmer to convert a value of one type into a value of a different type.To do this, you place the desired type in parentheses and immediately to the left of the value.

(desired-type) value

For example:

- (double)3 converts the int value 3 to the double value that is the closest to 3. In this case, the value is 3.0.
- (short)100 converts the int value 100 to the short value that is the closts to 100. In this case, the value is still 100 (but stored in 16 bits instead of 32 bits).
- (int)3.8 coverts the double value 3.8 to the int value 3

The typecast must "make sense". In Java, you can convert between all the numeric primitive types, but not between boolean and a numeric primitive so:

(int)false ← error, you can't convert a boolean to a numbers

(int)'A' ← this is legal. Java considers a char to also be a number

Values are automatically converted "narrower" to "wider" types. A conversion from a "wider" to a "narrower" type requires an explicit typecast. Generally, when converting to a wider type, Java converts the value to be as close as possible to the original value. When converting to a narrower type, Java generally truncates the value.

Widest: double, float, long, int, short/char, byte : **Narrowest**

Note that short and char are at the "same" level, and so you must explicitly typecast between these two types. Also note that boolean is not in this list.

- **You can not convert a value of type boolean to any other primitive.**
- **You can not convert another primitive value to boolean.**

Introduction to Variables

A variable is the name given to a location in memory. A variable is used to store data. In Java, you can only store values with the appropriate type into the variable.

Creating Variables: We call creating a variable "declaring the variable"**.** A declaration has the form: type variable-name
Two examples:

```
int x
double temperature
```

Java now sets aside a chunk of memory for each varialble. The first is a 32-bit chunk that is associated with the name "x". Java will make sure that only values of type int can be stored in here. (Remember that the type is what WE decide the value represents. The computer does not care. It is happy storing any 32-bit values there.)

The second is a 64-bit chunk of memory that is associated with the name " temperature ", and we can store values of type long in it. Storing Values in Variables. We call storing a value "**assigning a variable**".

To store a value in the variable, you use the = operator.

```
variable = value
```

For example:

```
x = 161
```

temperature = 40.75

This is a source of confusion because the assignment operator looks like math's equality, but it is not an equality test. Instead, we are storing into the memory location named "x" the data that represents the int 161, and we are storing into the memory that is named "temperature" the data that represents the double value 40.75.

Remember that Java is a strong typed language so:

x = 3.1415 ← error, you promised x would store int, and you are giving it a double

x = (int)3.1415 ← this is now legal, and it stores value 3 into the location named "x"

To read the value stored in memory at a location, just use the name of the location:

x ← this evaluates to whatever value is stored in the memory location in "x", interpreted as data type int

int y

y = x + 50 ← this gets what is stored in x, adds 50 to it, and stores the result in the variable y

int y

y = 3.0 ← This is illegal because 3.0 is type double, double is wider than int, and we need an explicity typecast

y = (int)3.0 ← This is legal because the typecast is explicit.

y = 'A' ← This is legal because 'A' is type char, char is narrower than int, so the 'A' will be automatically converted to type int

MODULE – III: Primitive operations and their type rules:

Following are the type rules for the different primitive operators.

1. **arithmetic operators:** +, -, *, /, %
2. **unary arithmetic operators:** +, -
3. **binary operators**: &, |, ^, ~, <<, >>, >>>

   The above operators have the same type rules:

   a. The two operands (or one operand for the unary operators) must be a numeric primitive type (i.e. not Boolean)
   b. The narrower operand is automatically widened to match the wider operand, and if both operands are narrower than int, they are automatically widened to int.
   c. The result is the type of the wider operand, or int if both operands are narrower than int

Ex: 5.0 + 3 → (the result type is double because of the 5.0, before the addition is performed, the 3 is widened to 3.0)

'a' + 'b' → (the result type is int and both operands are widened to int before the addition is performed)

(short)3 + (byte)5 (the result is type int, and both operands are widened to int before the operation is performed)

4. **comparison operators:** >, <, >=, <=

The type rules:

a. The operands must be a numeric primitive (a non-boolean primitive)
b. The result type is Boolean
c. The narrower operand type is automatically widened before the operation is performed

Ex: 3 <= 3.1 ← Converts the int 3 to double 3.0 and then compares 3.0 to 3.1

3 < 4 < 5 ← Illegal. First computes 3 < 4, but then the left operand of the second < is type boolean!

