Programación Interactiva
Fundamentos de Java

Escuela de Ingeniería de Sistemas y Computación
Facultad de Ingeniería
Universidad del Valle

Ejecución de un programa

- Para Java una clase ejecutable es aquella que es pública y tiene un método main()
- Todos los archivos con código Java deben tener la extensión .java
- En cada archivo .java puede haber máximo una clase pública, y esta debe coincidir con el nombre del archivo (Java distingue entre mayúsculas y minúsculas)
- Si se desea las clases pueden agruparse en paquetes (packages), esto es recomendable para aplicaciones grandes

Contenido de un archivo Java

- En cada archivo .java pueden haber varias clases, cada una puede tener métodos (funciones de clase) y atributos (variables de clase)

- La programación orientada a objetos se basa en la organización de la solución al problema en clases

- En la cabecera de un archivo .java pueden colocarse sentencias import, usadas para hacer visibles dentro del archivo las clases que hay en una librería

Variables en Java

- En Java se distinguen 2 tipos de variables:
  - ***Tipo de datos primitivos*** como *double*, *int*, *boolean*, entre otros. Cuando se le pasan a un método como parámetro se pasan por valor.
  - ***Referencias a objetos*** como los *String*, *Object*, *Mascota*, *int[]*, entre muchos otros. Cuando se pasan a un método como parámetro se pasan por valor (si, por valor), pero como son referencias a objetos, puede verse como que se está pasando el objeto por referencia.

Variables en Java

- Estas variables pueden ser variables locales a un método, o locales a una clase (***atributos*** de la clase

- Las variables deben ser declaradas antes de poder usarse (tal como en C++), y su ámbito de visibilidad (scope) se limita al bloque en el que fueron declaradas.

- Los nombres de las variables pueden contener ñ y letras con tildes.

Variables en Java

- Si un atributo no es inicializado cuando se declara la clase, este toma el valor por defecto (esto no aplica cuando es una variable local a una función).

- Los valores por defecto dependen del tipo de atributo:

  false para los ***boolean***

  0 para los atributos de tipo numérico (***int***,***double***,***float***,etc)

  null para las referencias a objeto

- Una referencia null es una referencia que no está “apuntando” a ningún objeto.

Modelo Clases-Objetos

- Una ***clase*** es como una plantilla a partir de la cual se crean ***objetos***, es decir, los objetos son ***instancias*** de una clase. Pueden crearse cuantos objetos se quiera a partir de una clase.

- Las variables que tiene cada objeto es independiente de las variables de los demás objetos de la misma clase, es decir, pueden tener distinto valor (a menos que se declaren como ***static***)

- De una clase abstracta no se pueden crear objetos

- Los objetos son creados mediante la instrucción ***new***, y a diferencia de C++, no hay necesidad de liberar la memoria ocupad, el GC se encarga de hacerlo cuando sea necesario.

Modelo Clases-Objetos

- Clase que representa una cuenta bancaria:

```
public class CuentaBancaria {
     protected double saldo;
     protected String nombreCuenta;

     public CuentaBancaria(String nombreCuenta)
     {
          this.nombreCuenta = nombreCuenta;
     }
     public double getSaldo() {
          return saldo;
     }
     public void debitarCuenta(double valor) {
          saldo -= valor;
     }
     public void acreditarCuenta(double valor) {
          saldo += valor;
     }
     public String getNombreCuenta(double valor) {
          Return nombreCuenta;
     }
     }
```

Herencia

- Una clase puede servir como plantilla para otras clases, es decir que los métodos y atributos declarados en una clase A pueden ser introducidos en una clase B con solo especificar que B ***extiende*** de A.

- A la clase B se le llamaría clase padre, o ***superclase***, y a la clase B se le llamaría clase hija, o ***subclase***

- Por ejemplo, podemos definir CuentaAhorros y CuentaCorriente como subclases que extienden de CuentaBancaria, ya que esta última agrupa las características comunes de ambos tipos de cuenta

Herencia

```
public class CuentaCorriente extends CuentaBancaria {
    protected int numeroCheques;
    protected int numeroChequesUtilizados;

    public int getNumeroChequesRestantes() {
        return numeroCheques - numeroChequesUtilizados;
    }
}
```

