Lenguajes de Programación
TDA y Orientación a Objetos

Dr. Mauricio Araya

(Autor Original: Jorge Valencia)

Agenda

Índice

1. Introducción a TDAs y OO

Problemas de desarrollo

1. ¿Cómo organizar una pieza de software, de manera que sea fácil de mantener y administrar?
2. ¿Cómo evitar tener que recompilar todo un sistema, en vista de un pequeño cambio?

Modularidad y Reutilización de Software

Ideas:

- Organizar el programa en grupos de subprogramas y datos, lógicamente relacionados, denominados módulos.
- Agrupar en subprogramas que puedan ser compilados de forma independiente, sin necesidad de recompilar todo el sistema.s

Modularidad y Reutilización de Software

- El proceso de diseñar *módulos* o *contenedores sintácticos* se denomina *modularización*.
- Una *unidad de compilación* es un conjunto de subprogramas que pueden compilarse de manera independiente al resto del sistema.

Encapsulación

Encapsulación
Agrupar un conjunto de subprogramas junto a los datos que ellos manipulan. Esto permite resolver los dos problemas presentados.

Ventajas:

- Mayor modularidad
  - Facilita la mantención
  - Permite la reutilización
- Ocultamiento de la información
  - Énfasis en la interfaz abstrae de la implementación
  - Permite la reutilización

Tipos de Datos Abstractos

- Un TDA corresponde a una encapsulación que incluye:
  - La representación de un tipo de dato específico
  - Las operaciones asociadas a ese tipo
- Una instancia de un TDA se denomina *objeto*

Tipos de Datos Abstractos

Ocultamiento de la información
Mediante control de acceso, detalles innecesarios de la implementación de un TDA se ocultan a aquellas unidades fuera de la encapsulación.

Un *cliente* es una unidad de programa que hace uso de una instancia de un TDA.

Ejemplo

```
import java.util.LinkedList;

public class EjemploTDA {

   public static void main (String[] args) {
      LinkedList<Object> lista = new LinkedList<Object>();

      lista.add(new String("EjemploTDA"));
      lista.addFirst(new Integer(42));
      lista.addLast(new Character('e'));

      for(Object o:lista)
         System.out.println(o.toString());

      System.out.println(lista.size());
   }
}
```

Problemas de Reuso

Los TDAs son las unidades a reutilizar, pero:

- Generalmente requieren ser adaptadas para el nuevo uso
- Hacer las modificaciones implicaría entender los detalles de implementación del TDA reutilizado
- De aquí surge la necesidad de establecer relaciones entre distintos TDAs, tales como padre-hijo o similitud de roles

2. Orientación a Objetos

Visión Orientada a Objetos

- Un programa es un conjunto de *objetos* (instancias de alguna clase), que corresponden a abstracción del mundo real (problema)
- Un objeto se comunica con otro mediante paso de *mensajes*
- Un objeto puede alterar su *estado* a causa del procesamiento de mensajes u otros eventos.

Niveles de OO en los Leguajes

Lenguajes OO puros
Realmente consideran que *todo* es un objeto. Ejemplos: Smalltalk, Eiffel, Ruby.

Lenguajes OO no puros
Diseñados para programación OO, pero mantienen algunas características de otros paradigmas. Ejemplos: Java mantiene tipos de datos primitivos (i.e. no-objetos); Python tiene elementos de la programación funcional y la programación orientada a aspectos.

Lenguajes extendidos a OO
Históricamente han seguido otro paradigma pero se han extendido para soportar características de la OO. Ejemplos: C++,Fortran 2003, Perl.

Soporte para Programación OO

Tipos de datos abstractos
Soporte para clases y objetos.

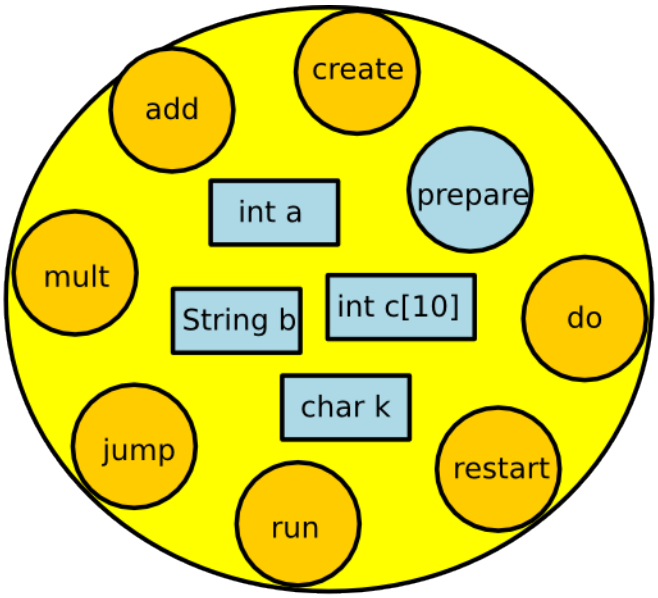
Herencia
Puede ser simple o múltiple.

Polimorfismo
Con especie particular de ligado de métodos a su implementación (clases y métodos virtuales en C++, interfaces en Java).

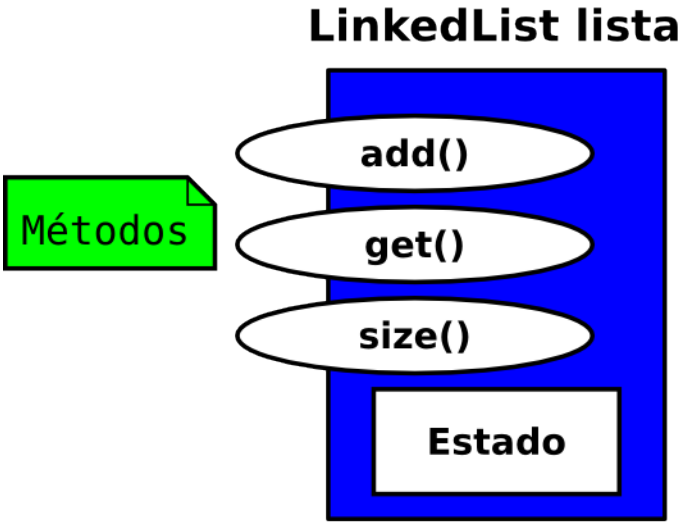
Objetos

- Un objeto tiene *estado* y *comportamiento*
- El estado se mantiene con variables
- El comportamiento se implementa mediante métodos
- Un objeto *encapsula* su estado a través de sus métodos
- Con esto también controla la manipulación de su estado

Idea General de O.O.



