



UNIVERSIDAD TÉCNICA
FEDERICO SANTA MARÍA

DEPARTAMENTO
DE INFORMÁTICA

LENGUAJES DE PROGRAMACIÓN

Equipo de Profesores:
Jorge Díaz Matte - Rodrigo Salas Fuentes

Unidad 5

Tipos de Datos Abstractos y Programación Orientada al Objeto

5.1 Tipos de Datos Abstractos

5.2 Orientación a Objetos (visión general)

5.3 Clases y Objetos

5.4 Herencia

5.5 Interfaces

5.6 Manejo de Excepciones

5.7 Paquetes

Introducción

- Mayoría de lenguajes modernos proveen soporte para proveer abstracciones de datos y de procesos.
- El concepto de tipo de datos abstracto (TDA) permite en una sola abstracción unificar las abstracciones de datos y de proceso, cuando éstas están relacionadas, en un solo tipo definido por el usuario.
- Este mecanismo permite unificar la representación de los datos y el código que lo manipula en un TDA que oculta los detalles de la implementación.
- La Orientación a Objetos está fuertemente basada en la idea de TDA.

5.1 Tipos de Datos Abstractos

Tipo de Dato Abstracto

DEFINICIÓN: Es un tipo de datos definido por el usuario que satisface dos restricciones:

- **Ocultamiento de Información:** separación de la interfaz del tipo definido respecto a la representación de objetos y el código de los operadores (implementación), estando esto oculto para unidades de programa que los utilizan.
- **Encapsulamiento:** la declaración del tipo y los protocolos de las operaciones sobre objetos del tipo (que define su interfaz) están contenidos en una única unidad sintáctica.

Tipo de Dato Abstracto: Ventaja (1/2)

- **Modularidad:** encapsulamiento de especificación de datos y operaciones en un solo lugar promueve la modularidad, siendo conocido por otras unidades de programa sólo por su interfaz, bajando así la carga cognitiva (variables, código, conflictos de nombres, etc.).
- **Modificabilidad:** es reforzada al proveer interfaces que son independientes de la implementación, dado que se puede modificar la implementación del módulo sin afectar el resto del programa. Promueve compilación separada.

Tipo de Dato Abstracto: Ventaja (2/2)

- **Reusabilidad**: interfaces estándares del módulo permite que su codificación sea reusada por diferentes programas.
- **Seguridad**: permite proteger el acceso a detalles de implementación a otras partes del programa.

Tipo de Dato Abstracto: Ejemplo

Stack: se puede abstraer mediante el siguiente conjunto de operaciones sobre el tipo `stack`:

- `create(stack):` Construye un nuevo *stack*
- `destroy(stack):` Destruye el *stack*
- `empty(stack):` Verifica que el *stack* está vacío
- `push(stack, elem):` Coloca un elemento en el *stack*
- `pop(stack):` Extrae un elemento del *stack*
- `top(stack):` Obtiene valor del tope del *stack*

Tipo de Dato Abstracto: Ejemplo 1 en C++

```
class Stack {  
    private:  
        int *stackPtr, maxLen, topPtr;  
  
    public:  
        Stack() { // constructor  
            stackPtr = new int [100];  
            maxLen = 99;  
            topPtr = -1;  
        };  
  
        ~Stack () {delete [] stackPtr;};  
  
        void push (int number) {  
            if (topPtr == maxLen)  
                cerr << "Error en push - stack esta lleno\n";  
            else stackPtr[++topPtr] = number;  
        };  
  
        void pop () {...};  
        int top () {...};  
        int empty () {...};  
};
```

Tipo de Dato Abstracto: Ejemplo 2 en C++

// Stack.h – Archivo de Header
de clase Stack

```
#include <iostream.h>
class Stack {
    private:
        int *stackPtr;
        int maxLen;
        int topPtr;
    public:
        Stack();
        ~Stack();
        void push(int);
        void pop();
        int top();
        int empty();
}
```

// Stack.cpp – archivo de implementacion de clase Stack

```
#include <iostream.h>
#include "Stack.h"
```

```
Stack::Stack() { /** Un constructor
    stackPtr = new int [100];
    maxLen = 99;
    topPtr = -1;
}
```

```
Stack::~Stack() {delete [] stackPtr;};
```

```
void Stack::push(int number) {
    if (topPtr == maxLen)
        cerr << "Error en push—stack lleno\n";
    else stackPtr[++topPtr] = number;
}
```

...

Tipo de Dato Abstracto: Java

Java es similar a C++, excepto:

- Todos los tipos definidos por el usuario son clases.
- Todos los objetos son asignados desde el Heap y accedidos mediante variables de referencia.
- Miembros de una clase tienen modificadores de control de acceso (*private* o *public*) en vez de cláusulas.
- Java tiene un segundo mecanismo de ámbito: paquete.

Tipo de Dato Abstracto: Ejemplo en Java

```
class StackClass {  
    private int [] stackRef;  
    private int maxLen, topIndex;  
    public StackClass() { // un constructor  
        stackRef = new int [100];  
        maxLen = 99;  
        topIndex = -1;  
    };  
    public void push (int num) {...};  
    public void pop () {...};  
    public int top () {...};  
    public boolean empty () {...};  
}
```

Tipo de Dato Abstracto: Python

Python provee soporte para definir clases y objetos. Sin embargo, la definición es bien relajada.