5- **boolean operators:** &&, ||, !, &, | (these are AND, OR, NOT, AND, and OR)

The type rules:

a. The operands (or operand for !) must be Boolean
b. The result is Boolean
c. && and || use "short-circuit evaluation". If the result value is known from just the left operand, the right operand is not evaluated:

6- **equality operators:** ==, !=

The type rules:

a- The operands can be any type (primitive or non-primitive, and the narrower operand is automatically widened to the wider operand's type.

- it is an error if the automatic type conversion is not allowed

b- The result is Boolean
c- The value is determined by comparing the boolean values of the two operands (after the automatic widening).

Ex: 5 == 5.0 → first converts 5 to 5.0 before comparing

1 != true → illegal because you can't convert between int and Boolean

false != true → legal because both operands are the same type

7- **assignment operator:** =

a. The left operand must evaluate to a variable. The variable can be any type (primitive or non-primitive).
b. The type of the right operand must be the same or narrower as the type of the variable on the left (*)
c. The result is the same type as the variable and the value that was stored in the variable.

Because = has a value and a type, we can place assignment operators anywhere in our code that expects a value.

Ex: double x
x = 1 ← legal, int 1 is widened to double 1.0 and stored. The result is type double and value 1.0.
int y
y = 1.0 ← illegal. 1.0 is wider than int
x = y = 3 ← legal. the y = 3 happens first. The result is the int value 3, now we get x = 3, and the 3 is widened to 3.0
The result of this expression will have 3 stored in y, 3.0 stored in x, and the expression will evaluate to the double 3.0

MODULE – IV: Java Non-Primitive Types (Reference/Compound):

The basic compound type is called a "class". A class consists of

1) variables of any type (Java calls variables inside of classes "fields")
2) functions that take 0 or more input, possibly performs some action, and returns 0 or 1 value (Java calls these functions "methods")
3) other reference/compound types (Java calls these functions "nested types")

There are a couple other kinds of reference/compound types that will be covered later. We will now give examples of 2 pre-defined classes:

Math: a collection of mathematical functions and constants

JFrame: represents a window on your computer screen

JFrame and Math are different kinds of classes. JFrame is a "normal" class in that you can create instances of it. Math on the other hand, is more limited in that you can't create an instance. (Formally, the Math class has a private constructor. You will see what that means soon.)

Operators for compound/reference types.

Here are all the operators you can do on a compound/reference type. The first and second are the same as for primitive values:

1) The equality operators: ==, !=
2) The assignment operator: =
3) new (used to create an instance/value of a type)
4) . (used to access the fields, methods, or nested types of the class)
5) instanceof (used to test what type a value is)

Terminology: A value of a nonprimitive type is an instance or an object. A class instance is called an "object" in Java.

Important: beginning programmers often struggle with the difference between a type and an instance of the type.

For example, if the type is Movie, instances of the type are "Get Out", "Star Wars: The Last Jedi", "The Devil Wears Prada"

Each instance refers to a specific, separate movie while the type Movie refers to the entire concept of what a film is.

Ex:

type	Instance of the type
int	-3, 5, 100
JFrame	All of the windows on your desktop
Movie	"Pilp Fiction", "Lord of the Rings"

What can we do with classes?

1- Create an instance of the class
2- Execute a method of a class
3- Access a field of a class

Using Classes:

1- To create an instance of a class, use the new operator

new desired-type(0 or more input values)

new JFrame() → this results in a value of type JFrame

→ note that to use JFrame, you must "import" the package.

The new operator does the following:

- Allocate space in memory for the type instance.
- Initializes the instance using the input data (it does this by calling a special method/function, called a constructor, whose sole purpose is initializing instances)
- Returns the location in memory of the instance. This location is called the "reference" of the instance.

Variables and Assignment with Reference Types:

The rules are EXACTLY the same as for primitive types

double x ← creates a variable called x that stores values of type double

JFrame frame1 ← creates a variable called window that stores values of type JFrame

x = 5.0 ← stores the value 5.0 in variable x.

frame1 = new JFrame() ← stores the location of the JFrame instance in variable window.