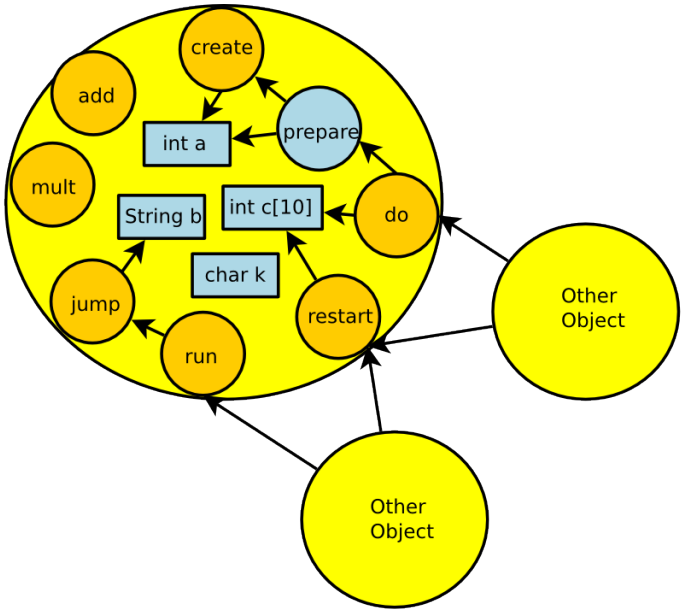
Ejemplo de Objeto: LinkedList



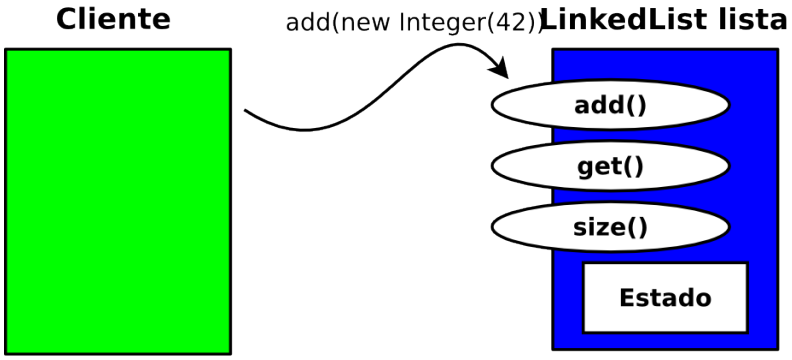
Mensajes

- Objetos interactúan intercambiando mensajes
- Un mensaje consiste en:
  - Identificador del objeto
  - Identificador del método
  - Parámetros, si aplica
- Los objetos no requieren estar en el mismo proceso o máquina

Mensajes (1)



Mensajes (2)



Mensajes: ejemplo en Java

```
import java.util.LinkedList;

public class EjemploTDA {

   public static void main (String[] args) {
      LinkedList<Object> lista = new LinkedList<Object>();

      lista.add(new String("EjemploTDA"));
      lista.addFirst(new Integer(42));
      lista.addLast(new Character('e'));

      for(Object o:lista)
         System.out.println(o.toString());
```

```
        System.out.println(lista.size());
    }
}
```

Clases

- Introducidas en SIMULA 67
- Una clase representa un conjunto de objetos con características comunes
- Se puede interpretar como una *plantilla* de objetos
- Un objeto es una *instancia* de una clase
- Una clase puede tener *variables de clase* (o *campos de clase*) como también *métodos de clase*; estos son comunes para todos los objetos de la misma clase

Clases vs. Objetos

- Una *clase* define un tipo abstracto de datos
- Los objetos son instancias de una clase y, por lo tanto, ocupan memoria
- Se les puede instanciar de forma estática, dinámica de stack o dinámica de heap (C++ permite las tres formas)
- Si se crean en el heap su destrucción puede ser explícita o implícita (en C++ es explícita, en Java es implícita)

Clases: ejemplo en C++

```
#include "clases.h"

complex::complex(double re, double im)
{
      real = re;
      imaginary = im;
}


double complex::getRealPart()
{
      return real;
}

double complex::getImaginaryPart()
{
      return imaginary;
}
```

Clases: ejemplo en C++

```
using namespace std;

#include <iostream>
#include "clases.h"

int main() {
  complex com(1.3, 2.0);

  cout << com.getRealPart() << ' '
    << com.getImaginaryPart() << endl;
  return 0;
}
```

Clases: ejemplo en Java (`ListaEnlazada.java`)

```
public class ListaEnlazada {
  private int dato, tam;
  private ListaEnlazada sgte;

  public ListaEnlazada (int dato) {
    this.dato=dato;
    sgte=null; tam=1;
  }

  public void add(int dato) {
    if (sgte==null)
      sgte=new ListaEnlazada(dato);
    else
      sgte.add(dato);
    tam++;
  }
  public int get(int index) {
    if (index==0)
      return dato;
    return (sgte.get(--index));
  }
  public int getFirst() { return dato; }
  public int getLast() { return get(tam - 1); }
  public int size() { return tam; }
}
```

Clases: ejemplo en Java (`EjemploListaEnlazada.java`)

```
public class EjemploListaEnlazada {
  public static void main(String[] args) {
    ListaEnlazada lista=new ListaEnlazada(6);
    lista.add(5); lista.add(4); lista.add(3);
    lista.add(2); lista.add(1); lista.add(0);

    System.out.println(lista.getFirst());
    System.out.println(lista.getLast());
    System.out.println(lista.get(3));
    System.out.println(lista.size());
  }
}
```