- Una clase se declara con la palabra reservada *class*.
- Tanto el constructor como los métodos deben llevar como primer parámetro el argumento *self*.
- Una instancia de una clase se realiza simplemente invocando el constructor.
 - *miObjeto = miClase(...)*
- Los métodos se invocan en forma similar al constructor.

Tipo de Dato Abstracto: Ejemplo en Python

```
class Stack:
    def __init__(self, max):
        self.maximo = max
        self.stack = [ ]

    def push(self, obj):
        if len(self.stack) < self.maximo:
            self.stack.append(obj)
            print("push", obj)
        else:
            print("stack overflow")

    def pop(self):
        if len(self.stack) == 0:
            print("stack is empty")
        else:
            print("pop", self.stack.pop())
```

Tipo de Dato Abstracto Parametrizado

DEFINICIÓN: un tipo de datos abstracto parametrizado permite diseñar un tipo que puede almacenar cualquier tipo de elemento, siendo este constructo de tipos sólo relevante para lenguajes estáticos.

- En orientación a objetos se les conoce también como “clases genéricas”
→ polimorfismo estático (o ad hoc)
- Soportado por lenguajes como Ada, C++, Ada y C#.

Tipo de Dato Abstracto Parametrizado: Ejemplo en C++

```
template <class Type>
class Stack {
private:
    Type *stackPtr;
    const int maxLen;
    int topPtr;
public:
    Stack(int size) // Constructor
    {
        stackPtr = new Type[size];
        maxLen = size - 1;
        topSub = -1;
    }
    ...
}
```

Instanciación: Stack<int> miIntStack;

Tipo de Dato Abstracto Parametrizado: Java

- Los usuarios pueden definir clases genéricas, donde los parámetros genéricos deben ser clases.
- No pueden almacenar datos primitivos.
- Evita tener diferentes tipos en la estructura y usar *casting* para objetos.
- Los tipos genéricos más conocidos son de tipo colección (paquete `java.util.Collection`), como *LinkedList* y *ArrayList*.
- Ejemplo:

```
ArrayList <Integer> miArreglo = new ArrayList <Integer> ();
```

```
miArreglo.add(0, 47); // agregar 47 en posición 0
```

Tipo de Dato Abstracto Parametrizado: Ejemplo en Java

```
import java.util.*;

public class Stack2<T> {
    private ArrayList<T> stackRef;
    private int maxLen;
    public Stack2(int size) {
        stackRef = new ArrayList<T> ();
        maxLen = size-1;
    }
    public void push(T val) {
        if (stackRef.size() == maxLen)
            System.out.println(" Error en push – stack lleno");
        else
            stackRef.add(val);
        ...
    }
}
```

Instanciación: Stack2<String> miStack = new Stack2<string> ();

5.2 Orientación a Objetos (visión general)

Introducción

CONCEPTOS CLAVES: la Programación Orientada a Objetos (POO) se fundamenta en tres características fundamentales:

- Tipo de Datos Abstracto
- Herencia
- Polimorfismo

Introducción

SABORES:

- Lenguajes OO puros (ej.: Smalltalk y Ruby)
- Lenguajes que soportan simultáneamente programación imperativa y POO - básicamente, son extensiones (ej.: ADA y C++)
- Lenguajes que no soportan otros paradigmas, pero usan su estructura imperativa (ej.: Java y C#)
- Lenguajes funcionales que soportan POO (ej.: CLOS)

Problemas de los TDA que ataca la OO (1/2)

- **Herencia**: permite extensión de datos u operaciones. Si se quiere reutilizar una componente de software a veces es necesario agregarle datos y nuevas operaciones.
- **Redefinición**: a veces se requiere redefinir el comportamiento específico de una operación (ej.: una “ventana de texto” tiene requerimientos específicos) o del estado (ej.: texto asociado).

Problemas de los TDA que ataca la OO (2/2)

- **Abstracción**: se requiere abstraer operaciones similares para diferentes componentes en un nuevo tipo (ej.: círculos y rectángulos tienen posición y son figuras que se pueden desplegar y trasladar).
- **Polimorfismo**: se requiere extender el tipo de datos sobre las cuales se pueden aplicar las operaciones. Existen diferentes tipos de polimorfismo (ej.: sobrecarga y tipos parametrizados).

Modelo Objetual (1/2)

- **Programa**: conjunto de objetos interactuantes.
- **Clase**: corresponde a una declaración de tipo, que especifica el estado y comportamiento que tendrán sus instancias u objetos.
 - El tipo queda definido por su interfaz, que especifica métodos y constantes.
 - La implementación del tipo queda especificado por las variables y código que define el comportamiento de los métodos y la manipulación de su estado

