Notas del Diplomado de Java - Módulo I

- Conceptos Fundamentales -

Variables y tipos de datos

- **Variables** - Almacenan valores de un tipo de dato.

```
short a = 127; // 16 bits
int b = 123; // 32 bits
long c = 12346753; // 64 bits
float d = 123.15f; // 32 bits
double e = 123.15674324677; // 64 bits
char f = '@';
String g = "Hola java :)";
boolean h = true;
```

Operaciones Lógicas y Artiméticas

> Operaciones lógicas

```
boolean P = true;
boolean Q = false;

boolean S = P && Q;
boolean R = P || Q;
boolean nP = !P;
boolean T = (P || Q) && (S || R) || nP;

boolean t1 = 1 == 2;
boolean t2 = 1 != 2;
boolean t3 = 1 > 5;
boolean t4 = 1 >= 5;
boolean t5 = 1 < 5;
boolean t6 = 1 <= 5;
```

> Operaciones aritméticas

Entre números podemos usar `+ - * / %` recordando que la expresión final tendrá el tipo de dato del tipo más amplio (double, float, int). Nota: Cuidado al dividir dos enteros ya que la división será entera, intente utilizar un `casting` : `((float)a / b)` .

Estructuras de control

Podemos cambiar el flujo de un programa mediante las estructuras de control.

> *if* - Ejecuta un bloque si la condición se cumple, opcionalmente se puede anidar en `else-if` y `else`

```
if (edad < 18) {
    //
} else if (edad < 60) {
    //
} else {
    //
}
```

> Existe un operador llamado `operador condicional` que sirve para asignar una variable mediante una expresión condicional.

```
int a = 15;
int b = 12;

int c = a > b ? a : b;

// condición ? expr_v : expr_f
```

> *switch* - Compara un valor en casos y ejecuta el bloque para el caso que se cumpla. Nota: Siempre utiliza `break` a menos que necesites anidar casos.

```
switch(opcion) {
    case 1:
        //
        break;
    case 2:
    case 3:
        //
        break;
    default:
        //
        break;
}
```

> *for* - Crea un iterador sobre una variable iterable.

```
for (int i = 0; i < 10; i++) {
    //
}

String[] nombres = { "Paco", "Diego", "Luis", "Ana" };

for (String nombre : nombres) {
    //
}
```

> *while/do-while* - Repite un bloque mientras la condición se cumpla, `do-while` ejecuta primero el bloque y luego comprueba la condición, si se cumple el bloque se repite.

```
Scanner sc = new Scanner(System.in);

int n;

do {
    System.out.print("Dame un número [0-100]: ");
    n = sc.nextInt();
} while(n >= 0 && n <= 100);
```

Formato de cadena

Es cuando a una cadena con un patrón le reemplazamos el patrón por los valores de variables con su tipo de

dato correspondiente.

```
String nombre = "Ana B.";
int edad = 12;
float estatura = 1.70f;

System.out.printf("%s (%d) [%.2f]", nombre, edad, estatura);

String datos = String.format("%s (%d) [%.2f]", nombre, edad, estatura);

System.out.println(datos);
```

Clases

Es la abstracción de un sistema o parte del sistema mediante atributos y métodos que desciban a objetos. Las clases tiene la finalidad de definir un grupo de métodos que resuelvan tareas específicas y que compartan las mismas variables.

```
class Persona {

    String nombre;
    int edad;
    float estatura;

    void saludar() {
        System.out.printf("Hola me llamo %s tengo %d años " +
            " y mido %.2f metros.", nombre, edad, estatura);
    }

}
```

Objetos

Son contextos independientes de atributos y métodos generados por una clase, esto llevado a memoria se conoce como *instancia*. Los objetos nos permiten implementar las clases en programas con sus propios valores internos y estos se podrían modificar mediante métodos. Los objetos tiene la finalidad de resolver tareas mediante sus métodos definidos, generalizando los atributos que require cada tarea, es decir, cada objeto tiene sus propios atributos.

```
Persona persona = new Persona();

persona.saludar();
```