To access something from inside a compound type, we use the "dot" operator.

2- Calling/executing methods of a class

There are two types of methods: instance methods (non-static) and class methods (static).

a. An instance method "acts" on an instance of the class.

To call (execute) an instance method you use:

instance-location.method-name(0 or more inputs separated by commas)

For example, JFrame has an instance method setVisible that takes a single boolean as input. To call the method on the instance whose address is stored in frame1, we use:

frame1.setVisible(true) or frame1.setVisible(false)

Note that the expression to the left of the dot must be an address. It does not have to be a variable. Any expression that gives an address of type JFrame is okay:

new JFrame().setVisible(true)

For another example, JFrame has a method setSize that takes two int values as input. To call the method on the instance whose address is stored in frame1:

frame1.setSize(300, 500)

b. A class method "acts" on the class as a whole and not instances of the class.

To call (execute) a class method you use either:
class-name.method-name(0 or more inputs)
instance-location.method-name(0 or more inputs)
While you can use either, we prefer the first. That make it obvious that you are using a class method. Here are some examples:

Math.cos(1.0) ← the cos method "operates" on the entire Math type.Math.sqrt(5.0)

3- Accessing fields

There are two kinds of fields: instance fields and class fields.

a) **Instance fields:** an instance field as a variable that belongs to an instance of the class.

As a result, each separate instance has its own version of the field. To access an instance field, you use

instance-location.field

Example: JFrame has a field (variable) of type boolean called rootPaneCheckingEnabled

boolean z = frame1.rootPaneCheckingEnabled

frame1.rootPaneCheckingEnabled = true

You can think of the . as an "apostrophe s". The expression frame1.rootPane in Java is asking for frame1's rootPane variable.

b) **Class fields:** a class field is a variable that belongs to the class as a whole.

As a result, there is a single copy of the variable that all instances of the class share.

To access a class field, you use either

class-name.field

instance-location.field

As with class methods, we prefer that you use class-name.field when accessing a class field to make it obvious.

Example, the Math class has a class field called PI that stores the value of PI.

```
double x = Math.PI
```

Class fields are also called "static" fields and class methods are also called "static" methods.

This is not a really good name because it sounds like a class field can't be changed. It can.

A field that cannot be changed is called a "final" field. So Math.PI is a "static final" field. "static" because it belongs to the class and "final" because the value of the field cannot be changed.

Important Terms to Remember

Here are some common terms you should get used to hearing and using.

1. **non-primitive type:** any type in Java that is not one of the 8 primitives types.
2. **reference type:** another name for a non-primitive type. The name "reference type" reminds us that the value of these types is not the data itself, but it is the location in memory of where the data for the value is stored.
3. **compound type:** another name for a non-primitive type. The name reminds us that a non-primitive type is made up of many other types, variables, and functions.
4. **class:** the most common type of non-primitive type in Java
5. **instance:** a value whose type is a class, or any non-primitive type
6. **object:** another name for an instance. A value whose type is a class, or any non-primitive type.
7. **field:** a variable that is part of a non-primitive type.
8. **method:** a function that is part of a non-primitive type. (In Java, all functions are methods.)
9. **instance field:** a field that belongs to an instance of a type.
10. **class field:** a field that belongs to a non-primitive type (not necessarily a class), and not to the instances of that type.
11. **static field:** another name for a class field.
12. **instance method:** a method that operates on a specific instance of a non-primitive type.
13. **class method:** a method that does not operate on specific instances of a non-primitive type, but on the type itself (not necessarily a class).
14. **static method:** another name for a class method.

MODULE – V: Java Programming:

1) A simple Java statement ends with a ;
2) A compound Java statement is bracketed by { } and contains 0 or more statements (simple or compound).

3) All Java programming consists of writing compound/reference types. **The basic reference type is the class.** A class definition consists of a header followed by a compound statement.

```
public class MyFirstClass extends Object {

}
```

- The part starts with "public" is called the "class header".
- The part between the { } is called the "class body". The class body is by definition a compound statement, and all elements of the class (fields, method, other compound types) go in the body (between the { }).