Modelo Objetual: ejemplo de una clase en Java

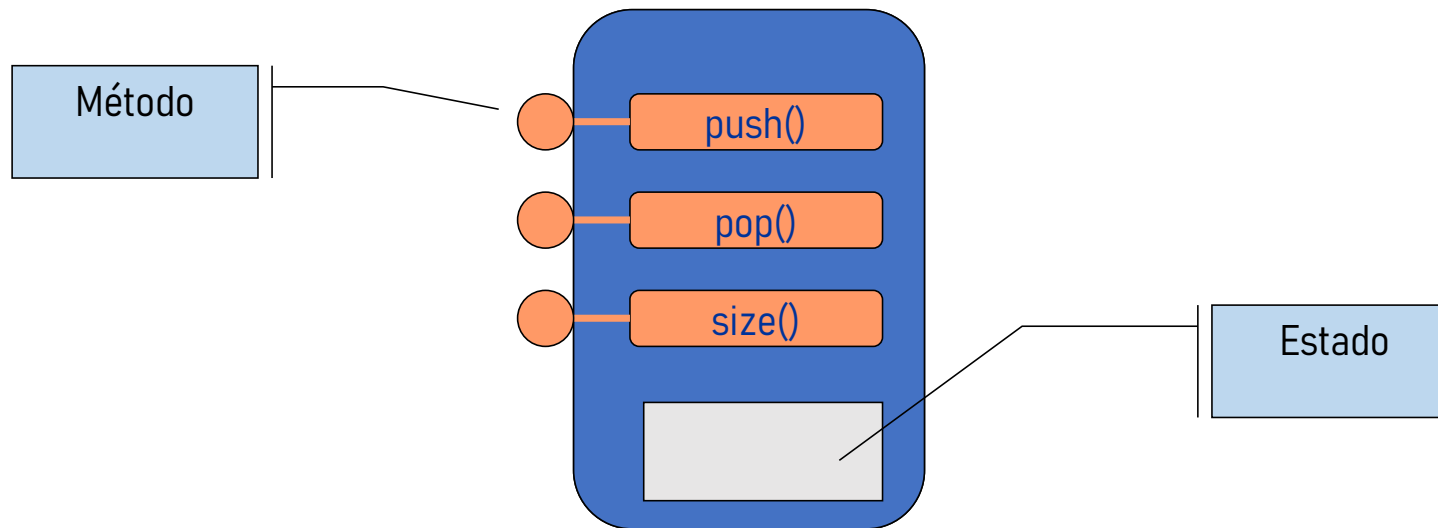
```
public class Stack {  
    private int Maximo;  
    private int top = -1;  
    private String[] buffer;  
  
    public Stack(int i) {  
        Maximo = i;  
        buffer = new String[i];  
    }  
  
    public void push(String nuevo) {  
        if (top < Maximo-1)  
            buffer[++top] = nuevo;  
        else    System.err.println("Oops, stack lleno");  
    }  
}
```

```
    public String pop() {  
        String respuesta = "";  
  
        if (top >= 0)  
            respuesta = buffer[top--];  
        else System.err.println("Oops, stack vacio");  
        return respuesta;  
    }  
  
    public int size() {  
        return top+1;  
    }  
}
```

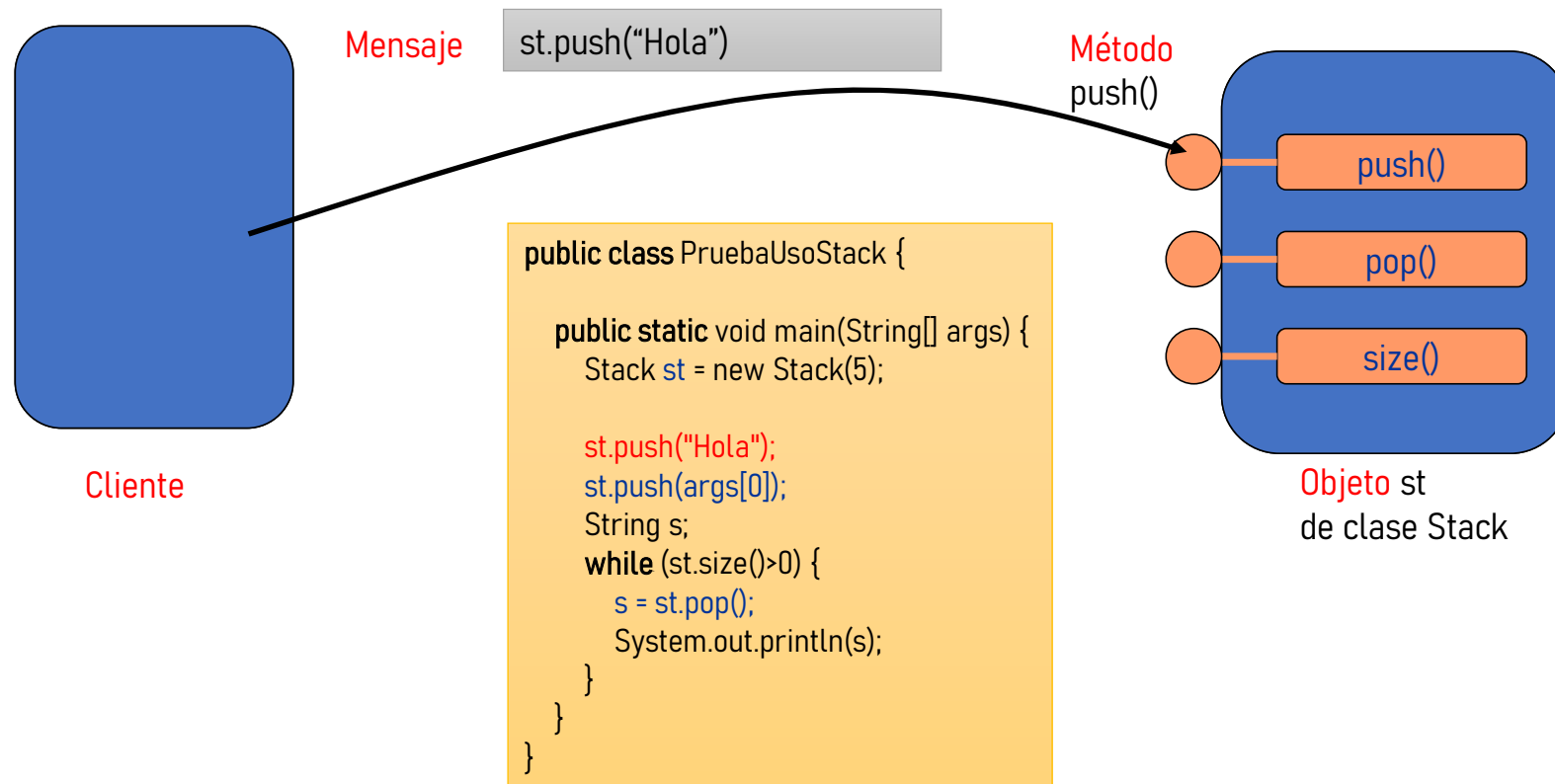
Modelo Objetual (2/2)