Constructores

Es método especial que se llama igual que la clase el cual no devuelve ningún tipo de dato. Sirve para inicializar los atributos del objeto. Siempre debemos pedir el mínimo de atributos en el constructor.

```
class Persona {

    String nombre;
    int edad;
    float estatura;
    int anio_nacimiento;

    Persona(String nombre, int edad, float estatura) {
        this.nombre = nombre;
        this.edad = edad;
        this.estatura = estatura;
        this.anio_nacimiento = 2017 - edad;
    }

}

//... En algún lado
Persona persona = new Persona("Pepe", 43, 1.87f);
```

Sobrecarga (Polimorfismo I)

Es definir varios métodos en la clase con el mismo nombre, pero con distos parámetros, a esto se le llama la *nomenglatura* y está dada por el nombre del método y el tipo de dato de cada parámetro.

```java
class Persona {

    // ...

    // saludar()
    void saludar() {
        System.out.printf("Hola soy %s", nombre);
    }

    // saludar(String)
    void saludar(String frase) {
        System.out.printf("%s %s", frase, nombre);
    }

}
```

Métodos de Acceso (Encapsulamiento I)

Son las palabras `public`, `private` y `protected` que anteceden un atributo o método y le dan la cualidad de por poder ser accedido fuera del objeto, dentro de una clase hija o sólo dentro de la clase.

```java
class A {
    public int x; // Clase, Hijos, Fuera
    protected int z; // Clase, Hijos
    private int y; // Clase
}

A a = new A();

a.x = 12; // SI
a.y = 12; // NO
a.z = 12; // NO
```

Herencia (Encapsulamiento II)

Extender o modificar una clase y a la nueva clase derivada se le conoce como la *clase hija*. La clase hija tiene todos sus atributos y métodos, pero igualmente puede seguir definiendo más o reemplazar algún método de la clase padre.

```
class A {
    int x;
    void foo() {
        //...
    }
}

class B extends A {
    int y;
    void bar() {
        //...
    }
}

class C extends B {
    int z;
    @Overrive
    void foo() {
        //...
    }
}

// ...

A a = new A(); // a.x, a.foo()
B b = new B(); // b.x, b.y, b.foo(), b.bar()
C c = new C(); // c.x, c.y, c.z, c.foo()*, c.bar()
```

Clases Abstractas

Es una clase tradicional, pero con la restricción que no podemos crear instancias de esa clase. Sirve para generalizar un prototipo de qué métodos y atributos deberían tener las clases hijas, y hacer notar que es importante quizas necesitemos reescribir algún método.

```
abstract class StreamCopy {
    void copy(InputStream in, OutputStream out) {
        throw new Exception("No se ha implementado este método");
    }
}
```

```
class BasicStreamCopy extends StreamCopy {
    @Override
    void copy(InputStream in, OutputStream out) {
        int b;
        while((b = in.read()) != -1) {
            out.write(b);
        }
    })
}
```

```
class BufferStreamCopy extends StreamCopy {
    @Override
    void copy(InputStream in, OutputStream out) {
        BufferedInputStream bin = new BufferdInputStream(in);
        BufferedOutputStream bout = new BufferdOutputStream(out);
        int b;
        while((b = bin.read()) != -1) {
            bout.write(b);
        }
    })
}
```

```
//StreamCopy s = new StreamCopy(); // ERROR
StreamCopy s = new BasicStreamCopy();

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

Interfaces (Polimorfismo II)

Nos permiten crear conjuntos de funcionales, es decir, definen `"instancias"` que sólo poseen una parte funcional de quién implementa la interfaz. Cuando un conjunto de métodos resuelven una tarea específica que puede ser realizada de distintas formas, podemos crear una interfaz que abreaíga dichos métodos y en lugar de requerir cierta clase en específico, ahora sólo requeriremos su parte funcional.

> IStreamSave.java

```
interface IStreamSave {
    boolean save(InputStream in);
}
```

> FileStreamSave.java