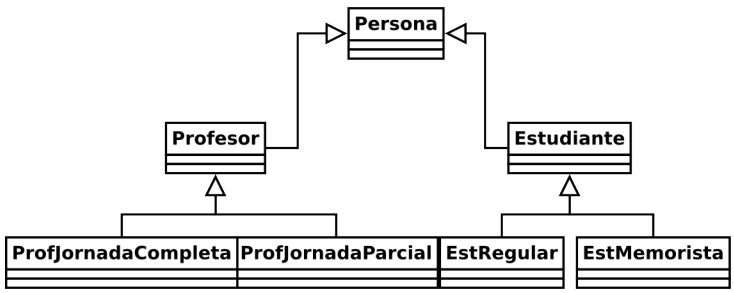
Herencia

- Permite reusar tipos de datos abstractos ya definidos, sin alterarlos
- La reutilización se realiza definiendo nuevos TDAs que modifican o extienden la funcionalidad del TDA base
- De este modo se permite adaptar un TDA a las condiciones particulares de un problema con menos esfuerzo

Jerarquía de Herencia

- Una clase que hereda de otra se denomina *clase derivada* o *subclase*
- La clase superior se denomina *clase padre* o *superclase*

Jerarquía de Herencia: ejemplo



Jerarquía de Herencia: ejemplo en Java (`Persona.java`)

```
public class Persona {
  private String rut;
  private String nombre;

  public Persona (String rut, String nombre) {
    this.rut=rut;
    this.nombre=nombre;
  }

  public String getRut() {
    return rut;
  }

  public String getNombre(){
    return nombre;
  }

  public String toString(){
    return nombre+"("+rut+")";
  }
}
```

Jerarquía de Herencia: ejemplo en Java (`Estudiante.java`)

```
public class Estudiante extends Persona {
  private String rol;

  public Estudiante (String rut, String rol) {
    super(rut,"Jorge");
    this.rol=rol;
  }

  public String getRol() {
    return rol;
  }
}
```

Jerarquía de Herencia: ejemplo en Java (`EjemploHerencia.java`)

```
public class EjemploHerencia {

  public static void main (String[] args) {
    Persona profesor = new Persona("12.345.678-9","profe");
    Estudiante estudiante = new Estudiante("13.456.789-2",
                                           "2405063-1");

    System.out.println(profesor.getRut());
    System.out.println(estudiante.getRut() +
                       "<->" + estudiante.getRol());

    // profesor.getRol()  /* Error */

    /* Se puede? */
    Persona x = new Estudiante("14.567.892-3",
                               "2391505-7");
    System.out.println(x.getRut());
    System.out.println("La persona " + x);
    // System.out.println(x.getRol());  /* Error */
  }

}
```

Herencia: Relaciones entre Clases

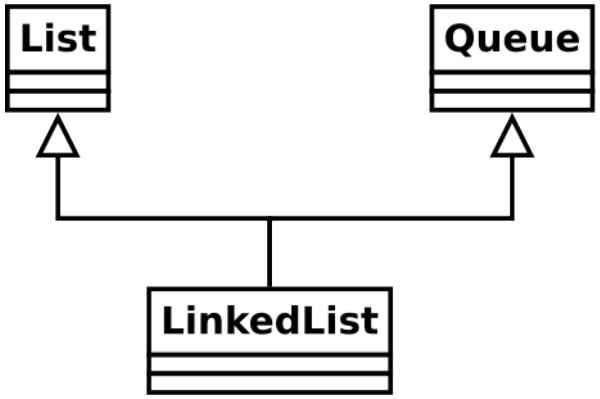
- El mecanismo de herencia permite extender una clase (agregar variables o métodos)
- Si `B` extiende a `A`, entonces `B` (subclase) hereda todo lo de `A` (superclase)
- Una subclase heredera tiene la opción de reimplementar a su modo algun método heredado (*override*)

Ventajas de la Herencia

- Subclases pueden proveer un comportamiento especializado basado en la superclase
- Los programadores pueden definir *clases abstractas*, que definen un comportamiento genérico y que debe ser definido por las clases herederas
- En resumen, representa un mecanismo de reutilización de interfaces y código

Herencia Múltiple: ejemplo

La nueva clase hereda a partir de dos o más clases:



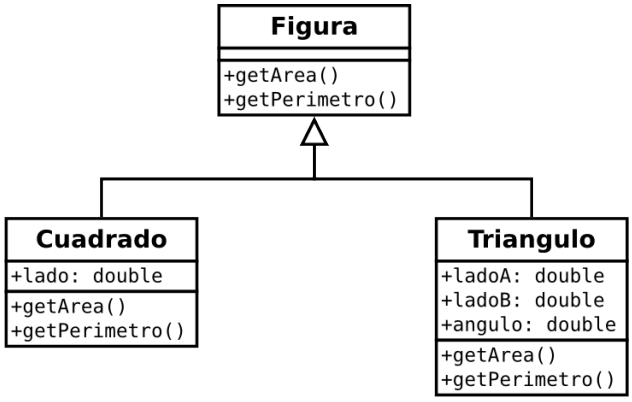
Problemas con la Herencia Múltiple

- Colisión de nombres
- Pérdida de eficiencia dada su complejidad (sobretodo en el ligado de métodos)
- No es claro que su uso mejore el diseño y la facilidad en la mantención de sistemas, teniendo en cuenta el aumento en la complejidad de la organización de la jerarquía

Polimorfismo

- Permite tratar a objetos como si fueran del tipo de la superclase
- Las subclases implementan métodos que llevan el mismo nombre que tienen en la superclase, cambiando su comportamiento
- Cuando se invoca a uno de estos métodos se debe hacer un ligado dinámico a la implementación que corresponda

Polimorfismo: ejemplo



Clases y Métodos Virtuales

- A veces no tiene sentido la instanciación de objetos de determinadas clases, dependiendo de la abstracción realizada
- En estos casos la clase puede implementar un grupo de métodos y definir un prototipo de los métodos que deben implementar sus subclases
- Los estos prototipos de métodos se llaman *métodos virtuales*; toda clase que tenga al menos un método virtual se denomina *clase virtual*