- **Objeto**: instancia concreta de una clase.
- **Método**: especifica el comportamiento de los operadores de una clase, y controla (encapsula) el acceso al estado. Conjunto de métodos define el protocolo para los mensajes.
- **Mensaje**: invocación de un método que contiene identificador del objeto y método (destino), y parámetros reales o resultados.

Modelo Objetual



Modelo Objetual

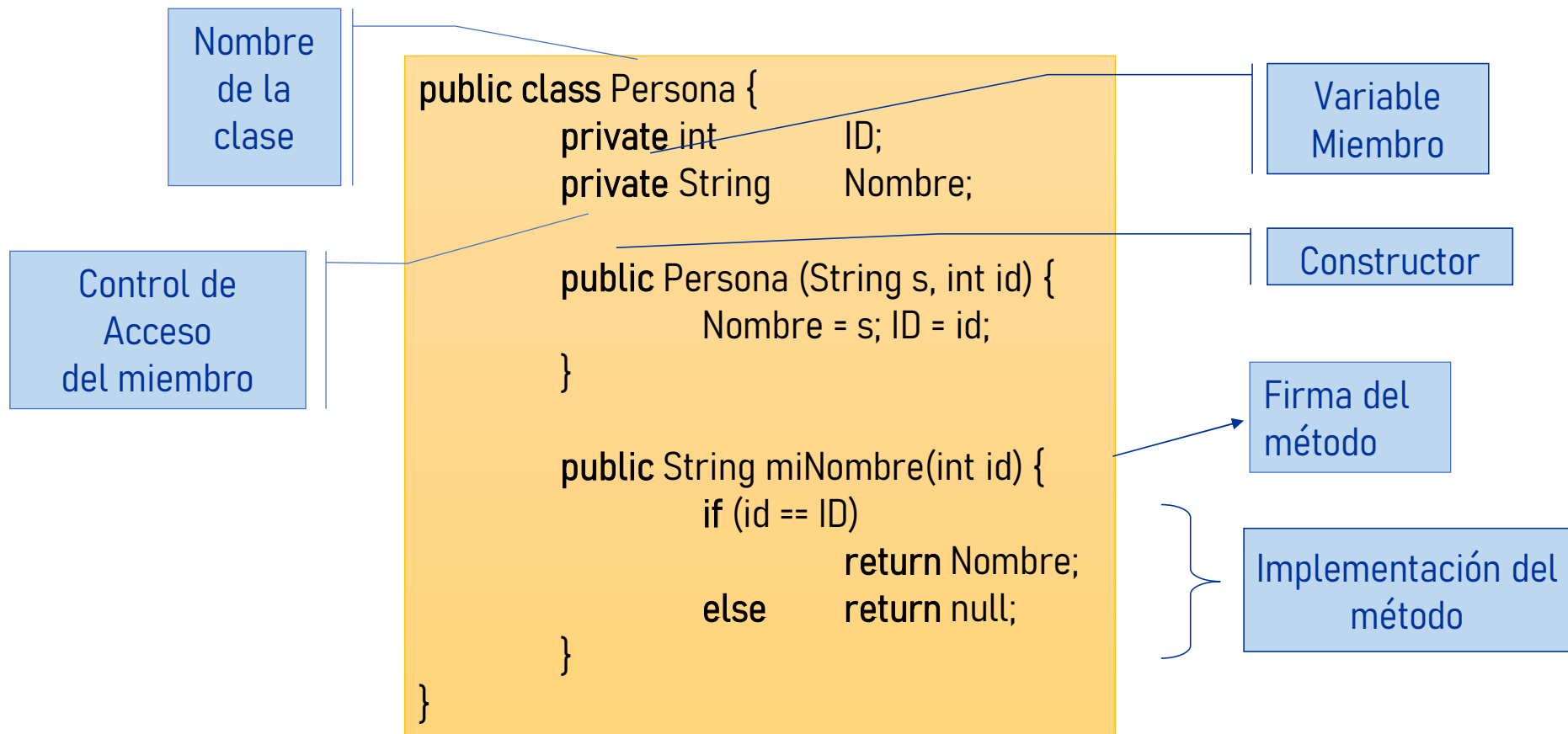


5.3 Clases y Objetos (en Java)

Conceptos Generales

- Las clases contienen los métodos que definen la computación.
- Los campos (*fields*) o variables miembros definen el estado.
- Las clases proveen la estructura para los objetos y el mecanismo para su creación.
- Un método tiene una firma o protocolo, pero su implementación define su semántica.
- Los métodos definen una suerte de contrato sobre lo que hace un objeto de la clase.