```
class FileStreamSave implements IStreamSave {
    public boolean save(InputStream in) {
        FileOutputStream out = null;
        try {
            out = new FileOutputStream("/temp/x");
            int b;
            while((b = in.read()) != -1) {
                out.write(b);
            }
        } catch(Exception e) {
            return false;
        } finally {
            if (out != null) {
                out.close();
            }
        }
        return true;
    }
}
```

> FakeStreamSave.java

```
class FakeStreamSave implements IStreamSave {
    public boolean save(InputStream in) {
        return true;
    }
}
```

```
InputStream in = ...;
IStreamSave fs_1 = new FileStreamSave();
fs_1.save(in);
IStreamSave fs_2 = new FakeStreamSave();
fs_2.save(in);
```

Flujos de datos (*Streams*)

Son secuencias de bytes procesadas a tráves de distintas clases que extraen dichos flujos de diversas partes, desde un archivo, desde una url, desde un socket, entre otros. Estos pueden ser de entrada (lectura) o de

salida (escritura).

> Proveniente de un archivo

```
InputStream in = new FileInputStream("C:/test/data.x");

int b;

while ((b = in.read()) != -1) {
    // ...
}
```

> Proveniente desde una URL

```
URL url = new URL("http://host/path");

InputStream in = url.openStream();

int b;

while ((b = in.read()) != -1) {
    // ...
}
```

Investiga el concepto de `Wrapper` para reconvertir los flujos de datos por ejemplo en `BufferedInputStream`.

Archivos

Son flujos de datos que en forma física terminan siendo arreglos de bytes. Existen clases para manipular los flujos de datos provenientes de un archivo binario, un archivo de texto, un archivo de imagen, entre otros.

> `FileInputStream`, `FileOutputStream`, `FileReader`, `FileWriter`

Colecciones (ArrayList, HashSet, HashMap)

En Java existen distintos tipos de colecciones ya programados para por ejemplo, crear listas no fijas de elementos, quitar elementos duplicados mediante conjuntos (`sets`), crear almacenes `clave-valor` mediante mapas (`maps`).

> List / ArrayList

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

frutas.add("Piña");
frutas.add("Manzana");
frutas.add("Pera");

frutas.remove("Manzana");
```

> Set / HashSet

```
Set<String> ufrutas = new HashSet(frutas);

ufrutas.add("Manzana");
ufrutas.add("Mango");
```

> Map / HashMap

```
Map<String, Product> productos = new HashMap<String, Product>();

productos.put("A179956", new Product("Coca Cola"));
productos.put("B792313", new Product("Galletas Marías"));

Product producto = productos.get("F9985788");
```

Hilos

Es un proceso que se ejecuta de forma independiente y sirven para paralelizar tareas estructuradas mediante métodos y clases.

Supongamos que tenemos la clase *A* que resuelve una tarea mediante los métodos *foo* y *bar*. Para proveer a la clase *A* la posibilidad de ejecutarse en un hilo, tenemos que implementar la interfaz `Runnable`, esta interfaz expone un método llamado `public void run()`, al implementar el método en la clase, cuando los objetos de la clase son envueltos en un hilo, al iniciar el hilo se ejecutará el método `run` realizando todas las tareas que se hayan programado ahí. Por ejemplo, dentro de *run* podemos llamar a los métodos `foo` y `bar` o realizar tareas comunes como creación de variables y flujos.

Si nosotros llamamos manualmente al método `run`, este se ejecuta en hilo principal del programa,

bloqueando el hilo principal hasta que termine:

```
A a = new A();

a.run(); // Tarda 20 segundos en completar su tarea

System.out.println(); // Se ejecuta tras 20 segundos
```

Usando hilos, podemos programar un hilo independiente al principal las clases que hayan implementado `Runnable`.

```
A a = new A();

Thread t = new Thread(a);

t.start(); // Comienza a ejecutar el hilo.

System.out.println(); // Se ejecuta inmediatamente.
```