Clases y Métodos Virtuales: ejemplo en C++ (`figura`)

```
using namespace std;

class figura {
  public:
    virtual double getArea() = 0;
    virtual double getPerimetro() = 0;
};
```

Clases y Métodos Virtuales: ejemplo en C++ (`cuadrado`)

```
using namespace std;

class cuadrado: public figura {
  private:
    double lado;

  public:
    cuadrado(double l) { lado = l; }
    double getArea() { return lado * lado; }
    double getPerimetro() { return 4.0 * lado; }
};
```

Clases y Métodos Virtuales: ejemplo en C++ (`circulo`)

```
class circulo: public figura {
  private:
    double radio;
    static const double pi = 3.14159265358979323846;

  public:
    circulo(double r) { radio = r; }
    double getArea() { return pi * radio * radio; }
    double getPerimetro() { return 2 * pi * radio; }
};
```

Clases y Métodos Virtuales: ejemplo en C++ (`uso_figuras.cpp`)

```
#include <iostream>
#include "figura"
#include "cuadrado"
#include "circulo"

using namespace std;

int main ()
{
  figura *fig[2];
  fig[0] = new cuadrado(2.3); fig[1] = new circulo(3.0);

  for(int i = 0; i < 2; i++)
    cout << fig[i]->getArea() << ' '
         << fig[i]->getPerimetro() << endl;

  return 0;
}
```

Interfaces

- Definen un protocolo para la interacción entre objetos sin necesidad de conocer su clase
- Una o más clases pueden implementar una misma interfaz
- Una clase que implemente a una interfaz debe *necesariamente* implementar cada uno de sus métodos

Interfaces: ejemplo en Java (`Ejecutable.java`)

```
public interface Ejecutable {
  public void ejecutar();
}
```

Interfaces: ejemplo en Java (`Tarea.java`)

```
public class Tarea implements Ejecutable {

  private int n;
  private String mensaje;

  public Tarea(int n, String mensaje) {
    this.n=n;
    this.mensaje=mensaje;
  }

  public void ejecutar() {
    for (int i=0; i<n; i++)
      System.out.println(mensaje);
  }
}
```

Interfaces: ventajas

- Capturan similitudes entre clases no relacionadas, sin forzar una relación artificial entre ellas
- Permiten declarar métodos que se espera que implementen las clases que *responden* a la interfaz
- Permiten establecer un *contrato* de programación, sin revelar la implementación

3. Introducción al lenguaje Java

Historia

1991: James Gosling inicia un proyecto para escribir código independiente de la arquitectura para sistemas empotrados. Intenta con C++ pero no le satisface.

1993: Desarrolla un nuevo lenguaje llamado *OAK*, similar a C++ pero más seguro y portable.

1994: Aparece la *World Wide Web* y *Mosaic*.

1995: Sun anuncia la disponibilidad del lenguaje *Java* y el browser *HotJava* con soporte para *applets*

En la actualidad se reconoce al leguaje Java por ser una tecnología clave en la explosión de la WWW.

Tecnología Java

Leguaje de Programación
Especificado por Sun Microsystems, orientado a objetos, con sintaxis similar a la de C++.

Plataforma
Incluye a la *Java Virtual Machine* y la especificación de una extensa *API*.

Tres sabores...
Se distribuye, segun la necesidad, en al menos tres ediciones:

- **JSE**: Java Standard Edition
- **JME**: Java Micro Edition
- **JEE**: Java Enterprise Edition

Características

- Leguaje simple, orientado a objetos, sintaxis similar a C++.
- Facilidades para distribución de componentes (RMI, integración CORBA).
- Soporte multihebra en la JVM, con tipos de dato *thread-safe* para concurrencia.
- Robusto y seguro, diseñado para producir software confiable. Realiza revisiones en tiempo de compilación y ejecución.
- Protable, neutro de la arquitectura.
- Alto rendimiento, con posibilidad de compilar parte de un programa a código de máquina.

API Java

- `java.applet`: Desarrollo de applets
- `java.awt`: Interfaces gráficas de usuario (GUIs)
- `java.io`: Entrada/salida en general; archivos, consola, streams, etc.
- `java.lang`: Tipos de datos fundamentales del lenguaje; strings, fecha, hora, propiedades del sistema, etc.
- `java.math`: Clases utilitarias para matemáticas
- `java.net`: Redes, sockets, streams, etc.
- `java.rmi`: Remote Method Invocation

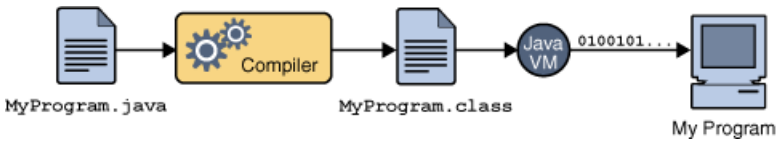
API Java

- `java.security`: Encriptación, control de acceso, autenticación, políticas de seguridad, etc.
- `java.sql`: Java DataBase Conectivity (JDBC)
- `java.text`: Procesamiento de textos
- `java.util`: Utilidades varias como el Collection Framework (colecciones, listas, pilas, colas, vectores, matrices, etc.), internacionalización (i18n, l10n), eventos, etc.
- `javax`: contiene paquetes adicionales con clases para accessibilidad, funciones criptográficas, manejo de imágenes, servicios de nombres, impresión, sonido, SWING, transacciones, XML, CORBA, Java2D, Java3D, extensiones a la API de red, rmi, seguridad, etc.

¿Dónde aprender sobre Java?

- Java 2: Bruce Eckel, *Thinking in Java, 3rd Edition*, Prentice Hall, http://www.mindview.net/Books/TIJ/.
- Java 5/6:
  - Bruce Eckel, *Thinking in Java, 4th Edition*, Prentice Hall, http://mindview.net/Books/TIJ4.
  - Sun's Java Tutorials: http://java.sun.com/docs/books/tutorial/.
  - Java API Specification: http://java.sun.com/javase/6/docs/api/.
  - Java Language Specification: http://java.sun.com/docs/books/jls/index.html.