Estructura de una clase



Variables miembros

- Las variables de clases se denominan campos.
- Cada objeto de una clase tiene sus propias instancias de cada variable miembro, por lo que cada objeto tiene su propio estado.
- Variables miembros se pueden compartir entre todos los objetos de una clase con el modificador `static`, es decir todos los objetos de una clase comparten una única copia de un campo declarado como estático.
- En general, cuando se habla de variables y métodos miembros se refiere a aquellos no estáticos.

Variable Miembro Estática

Variable
Estática
(o de
Clase)

```
public class Persona {  
    private static int nextID = 0;  
    private int ID;  
    private String Nombre;  
  
    public Persona (String s) {  
        ID = ++nextID;  
        Nombre = s;  
    }  
  
    public int getID() {  
        return ID;  
    }  
}
```

Control de Acceso de Miembros

- 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: sólo accesibles por la propia clase.
 - Paquete: miembros sin modificador de acceso son sólo accesibles por código en el mismo paquete.
 - Protegido: accesibles por una subclase, como también por código del mismo paquete.
 - Público: accesibles por cualquier clase.

Creación de Objetos

```
Persona p1;  
Persona p2 = new Persona("Pedro");
```

- Se han declarado 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 *runtime* crea un objeto, asignando suficiente memoria, e inicializando el objeto con algún constructor.
- Si no existe suficiente memoria se ejecuta el recolector de basura; y si aún no existe suficiente memoria, se lanza una excepción del tipo `OutOfMemoryError`.
- Terminada la inicialización, el *runtime* retorna la referencia al nuevo objeto.

Constructores (1/2)

- Un objeto recién creado debe inicializar las variables miembro.
- Las variables miembro pueden ser inicializadas explícitamente en la declaración...pero hay veces que se requiere algo más, como por ejemplo el ejecutar código para abrir un archivo.
- Los constructores cumplen ese propósito; tienen el mismo nombre de la clase y pueden recibir parámetros.
- Los constructores no son un método (no retornan un valor).
- Una clase puede tener varios constructores.

Constructores (2/2)

- Algunas clases no tienen un estado inicial razonable sin inicialización (ej.: `tope = -1`).
- Proveer un estado inicial conveniente y razonable (ej.: `Nombre = "Juan"`).
- Construir un objeto puede ser costoso (ej.: objeto que crea una tabla interna a la medida).
- Por motivos de seguridad de acceso (ej.: constructores pueden ser privados o protegidos).

Constructores: ejemplo

Constructor

```
public class Persona {  
    private static int nextID = 0;  
    private int ID;  
    private String Nombre;  
  
    public Persona (String s) {  
        ID = ++nextID;  
        Nombre = s;  
    }  
  
    public int getID() {  
        return ID;  
    }  
}
```

Constructores: Control de Acceso

- **Private:** ninguna otra clase puede instanciar la clase (ej.: Clase fábrica).
- **Protected:** sólo la clase, subclases de la clase y clases del mismo paquete pueden crear instancias.
- **Public:** cualquier clase puede crear una instancia u objeto.
- ***Sin Especificador*** (acceso de paquete): sólo clases del mismo paquete y de la misma clase pueden crear instancias.

Constructores: por defecto (omisión)

- Si no se define constructor, se asume un constructor sin argumentos, que no hace nada.
- Este **constructor por defecto** se asume que siempre existe, en el caso de no haberse definido un constructor sin argumentos
- El constructor por defecto es público si la clase lo es, sino no lo es.

Constructores: múltiples

```
public class Persona {  
    private static int cantidad;  
    private int ID;  
    private String Nombre;  
    private int edad = -1;  
  
    static { cantidad = 0; }  
  
    public Persona (String s) {  
        cantidad++;  
        Nombre = s;  
    }  
  
    public Persona(String s, int ed) {  
        this(s);  
        edad = ed;  
    }  
  
    //... Otros métodos  
}
```

Segundo
Constructor

Invoca
Constructor
anterior

Métodos

- Un método se entiende normalmente para manipular el estado del objeto (variables miembro).
- Algunas clases tienen variables miembro públicas, lo que permite su manipulación directa por otros objetos, pero no es seguro.
- Cada método se llama con la sintaxis: `referencia.metodo(args)`.

Métodos: Valores de Parámetros

- En Java todos los parámetros se pasan por valor (tanto primitivos como referencias), es decir se pasa una copia del dato al método.
- Si el parámetro es una referencia, es ésta la que se copia, no el objeto... por lo tanto, un cambio de estado del objeto referenciado realizado por el método será visible después del retorno.