- Debido a que CuentaCorriente extiende de CuetnaBancaria, todos los métodos de esta última son heredados por CuentaCorriente. Así que las siguientes líneas serían válidas:

```
CuentaCorriente ctaCte = new CuentaCorriente();
double saldo = ctaCte.getSaldo();
```

Herencia

- Veamos otro concepto de la programación orientada a objetos relacionado con la *herencia:*

```
public class CuentaAhorros extends CuentaBancaria {
    public double totalInteresesMes;
    public int getTotalInteresesMes() {
        return totalInteresesMes;
    }
    public double getSaldo() {
        return saldo + totalInteresesMes;
    }
}
```

- Como puede verse, la clase CuentaAhorros está definiendo el método getSaldo, que ya estaba definido en CuentaBancaria.

- A esto es lo que se le llama *sobrecarga de métodos*

Herencia: Resumen

- La herencia permite que se puedan definir nuevas clases basadas en clases existentes, lo cual facilita reutilizar código previamente desarrollado.
- Si una clase deriva de otra (*extends*) hereda todas sus variables y métodos
- La clase derivada puede *añadir* nuevas variables y métodos y/o *redefinir* las variables y métodos heredados.

Tomado de las Clases de Simena Dinas

Interfases

- Una interfaz es, al igual que una clase, una plantilla de la cual otras clases “heredan”. Pero a diferencia de las clases, estas no pueden tener atributos variables (solo constantes), ni métodos implementados (sólo se puede hacer la declaración de métodos)

```
public interface Sobregirable {
      public double getSobregiro();
      public void setSobregiro(double nuevoSobregiro)
}
```

Interfases

- Para hacer que una clase implemente una interfaz se debe usar la palabra ***implements***, y como la interfaz solo define la existencia de los métodos, pero no lo que hacen los métodos, debemos proveer esta implementación dentro de la clase (a menos que la declaremos como *abstracta*):

```
public class CuentaCorriente
     extends CuentaBancaria implements Sobregirable
{
     protected double sobregiro;
     protected int numeroCheques;
     protected int numeroChequesUtilizados;

     public int getNumeroChequesRestantes() {
          return numeroCheques - numeroChequesUtilizados;
     }
     public double getSobregiro() {
          return sobregiro;
     }
     public void setSobregiro(double nuevoSobregiro) {
          sobregiro = nuevoSobregiro;
     }
     }
```

Interfases

- Como una interfaz no tiene declarados sus métodos, esta no se puede instanciar (al igual que las clases abstractas).

- Cuando una clase hereda de una clase abstracta, o implementa una interfaz, debe implementar todos los métodos que no posean implementación.

- **Los interfases NO proporcionan Herencia Múltiple**

- Algunas veces se trata a los interfaces como una alternativa a la herencia múltiple en las clases. A pesar de que los interfaces podrían resolver algunos problemas de la herencia múltiple:
  - No se pueden heredar variables desde un interfase.
  - No se pueden heredar implementaciones de métodos desde un interfase.

Herencia de Interfases

```
public interface X {
        public final int constante1, constante2;
        public void metodo1();
        public double metodo2(int atributo);
}
```

SuperInterfaz

```
public interface Y extends X {
        double metodo3();
}
```

SubInterfaz

Herencia Múltiple ?

- Herencia múltiple significa que una clase puede heredar (***extends***) de varias clases a la vez.
- En Java ***NO*** es posible la herencia múltiple ***de clases***, pero si es posible la herencia múltiple de ***interfaces***.
- La herencia múltiple de interfaces ***NO*** reemplaza totalmente la herencia múltiple de clases

Herencia Múltiple ?

- Siguiendo con los ejemplos anteriores, en los que X y Y son interfases y A y B son clases:

```
public class C extends A implements X {
}
```

ES VALIDO

```
public class D extends A,B implements X {
}
```

NO VALIDO!

```
public class E extends A implements X, Y {
}
```

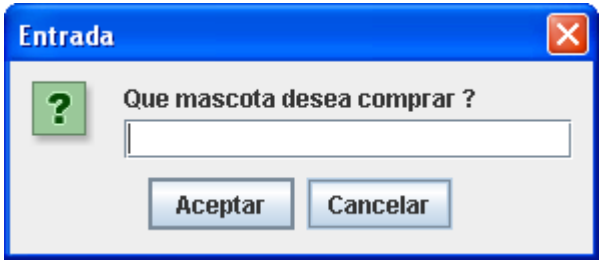
ES VALIDO?