Ciclo de un programa Java



Ejecución de un programa Java

- Para ejecutar un programa se debe iniciar la *Máquina Virtual Java* (JVM), indicándole el nombre de la clase cuyo método `main` será ejecutado.
- Esta clase **no** es instanciada.
- Este método `main` es el encargado de instanciar los objetos que compondrán el programa y hacerlos interactuar.

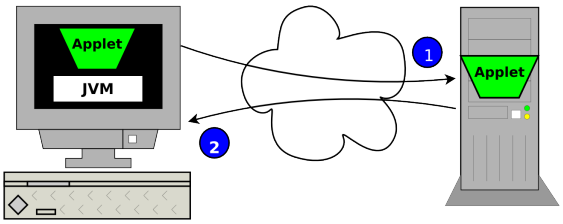
¡Hola Mundo!

```
public class HolaMundoApp {
        public static void main(String[] args) {
                System.out.println("Hola Mundo!");
        }
}
```

Applets

- Significa "aplicacioncita".
- Su uso normal es a través de la Web.
- Difiere de un programa normal en su modelo de ejecución.
- El browser es el encargado de iniciar una máquina virtual que **no** correrá un método `main`, sino que instanciará la clase especificada y la dibujará con ayuda de la API AWT.

Applets: modelo de ejecución



Applets: ejemplo

```
import java.util.*;
import java.sql.*;

/**
 * la clase HolaMundo implementa una applet que
 * simplemente despliega "Hola Mundo!".
 */
public class HolaMundoApplet extends java.applet.Applet {
        public void paint(java.awt.Graphics g) {
                java.util.Date fecha;
                g.drawString("Hola Curso Dormido!", 50, 25);
        }
}
```

Applets: ejemplo

```
<HTML>

<HEAD>
<TITLE>Un Programa Simple con Applet</TITLE>
</HEAD>

<BODY>
Aqui viene la salida del programa:
<APPLET CODE="HolaMundoApplet.class"
                WIDTH=150 HEIGHT=25>
</APPLET>
</BODY>

</HTML>
```

Nombres

- Los *nombres de las clases* se agrupan en *paquetes*, los que se comportan como *espacios de nombres*.
- Para utilizar una clase A desde una clase B, entonces hay que *importar* el espacio de nombres (paquete) al que pertenece A.
- Este mecanismo permite evitar colisiones de nombres.
- Para declarar el paquete al que pertenece una clase se usa la palabra reservada `package`.

```
package sinsentido;
public class ClaseNula{
}
```

Nombres

- Por ejemplo, la clase `Date` existe tanto en el paquete `java.util` como en `java.sql`. Para seleccionar una de las dos a utilizar se debe hacer referencia a su nombre completo (`paquete.Clase`) o simplemente importar una vez el paquete en que se encuentra.
- En el ejemplo del applet se utilizan las sentencias `import java.applet.*` e `import java.awt.*` para poder utilizar todas las clases de los paquetes `java.applet` y `java.awt`.

Subclases

- La palabra clave `extends` permite establecer una relación de herencia.
- En el ejemplo del applet, la clase `HolaMundoApplet` hereda la implementación de la clase base `java.applet.Applet`:

public class HolaMundoApplet **extends** Applet ...

Métodos

- Una clase puede implementar uno o más métodos; estos definen el comportamiento de los objetos de dicha clase. En el ejemplo:

```
public void paint(Graphics g) {
    g.drawString("Hola Mundo!", 50, 25);
}
```

- En el caso de los applets, el método `paint` se debe implementar para indicar la forma en que dicho applet se dibujará en alguna región de la pantalla.

Codificación

- Java utiliza el estándar Unicode, separando el *repertorio de caracteres* del esquema de *codificación*.
- La codificación particular utilizada es UTF-16, que utiliza una cantidad variable de bits para representar un caracter.
- UTF-16 interpreta de forma correcta, y sin mayor esfuerzo, codificaciones como ASCII básico (7 bits) con extensiones como `latin1` (ISO 8859-1).

Identificadores

- Deben comenzar con una letra, incluido _ o $, seguido de letras o dígitos
- Debido a que el repertorio de caracteres Unicode es más amplio, se permiten identificadores como `Árbol`, `Hähnchen`, etc.
- Es sensible a la capitalización (e.g. `árbol` es diferente a `Árbol`)
- Java define algunas palabras reservadas que no pueden ser usadas como identificador (ver especificación del lenguaje)

4. Variables, Constantes y Tipos Primitivos

Datos primitivos

- En general se comportan igual que C
- Java hace inicialización automática de sus valores
- Cada tipo primitivo (excepto `short` y `byte`) tiene una clase declarada en el paquete `java.lang` que define constantes tales como:
  - `MIN_VALUE` y `MAX_VALUE`
  - `NEGATIVE_INFINITY` y `POSTIVE_INFINITY`
  - `NaN` (Not a Number)

Declaración de variables

[modificador] tipo variable {, variable}*

- Modificador es opcional. Opciones: `static` o `final`
- `final` sólo se puede usar en campos (atributos)
- Ejemplo: **float** x, y;
- Declaraciones pueden aparecer casi en cualquier parte del código
- La visibilidad de una variable se limita al bloque en que se declara

Resolución de nombres

- Declaración local a un bloque (e.g. Loop)
- Parámetro de un constructor o método
- Un miembro de una clase o interfaz
- Tipos explícitamente importados
- Otros tipos declarados en el mismo paquete
- Tipos importados implícitamente nombrados
- Paquetes disponibles en el sistema host

Valores iniciales

- Una variable se puede inicializar en su declaración. Por ejemplo: **final double** pi = 3.14159;
- Java asigna valores por omisión a los campos de una clase si no se especifica
- Este valor es cero, `false` o `null`, para datos primitivos, datos lógicos o referencias, respectivamente
- Variables locales de un método, constructor o inicializador estático no se inicializan
- Una variable se inicializa cada vez que se alcanza su declaración

5. Operadores

Precedencia y asociatividad

- A igual precedencia, se asocia por la izquierda, excepto asignación que es por la derecha
- La precedencia se pueda cambiar usando paréntesis
- Para legibilidad se recomienda usar paréntesis, cuando sea posible, de manera de hacer explícita la precedencia, sin exagerar. Por ejemplo:

```
while ((v = stream.next()) != null)
   procesar(v);
```

Expresiones

- Una expresión se evalúa de izquierda a derecha
- Orden puede ser importante cuando existen efectos laterales o corto-circuito
- Cada operando se evalúa antes de realizar la operación
- Cada expresión tiene un tipo
- En general domina el operando de mayor rango de valores (e.g. **double** + **long**→**double**)

Conversión de tipo: implícita

Esta se realiza de forma automática.