Métodos: Control de Acceso

- Para proteger campos de un acceso externo se usa el modificador `private`.
- Java no provee un modificador que sólo permita lectura de un campo.
- Para asegurar acceso de sólo lectura se debe introducir un nuevo método que lo haga indirectamente y en forma segura, denominado `accesor`.
- Es recomendable usar accesorios para proteger el estado de los objetos.

Métodos: Ejemplo de método accesor

Método
accesor

```
public class Persona {  
    private static int    nextID = 0;  
    private      int      ID;  
    private      String   Nombre;  
    private      int      edad = -1;  
  
    public int getID() {  
        return ID;  
    }  
  
    public String getNombre() {  
        return Nombre;  
    }  
  
    //... Otros métodos  
}
```

Métodos: Sobrecarga

- Cada método tiene una firma, compuesta por el nombre del método, cantidad y tipos de los parámetros, y el tipo de retorno.
- Java permite tener varios métodos con un mismo nombre, pero con diferentes parámetros...esto se denomina sobrecarga.
- Cuando se invoca un método sobrecargado, se calza por cantidad y tipo los parámetros usados (reales), para seleccionar el adecuado.

Referencia this

- Sólo se puede usar en un método no estático, y se refiere al objeto actual sobre el que se invocó el método.
- Implícitamente se usa `this` al comienzo de cada miembro propio referenciado por el código de la clase.
- Ejemplo: para el caso anterior



Referencia this

- Se usa usualmente para pasar una referencia del objeto actual como parámetro a otro método. Ej.: `Timer.wakeup(this, 30);`
- Otro uso es cuando existe ocultamiento de un identificador debido a colisión de nombres. Ej.:

```
public SetEdad (int edad) {  
    this.edad = edad;  
}
```

- Similarmente a lo anterior, para tener acceso a miembros ocultos de una superclase (por redefinición: overridden), se puede usar `super`.

Miembros Estáticos

- Un miembro estático es único para toda una clase, en vez de tener uno por objeto.
- **Variable miembro estática**: existe una única variable para todos los objetos de una clase, y se deben inicializar antes de que se use cualquier variable estática o método de la clase.
- **Método estático** (o de clase): típicamente, se invoca usando el nombre de toda la clase, pudiendo acceder sólo a miembros estáticos (ej.: no permite usar *this*).

Miembros Estáticos: Ejemplo

Método
estático

Clase
System,
miembro
estático **out**

```
public class PruebaUsoStack {  
    public static void main(String[] args) {  
        Stack st = new Stack(5);  
  
        st.push("Hola");  
        st.push(args[0]);  
        String s;  
        while (st.size()>0) {  
            s = st.pop();  
            System.out.println(s);  
        }  
    }  
}
```

Bloques de Inicialización Estática

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

Inicialización
estática

```
public class Primos {  
    protected static int[] PrimosConocidos = new int[4];  
  
    static {  
        PrimosConocidos[0] = 2;  
        for (int i = 1; i < PrimosConocidos.length; i++)  
            PrimosConocidos[i] = ProximoPrimo();  
    }  
}
```

Clases Anidadas (1/2)

- Java permite definir una clase como miembro de otra clase.
- Útil cuando una clase sólo tiene sentido en el contexto específico de una clase.
- Una clase anidada tiene acceso total a todos los miembros de la clase a que pertenece (como lo haría cualquier otro método miembro de la clase superior).

Clases Anidadas (2/2)

- Si se declara estática, se llama clase anidada estática.
 - No tiene acceso a miembros no estático.
 - Permite sólo relacionar clases (no instancias).
- Si no es estática, es una clase interna.
 - No puede definir miembros estáticos.
 - Permite crear objetos de la clase interna dentro de la clase a que pertenece.
 - Permite relacionar objetos de las clases relacionadas.

Clases Anidadas: Ejemplo de clase anidada

```
public class Stack {  
    private Vector items;  
  
    // código de métodos y constructores  
    // de Stack  
  
    public Enumeration enumerator() {  
        return new StackEnum();  
    }  
}
```

```
class StackEnum implements Enumeration {  
    int itemActual = items.size() - 1;  
  
    public boolean hasMoreElements() {  
        return (ItemActual >= 0);  
    }  
  
    public Object nextElement() {  
        if (!hasMoreElements())  
            throw new NoSuchElementException();  
        else  
            return items.elementAt(ItemActual--);  
    }  
}  
  
} // fin de declaración de Stack
```

Recolección de Basura

- Java realiza recolección automática de basura, es decir el programador no requiere liberar explícitamente los objetos
 - Sólo existe `new` (no `delete` como C++).
 - Objeto que se detecta sin referencia, el sistema lo libera.
- Ventajas:
 - No requiere invocar al administrador de memoria para liberar cada objeto.
 - No se produce el problema de *dangling*.