Nuestro primer programa real

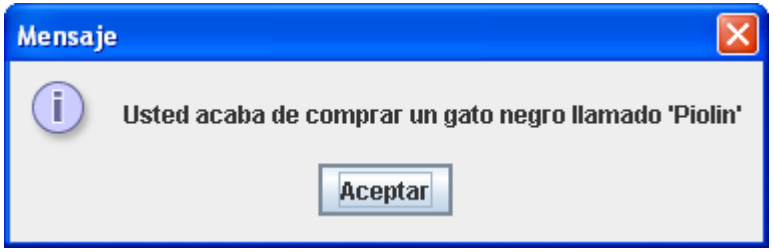
```
public class Banco {
   public static void main(String[] args) {
     CuentaBancaria cuenta;
     String nombreCuenta, tipoCuenta;

     tipoCuenta = JOptionPane.showInputDialog("Escriba el tipo de cuenta:");
     nombreCuenta = JOptionPane.showInputDialog("Que nombre desea darle a su cuenta?");

     if (tipoCuenta.toUpperCase().equals("CORRIENTE")) {
          cuenta = new CuentaCorriente(nombreCuenta);
     }
     else if (tipoCuenta.toUpperCase().equals("AHORROS")) {
          cuenta = new CuentaAhorros(nombreCuenta);
     }
     else {
          // Si no es ninguno de los tipos válidos de cuentas:
          cuenta = new CuentaBancaria(nombreCuenta);
     }

     JOptionPane.showMessageDialog(null, "Usted acaba de crear una "+
          cuenta.getSimpleName()+'");
   }
}
```

Nuestro primer programa real

- JOptionPane.showInputDialog
  Muestra un cuadro de dialogo donde podemos escribir.



- JOptionPane.showMessageDialog
  Muestra un cuadro de dialogo con cierta información



Visibilidad de métodos y atributos

- Los métodos o atributos declarados como ***public*** pueden ser accedidos desde cualquier otra clase.
- Los métodos o atributos declarados como ***protected*** pueden ser accedidos desde la misma clase, y sus subclases; pero no desde otras clases.
- Los métodos o atributos declarados como ***private*** solo pueden ser accedidos desde la misma clase donde se están declarando, no desde otras clases ni desde sus subclases.
- En Java ***NO*** hay equivalente al modificador de visibilidad ***friend*** que hay en C++

this

- Dentro de una clase se puede acceder a sus atributos y métodos mediante la palabra reservada ***this***, que actúa como una referencia al objeto que realiza la invocación.

- Cuando sería necesario usarla?

```
public class A
{
        int variable;
        int f (int variable)
        {
                return variable*2;
        }
}
```

new

- El operador ***new*** es usado para crear nuevos objetos a partir de una clase:

```
CuentaAhorros miMascota = new CuentaAhorros();
```

- No hay equivalente al operador ***delete*** de ***C++***, ya que el ***GC*** de Java “sabe” cuando debe liberar la memoria ocupada por un objeto.

static

- Recordemos que un atributo de un objeto de la clase A puede tener un valor diferente de otro objeto de la clase B:

```
public class A {
        int v;
}
public class B {
     public static void main (String[] args) {
        A objeto1 = new A();
        A objeto2 = new A();
        objeto1.v = 3;
        Objeto2.v = 5;
        System.out.println("objeto1.v= "+ objeto1.v);
        System.out.println("objeto2.v= "+ objeto2.v);
        }
}
```

- Imprimiría en la consola:

```
objeto1.v= 3
objeto2.v= 5
```

static

- Pero si al atributo ***v*** lo declaramos como ***static***:

```
public class A {
        static int v1;
}
public class B {
     public static void main (String[] args) {
       A objeto1 = new A();
       A objeto2 = new A();
       objeto1.v = 3;
       Objeto2.v = 5;
       System.out.println("objeto1.v= "+ objeto1.v);
       System.out.println("objeto2.v= "+ objeto2.v);
       }
}
```

- Imprimiría en la consola:

```
objeto1.v= 5
objeto2.v= 5
```

static

- Debido a que ***v*** es un atributo de la clase ***A***, y no de cada uno de sus objetos. Esto es lo que se logra con el modificador ***static***.