- Conversión implícita... ¡Es automática!
- Se permite entre valores primitivos cuando se soporta un mayor rango de valores
- No se permite de punto flotante a entero
- En la conversión se puede perder precisión
- Una referencia a un objeto de una clase incluye la compatibilidad para los *supertipos*
- Se puede usar una referencia a un objeto de un tipo cuando se requiera una referencia a un supertipo (upcast)

Conversión de tipo: explícita

- Se requiere cuando un tipo no se puede asignar a otro por conversión implícita
- La conversión explícita se denomina *cast*
- Ejemplo: **double** d = 7.99; **long** l = (**long**) d;
- Se puede usar para referencias a objetos con conversión no segura (downcast)
- Operador `instanceof` permite verificar si se puede aplicar cast a un objeto **if** (obj **instanceof** clase) ...

6. Clases y objetos

Conceptos generales

- Las clases definen los métodos que detallan el comportamiento de un objeto
- Los campos o atributos definen el estado
- Un método tiene una firma sintáctica, pero es su implementación la que define su semántica

Modificadores de clases

public
Por omisión una clase sólo es accesible por clases del mismo paquete, salvo que sea declarada pública

abstract
La clase no puede ser instanciada

final
La clase no puede ser derivada

Atributos

- Cada objeto de una clase tiene sus propias instancias de cada variable miembro
- Significa que cada objeto tiene su propio estado
- Cambio de estado en un objeto no afecta a otros objetos
- Variables miembros se pueden compartir entre todos los objetos de una clase con el modificador `static`
- Todos los objetos de una clase comparten una única copia de un campo declarado como `static`
- En general, cuando se habla de variables y métodos miembros se refiere a aquellos no estáticos

Control de acceso

- Todos los métodos y variables miembro están disponibles para el código de la propia clase
- Para controlar el acceso a otras clases y subclases, los miembros tienen 4 posibles modificadores:
  - Privado: Declarados con `private` son sólo accesibles por la propia clase
  - Paquete: Miembros sin modificador de acceso son sólo accesibles por código y heredados por subclases en el mismo paquete
  - Protegido: Declarados con `protected` son accesibles por una subclase (que puede estar fuera del paquete), como también por código del mismo paquete
  - Público: Declarados con `public` son accesibles por cualquier clase

Creación de objetos

```
Persona p1;
Persona p2=new Persona("Pedro","10.333.291-6");
```

- Se han declarados dos referencias a objetos de clase `Persona`
- La declaración de una variable **no** crea un objeto, sino que una **referencia** a un objeto, que inicialmente es `null`
- Cuando se usa el operador `new`, el JRE crea un objeto, asignando suficiente memoria, e inicializando el objeto con algún constructor
- Si no existe suficiente memoria se ejecuta el recolector de basura (*garbage collector*), y si aún no existe suficiente memoria, se lanza un `OutOfMemoryError`
- Terminada la inicialización, el JRE retorna la referencia al nuevo objeto

Constructores

- Un objeto recién creado debe inicializar las variables miembro
- Las variables miembro pueden ser inicializadas explícitamente en la declaración
- Muchas veces se requiere algo más (e.g. ejecutar código para abrir un archivo)
- Los constructores cumplen ese propósito
- Son "métodos" especiales, tienen el mismo nombre de la clase y pueden recibir parámetros, pero no retornan un valor
- Una clase puede tener varios constructores

Bloques de inicialización estáticos

- Permiten a una clase inicializar variables miembro estáticas u otros estados necesarios
- Se ejecutan en el orden que aparecen declarados
- Ejemplo: `UnicaInstancia.java`

```
class UnicaInstancia {
   private static UnicaInstancia instancia;

   static {
      instancia=new UnicaInstancia();
   }

   public UnicaInstancia() {}

   public static UnicaInstancia getInstancia() {
      return instancia;
   }
}
```

Método `finalize`

- Permite ejecutar código de finalización antes de liberar memoria
- Es el equivalente al concepto de destructor en C++
- Es útil cuando se usan recursos externos y que deben liberarse (evitar fuga de recursos)
- Ejemplo: cerrar archivos o conexiones de red abiertas

```
protected void finalize() throws Throwable {
    super.finalize();
    archivo.close();
}
```

Método `toString`

- Si un objeto soporta el método `toString`, este método se invocará en cualquier expresión que requiera hacer transformación implícita al tipo `String`.

```
public class Persona {
  private String rut;
  private String nombre;

  public Persona (String rut, String nombre) {
    this.rut=rut;
    this.nombre=nombre;
  }

  public String getRut() {
    return rut;
  }

  public String getNombre(){
    return nombre;
  }

  public String toString(){
    return nombre+"("+rut+")";
  }
}
```

Método `toString`

```
Persona p=new Persona("12.345.876-9",
                      "Gonzalo Gonzalez");
System.out.println("Se creo la persona: " + p);
```

Constructores y herencia

- Al crear un objeto, el orden de ejecución es:
  - Se invoca el constructor de la superclase (si no se especifica cuál, se usa el por omisión)
  - Inicializar los campos usando sentencias de inicialización
  - Ejecutar el cuerpo del constructor
- Si se quiere usar un constructor específico de la superclase debe invocarse con `super(...)`

Otros métodos de `Object`

public boolean equals(Object obj)
Compara si dos objetos tienen el mismo valor. Por omisión se supone que un objeto es sólo igual a sí mismo. No es igual a verificar que tienen la misma referencia (se puede hacer con el operador ==)

public int hashCode()
Retorna código hash del objeto, que es usualmente único para cada objeto. Sirve para ser usados en tablas de hash.