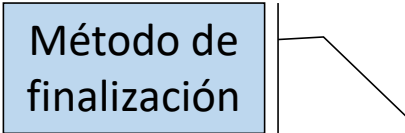
Método finalize

- Permite ejecutar un método de finalización antes de liberar memoria.
- Es útil cuando se usan recursos que no son de Java y que deben liberarse (evitar fuga de recursos). Ej.: cerrar archivos abiertos.
- Forma de declararlo:

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


Método finalize: Ejemplo

Método de
finalización



```
public class ProcesarArchivo {  
    private Stream Archivo;  
  
    public ProcesarArchivo (String nombre) {  
        Archivo = new Stream(nombre);  
    }  
    // ...  
    public void close() {  
        if (Archivo != null) {  
            Archivo.close();  
            Archivo = null;  
        }  
  
    protected void finalize() throws  
        Throwable {  
            super.finalize();  
            close();  
        }  
}
```

Método main

- El método `main` se debe encontrar en cada aplicación Java (única vez).
- El método debe ser público, estático y void.
- El argumento único es un arreglo de *String*, que se consideran los argumentos del programa.

```
class Echo {  
    public static void main (String[] args) {  
        for (int i = 0; i < args.length; i++)  
            System.out.print(args[i] + " ");  
        System.out.println();  
    }  
}
```

Archivo fuente Echo.java

```
>java Echo que argumentos recibe  
que argumentos recibe  
>
```

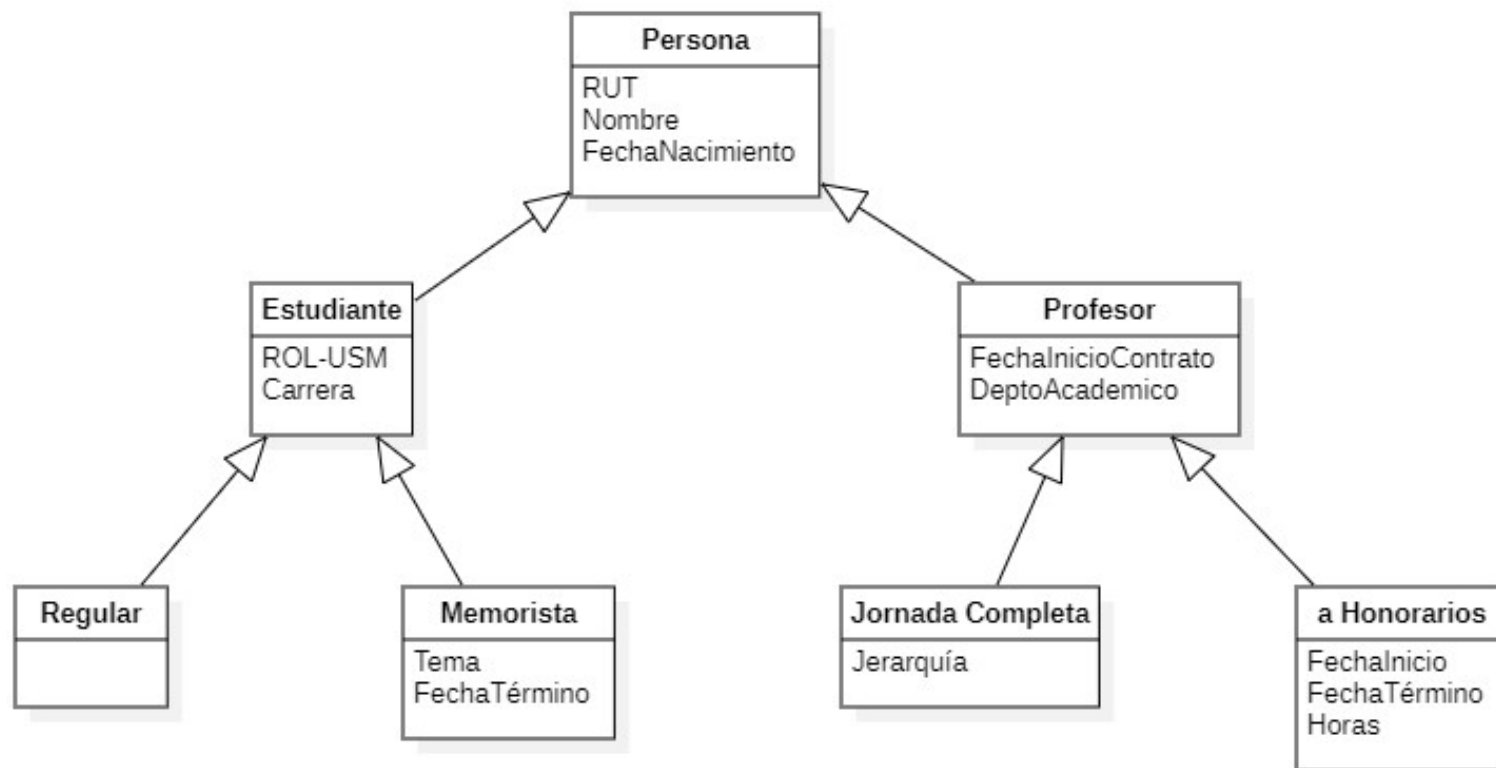
Ejecución del programa

5.4 Herencia

Herencia

- **DEFINICIÓN:** permite reutilizar tipos o clases ya definidos o existentes, definiendo a partir ellos nuevos tipos o clases relacionadas, para así mejorar la productividad en el desarrollo de software y organizar conceptos relacionados.
- Se definen relaciones de herencia para clases, pero también para interfaces.
- Herencia define una jerarquía de relaciones.
 - Se dice que una definición se deriva (o hereda) de la base.
 - La derivación es un subtipo o subclase, y donde proviene se le llama el supertipo o superclase.