- Con ***static*** todos los objetos de la clase ***A*** comparten la misma variable ***v*** y la modificación de ***v*** desde algún objeto de esta clase, se reflejará en los demás objetos.

- Para que puede ser útil el modificador ***static***? Como contador de objetos creados, por ejemplo.

- No hay necesidad de declarar un objeto para acceder a una variable estática, es suficiente con el nombre de la clase

static

- En Java ***NO*** hay variables locales estáticas a diferencia de ***C++***

- Continuando con el ejemplo anterior:

```
public class A {
        static int v;
}
public class B {
     public static void main (String[] args) {
        A objeto1 = new A();
        A objeto2 = new A();
        objeto1.v = 3;
        ...
        }
}
```

- La línea señalada es equivalente a

```
objeto2.v = 3;
```

y a

```
A.v = 3;
```

final

- El modificador final se usa para indicar que una referencia o atributo es constante, es decir, que su valor no puede ser cambiado.

- Por convención, se usan solo mayúsculas para nombrar las variables final.

```
public final int PI = 3.1415;
```

- Esta línea no sería legal:

```
PI = 2; // no se puede, PI es constante
```

final

- También puede declararse una clase como final, con el objetivo de que no se puedan crear subclases a partir de ella

- Un método declarado como final no puede ser sobrecargado (***override***) por una subclase

Clases abstractas

- Una clase abstracta es una clase que sirve de plantilla para otras clases (para que la hereden), pero de la cual no se pueden crear objetos.

- Cuando debe ser declarada abstracta una clase:
  - Cuando una clase declara un método pero no lo implementa (llamados métodos abstractos)
  - Cuando una clase implementa una interfaz pero no implementa alguno de los métodos declarados en esta
  - Cuando una clase hereda de una clase abstracta y no implementa todos los métodos abstractos de esta.

Clases abstractas

- Cómo se declara una clase abstracta ?

```
public abstract class MiClaseAbstracta {
      public abstract void metodoAbstracto();
      public int f(int i) {
            return i*2;
      }
}
```

- La siguiente instrucción sería inválida:

```
MiClaseAbstracta m = new MiClaseAbstracta();
```

- Pero la siguiente instrucción si sería válida:

```
MiClaseAbstracta n = null;
```

super

- ***super*** sirve como referencia a los métodos y atributos de la superclase del objeto desde el que se está llamando

- Si tenemos la siguiente jerarquía de clases:

```
public class A {
        int i;
        public A(int i) {
                this.i = i;
        }
}
```

```
public class B extends A {
        double k;
        public B(int j, int k) {
                super(j);
                this.k = k;
        }
}
```

  - La instrucción

    ```
    new B();
    ```

    Llamaría primero al constructor del padre, y luego continuaría con la instrucción:

    ```
    this.k = k;
    ```

Sobrecarga de métodos

- Al igual que C++, Java permite métodos sobrecargados (*"overload"*), es decir métodos distintos con el mismo nombre que se diferencian por el número y/o tipo de los argumentos.

- No pueden haber 2 métodos que difieran solo en el tipo de valor retornado !!

- Por ejemplo, podríamos tener un par de métodos que calcularan el mayor de sus argumentos declarados así:

```
int mayor (int numero1, int numero2) { … }
int mayor (int… listaNumeros) { … }
```

Redefinición de métodos

- Al igual que C++, Java permite métodos redefinidos (*"override"*), es decir métodos con el mismo nombre y mismos parámetros pero definidos en clases diferentes, en las cuales una hereda de la otra.

- Es decir, si A declara un método f(), y B hereda de A, es posible que B tenga también un método llamado f() .

Invocación de métodos

- Cuando Java va a invocar un método sobrecargado, escoge el método que más se adecue al número y tipos de parámetros pasados en la invocación.

```
int mayor (int a, int b) {…}
int mayor (long a, long b) {…}
```

Si llamamos al método mayor así:

```
int x = 0, y = 1, z;
long w = 0;
z = mayor (x,y);        // Se llama a mayor (int, int)
z = mayor (x,w);        // Se llama a mayor (long, long)
```

Material

- Recuerden que pueden descargar de

  http://eisc.univalle.edu.co/~dwilches/

  el material correspondiente a las clases 1 y 2

- Para los más avanzados y/o para aclarar conceptos:
  - Thinkin in Java de Bruce Eckel