Otros métodos de `Object`

protected Object clone() throws CloneNotSupportedException
Retorna un clon del objeto.

public final Class getClass()
Retorna un objeto de tipo `Class` que representa la clase del objeto `this`. Permite utilizar reflexión.

Métodos `equals` y `hashCode`

- Los métodos `equals` y `hashCode` deben ser redefinidos conjuntamente
- Normalmente dos objetos no son iguales, dado que retornan diferentes códigos de hash
- Para que dos objetos sean iguales en valor deben retornar el mismo código de hash

Clonación de objetos

- El método `clone` permite clonar un objeto
- Cambios posteriores en el clon no afectan al estado del objeto original
- Una clase que permite clonar objetos normalmente implementa la interfaz `java.lang.Cloneable`
- Aquellas clases que tienen atributos que son referencias deben redefinir el método `clone` para clonar los objetos referenciados

Clonación de objetos: ejemplo

```
import java.util.*;

public class Stack implements Cloneable {
  private Vector items;

  //  codigo de los metodos y constructor de Stack

  protected Object clone()
     throws CloneNotSupportedException {
    Stack s=new Stack();
    s.items = (Vector) items.clone(); // clona el vector
    return s; // retorna el clon
  }
}
```

7. Tipos de datos frecuentes

Arreglos

- En Java los arreglos son objetos. Los objetos arreglo son instancias de clases especiales que heredan todos los miembros de la clase `Object`
- El único método que es sobreescrito es `clone`, cuyo tipo de retorno es `T[]`, donde `T` es el *tipo componente* del arreglo; además agrega el atributo **public final** length, que indica el largo del arreglo
- La clase utilitaria `java.util.Arrays` provee algunos algoritmos interesantes para operar con arreglos
- '?Cómo se llama la clase de un objeto arreglo?
- '?Qué pasa si intento acceder a un índice fuera del rango permitido?

Arreglos: ejemplo (`EjemploArreglos.java`)

```
class EjemploArreglos {
  public static void main(String[] args) {
    int ia[][] = { {1, 2}, null };

    System.out.println(ia.getClass().getName());
    System.out.println(ia[0].getClass().getName());
    System.out.println();

    try {
      for (int[] ea : ia)
        for (int e: ea)
          System.out.println(e);
    } catch (Exception e) {
      System.err.println("Ups!␣"+e);
    }

    System.out.println(ia[10]);
  }
}
```

Strings

- Para uso básico de strings Java provee las clases

  `Character`: para manipulación de caracteres

  `String`: para utilización de cadenas *inmutables*

`StringBuffer`: para utilización de cadenas *mutables*

- `Character` es una clase *wrapper* para el tipo primitivo `char`
- La documentación de la API Java describe ampliamente los distintos usos de estas clases y otras más para manipulación de strings como `java.util.StringTokenizer`, `java.io.StringReader` o `java.io.StringWriter`

Strings: ejemplo (`EjemploStrings.java`)

```
public class EjemploStrings {
  public static void main(String[] args) {
    String invertir = "lenguajes";
    if (args.length > 0)
      invertir=args[0];

    int largo = invertir.length();
    StringBuffer destino = new StringBuffer(largo);

    for (int i = (largo - 1); i >= 0; i--)
      destino.append(invertir.charAt(i));

    System.out.println(destino);
  }
}
```

Números

- La clase abstracta `java.lang.Number` provee una base para conversión entre tipos numéricos
- Sus únicos métodos son `byteValue`, `doubleValue`, `floatValue`, `intValue`, `longValue` y `shortValue`
- De ella heredan las clases wrappers `Byte`, `Double`, `Float`, `Integer`, `Long` y `Short`
- Todas estas clases incluyen métodos de clase para conversión de strings al tipo primitivo específico, permitiendo la especificación de una base (sólo para enteros): **float** pi=Float.parseFloat("3.14"); **int** x=Integer.parseInt("101",2); // *x*

Números

La clase utilitaria `java.lang.Math` provee las constantes de clase `E` (para Euler) y `PI` (para π) como `double` para su utilización en conjunto con una serie de funciones matemáticas como:

- Valor absoluto
- Funciones trigonométricas y sus inversas
- Aproximación a enteros
- Exponenciación y logaritmos
- Números al azar
- Signo, máximos y mínimos
- Conversión de ángulos en grados $\leftrightarrow$ radianes

8. Excepciones

Motivación

- Cuando ocurren errores es importante que un programa sea capaz de reaccionar al evento; por ejemplo:
  - Notificar al usuario de un error
  - Guardar el trabajo hecho
  - Volver a un estado anterior seguro
  - Terminar limpiamente el programa
- Los errores se pueden producir por problemas en la entrada de datos, error en un dispositivo (e.g. Impresora apagada, disco lleno), errores en el código del programa, etc.

Excepciones

- Una excepción corresponde a un evento que interrumpe el flujo normal de ejecución
- Cuando ocurre tal tipo de evento, la JVM avisa de este al programa *lanzando* un objeto de excepción
- De este modo el control se traslada a algún punto definido por el programador
- Si este punto se define, entonces se captura la excepción invocando al manipulador de la excepción; sino se se termina el programa
- Se dice que el punto en que se produce el problema *lanza* (*throws*) una excepción; si existe un punto de control de la excepción, se dice que este *captura* (*catch*) la excepción

Ventajas de excepciones

- Provee un mecanismo limpio para el tratamiento de errores (evita tener que estar retornando valores no esperados para inidicar problemas)
- Separa el código para tratamiento de errores del código de ejecución normal
- Permite agrupar las condiciones de error
- Como toda excepción es una instancia de una clase, estas se pueden heredar para afinar el control de errores
- Se propagan entre llamadas anidadas, por lo que es posible controlar una excepción en cualquier parte del stack de llamados