The parts are:

- **public:** the access modifier
- **class:** this is a class
- **MyFirstClass:** the name of the class (a name can be almost anything, but professional Java style is to always start with a capital letter)
- **extends Object:** indicates the super class of this class. Every class has exactly 1 super class (also called the "**parent**" class).

NOTE: The access modifier determines where the code can be directly accessed. (This is not a security feature. All code can still be indirectly accessed.)

The access modifier can be any of the four:

- **public:** the code can be used/accessed anywhere in the program
- **protected:** the code can be used in this type and in any type that extends this type (or in any type in the same folder as this type)
- **private:** the code can only be used in the body of this type

If you omit the access modifier, the default is "**package**": the code can be used in any type that is in the same package (i.e. in the same folder) as the containing type.

What can go in a class body?

- field declarations (these can include assignment operators)
- methods
- other non-primitive type definitions (these are called inner types or nested types)
- **constructors**: special methods used to initialize instances of the class

4) Each public class must go in its own file. The file name must be the same as the class name, and the file extension must be .java. The class must be compiled before you use it. The compiler creates a file with the same name but .class extension that contains the Java bytecode for the class.

5) What's the point of a super class? (More about this later)

- Every class "inherits" all public and protected instance methods of the super classç
- Every class has access to all public and protected constructors, fields, class (i.e. static) methods, and nested types of the super class.

Fields:

There are two kinds of fields.

- Instance fields are memory that is allocated inside each instance. Thus, every instance has its own copy of an instance field.
- Class fields are memory that is allocated outside of the instances. There is only one copy of a class field that all the instances share.

To create an instance field is the same as declaring a variable except you add an access modifier:

access-modifier type name

To create a class field, you do the same but add the word "static" before the variable type:

```
public class MyFirstClass extends Object {
      private int myInstanceField;
      private static int myClassField;
  }
```

Every instance gets its own copy of the instance fields. (Note you will have to change the access modifiers to public if you want to test this in the interactions pane.)

```
MyFirstClass c1 = new MyFirstClass();
c1.myInstanceField = 5;
MyFirstClass c2 = new MyFirstClass();
c2.myInstanceField = 6;
c1.myInstanceField   ← returns 5
c2.myInstanceField   ← returns 6
  -- these are two different variables with the same name stored inside two different objects
c1.myClassField = 10;
c2.myClassField = 20;
c1.myClassField    ← returns 20 because there is only 1 copy of this variable.
MyFirstClass.myClassField   ← also returns 20.  Since the field belongs to the class, you can use the classname to access it
```

Methods:

An instance method definition is a method header followed by a compound statement.

access-modifier return-type name(input parameters) {

}

Here is an example that takes two inputs of type int and returns a value of type int

```
public int myMethod(int input1, int input2) {
```

A class method uses the same declaration but adds "static" before the return type:

```
public static int myClassMethod(int input1, int input2) {
```

(An instance method "operates" on an instance while a class method does not. More on this very soon!)

- the input parameters are a sequence of 0 or more variable declarations separated by commas. There is one variable declaration for each input your method will take.
- the return type can be any type, or if the method will not return a value, it is "void".
- the part starting with the access modifier is called the "method header"
- the part between the { } is called the "method body".
- all code describing the behavior of the method goes in the body. The body is by definition a compound statement.

```
public class MyFirstClass extends Object {
        public int average(int input1, int input2) {
                int sum = input1 + input 2;
                int avg = sum / 2;
                return sum;
         }
}
```

A return statement must be included in any method that has a non-void return type. The return statement gives the output of the method.

Now we can use this method:

> MyFirstClass c = new MyFirstClass();

> c.average(300,500) ← 400

When you call the method, Java assigns the first value to the first variable of the input parameters and the second value to the second variable. **All type rules apply!** So we have to use two values that are type int (or whose type Java will automatically convert to int).

We also have to give two values as input.

> c.average(300) ← Error no method with input (int), only (int, int)

What is the point of the super class? Your class inherits all the public and protected instance methods of the super class:

```
public class MyFirstFrame extends JFrame {

}
```

→ Now MyFirstFrame inherits all the methods of JFrame, and so we can use them:

```
MyFirstFrame f = new MyFirstFrame();

f.setVisible(true);

f.setSize(300,500);
```