Tratamiento de errores: ejemplo en C

De la página de manual de `fopen` (`man fopen`):

```
...
VALOR DEVUELTO
       Cuando acaban bien, fopen, fdopen y freopen
       devuelven un puntero a FILE. Cuando no,
       devuelven NULL y la variable global errno
       contiene un valor que indica el error.

ERRORES
       EINVAL El modo pasado a fopen, fdopen, o a
       freopen no era válido.
...
```

Tratamiento de errores: ejemplo en Java

```
import java.io.*;

public class EjemploExp {
  public static void main(String[] args) {

     try {
        BufferedReader archivo = new BufferedReader(
                                      new FileReader("EjemploExp.java"));

        for (String s=archivo.readLine() ; s!=null ; s=archivo.readLine())
           System.out.println(s);

     } catch (EOFException e) {
        System.out.println("Fin del archivo");
     } catch (FileNotFoundException e) {
        System.err.println("El archivo no existe");
     } catch (IOException e) {
        System.err.println("Error de E/S");
     }
  }
}
```

Anidamiento de excepciones: ejemplo

```
void metodo1() {
  try {
    metodo2();
  } catch (Exception e) {
    System.err.println("Ups! Ocurrio el siguiente error:" + e);
  }
}

void metodo2() throws IllegalArgumentException {
  metodo3(15.0);
}

void metodo3(double f) throws IllegalArgumentException {
  if (f>1)
    throw new  IllegalArgumentException("El argumento debe ser menor a uno.");
}
```

Jerarquía de errores

- En Java toda excepción se deriva de la clase `Throwable`
- Existen dos subclases conocidas:
  - **`Error`**. Representa un error interno o agotamiento de recursos en el sistema runtime de Java
  - **`Exception`**. Representa un error en el programa.
- Las excepciones se pueden clasificar en:
  - *Unchecked Exceptions*: Identificadas for ser instancias de `RuntimeException`, `Error` o cualquiera de sus subtipos. Representan problemas que pueden ocurrir en demasiadas partes de un programa y de los cuales, a menudo, es díficil recuperarse.
  - *Checked Exceptions*: Corresponden a todo el resto de las excepciones y toman este nombre porque el compilador Java hace un revisión del correcto tratamiento de éstas excepciones que puedan ocurrir.

Tratamiento de excepciones

- Un método debe advertir al compilador sobre **todas** las excepciones verificadas (*checked* que no puede tratar por medio de la palabra reservada `throws`: **public** String readLine() **throws** IOException {...}
- Al advertir sobre una clase de excepciones sin tratar, entonces puede lanzar cualquier excepción de alguna subclase de ella
- Aquellas excepciones que son capturadas (`catch`) no salen del método y no debieran ser declaradas

Tratamiento de excepciones: lanzar

- Se debe elegir una clase apropiada de excepción (se pueden crear excepciones personalizadas si es necesario)
- Hay que instanciar un objeto de excepción de esa clase
- Finalmente la excepción es lanzada con `throw`

```
if (x>1)
   throw new IllegalArgumentException(
         "El argumento debe ser menor a uno.");
```

Tratamiento de excepciones: capturar

- Palabra clave `try` permite definir un bloque de sentencias para las cuales se quiere controlar excepciones
- Para cada clase de excepción se define un bloque de control diferente con `catch`
- Usando una superclase común, varias subclases diferentes de excepciones pueden tener un único bloque de control
- Se puede utilizar la cláusula `finally` para definir un bloque de código que se debe ejecutar en caso de excepción o ejecución normal
- Estos bloque s de control puede relanzar la excepción o incluso lazar una excepción distinta

Captura de excepciones: ejemplo

```
BufferedReader archivo;
try {
   archivo=new Bufferedreader(new FileReader("miarchivo");
   ...
} catch(IOException e) {
   System.err.println("Error de E/S con el archivo:" + e);
} catch(Exception e) {
   System.err.println("Se encontro un error desconocido: " + e);
} finally {
   if (archivo != null)
      archivo.close();
}
```

9. Streams y archivos

Streams

- E/S es en principio una secuencia de bytes (e.g. archivos, dispositivos, conexiones de red, etc.)
- Java provee dos clases básicas para E/S de bytes: `InputStream` y `OutputStream`
- Estas clases no son convenientes para manejo de caracteres Unicode (y menos para codificación UTF-16 en la que un caracter tiene un largo variable múltiplo de 2 bytes)
- Para ello la API Java incluye los tipos `Reader` y `Writer` que son abstracciones sobre los streams de bytes
- De estas cuatro clases abstractas Java deriva las demás clases para E/S

Clases útiles para lectura

- `java.io.FileReader`: Implementación de `Reader` para archivos. Incluye un construtor para instanciar objetos a partir de un nombre de archivo.
- `java.io.BufferedReader`: Permite realizar lectura línea a línea desde un stream. Incluye un constructor para crear objetos desde cualquier implementación de `Reader` (e.g. `FileReader`).
- `javax.sound.sampled.AudioInputStream`: Permite leer audio en distintos formatos y calidad. Es capaz de obtener datos desde un archivo, un stream ya abierto o una URL.

Clases útiles para escritura

- `java.io.PrintWriter`: Permite escritura para distintos tipos de datos, con métodos polimórficos `print` y `println`. Se puede instanciar a partir de objetos `File`.
- `java.io.ObjectOutputStream`: Permite serializar objetos completos para almacenar en medio persistente, para transmisión a través de la red, etc. La serialización realizada puede luego ser reconstruida con un objeto `java.io.ObjectInputStream`.

Archivos

- E/S en archivos se puede realizar con cualquiera de las clases nombradas
- Para acceso aleatorio a archivos se puede utilizar `java.io.RandomAccessFile` que, además de operaciones comunes de E/S, permite operaciones como seek(**long** pos) y skipBytes(**int** n)
- Para operaciones que no sean E/S en streams existe la clase `java.io.File`
- Esta clase permite crear, listar,renombrar y eliminar archivos y directorios, comprobar y cambiar permisos, revisar hora y fecha de modificación, entre otras operaciones.

EOC

Fin del capítulo 3.