Herencia: Ejemplo



Herencia

- En general, una clase define un contrato de uso a través de sus miembros accesibles y el comportamiento esperado de los métodos.
- Una clase extendida agrega funcionalidad, creándose una nueva clase, con un contrato extendido.
 - No se cambia el contrato que se hereda de la superclase, sino que se extiende.
 - Se puede cambiar la implementación de algunos métodos heredados, pero respetando el contrato de la superclase.

Herencia

- Las subclases proveen comportamiento especializado basada en la superclase, representando un mecanismo de reutilización de interfaces y código.
- Se pueden definir clases abstractas, que especifican un comportamiento genérico, que luego debe ser específicamente definido por las clases herederas (que lo implementan).

Herencia: Principio de Sustitución

- Donde se espera una referencia a una superclase A, en su lugar se puede utilizar una referencia a una subclase B que la extienda directa o indirectamente – esto no funciona en sentido contrario.
- Ejemplo:

A refA; //A es una clase
B refB; //B subclase de A

Es correcto:

refA = new A(...);
refA = new B(...);

No es correcto:

refB = new A(...);

¿Cuándo y cómo extender Clases?

- Una clase que se extiende sigue siendo del tipo de la superclase (**isA**), no viceversa. Ej.: si pixel deriva de punto, entonces un pixel es un punto, pero un punto no es un pixel.
- A veces se extiende agregando nuevos miembros (**hasA**). Ej.: un círculo tiene un punto, pero no es un punto.
- En casos que la extensión permite tener diferentes roles es más conveniente usar agregación. Ej.: una persona que tiene roles diferentes no es simple extenderla por derivación.

Herencia en Java

- Todas las clases son extendidas, aún cuando no lo sean explícitamente, pues derivan directa o indirectamente de la clase *Object*.
- Si no se especifica *extends* se supone que se deriva de *Object*.
- La clase *Object* implementa el comportamiento que requiere todo objeto Java.
- Sólo permite tener una única superclase (herencia simple).

Herencia en Java: Ejemplo

DEFINICIÓN

```
public class StackPrint extends Stack {  
  
    public StackPrint(int i) {  
        super(i);  
    }  
  
    public void print() {  
        System.out.println("tamaño: " + (top+1));  
        for (int i=0; i<top+1; i++) {  
            System.out.println(i + ": " + buffer[i]);  
        }  
    }  
}
```

USO

```
public class Prueba {  
  
    public static void main(String[] args) {  
  
        StackPrint stp = new StackPrint(5);  
        stp.push("Hola-print");  
        stp.push(args[0]);  
        stp.print();  
    }  
}
```

Herencia en Java: Reglas

- Una subclase hereda todos los miembros protegidos y públicos de la superclase.
- La subclase hereda los miembros sin modificador (paquete), mientras la subclase sea declarada en el mismo paquete.
- La subclase no hereda los miembros de la superclase que tienen el mismo nombre de un miembro de la subclase.
- En el caso de una variable miembro, ésta se oculta.
- En el caso de un método, éste es redefinido o sobrescrito.

Herencia: Redefinición de Variables Miembros

- Los campos no se pueden realmente redefinir, sólo pueden ser “ocultados”.
- Si se declara un campo con el mismo nombre de uno de la superclase, este último sigue existiendo, pero no es accesible directamente desde la subclase.
- Para accederlos se puede usar referencia **super**.

```
public class SuperClase {  
    Number numero;  
}
```

```
public class SubClase extends SuperClase {  
    Float numero;  
}
```

Acceso a SuperClase en SubClase:
super.numero

Herencia: Redefinición de Métodos (1/2)

Para la definición de métodos donde existe coincidencia de nombres:

- **Sobrecarga**: se define más de un método con el mismo nombre, pero con diferente firma.
- **Redefinición**: se reemplaza la implementación de la superclase.
 - Firma y tipo del retorno deben ser idénticas.
 - Permite modificar o complementar el comportamiento de un método de la superclase.
 - Se puede invocar el método redefinido desde la subclase con la referencia `super`.

Herencia: Redefinición de Métodos (1/2)

- Un método redefinido puede usar modificadores de acceso, pero sólo puede dar más acceso, no menos. Ej.: un método público de la superclase no se puede declarar privado o protegido en la subclase.
- Métodos que no se pueden redefinir:
 - Métodos finales.
 - Métodos estáticos (de la clase): sí se pueden ocultar declarando un método con la misma firma.
- Métodos que se deben redefinir: aquéllos que han sido declarados abstractos en la superclase.

Herencia: Ejemplo de Redefinición de Miembros

```
public class SuperClase {  
    public String str = "SuperClase.str";  
  
    void mostrar() {  
        System.out.println(  
            "SuperClase.mostrar = " + str);  
    }  
}
```

```
public class SubClase extends SuperClase {  
    public String str = "SubClase.str";  
  
    void mostrar() {  
        System.out.println(  
            "SubClase.mostrar = " + str);  
    }  
}
```

```
public class Principal {  
    public static void main(String[] args) {  
        SubClase sub = new SubClase();  
        SuperClase sup = sub;  
        sup.mostrar();  
        sub.mostrar();  
        System.out.println("sup.str = " + sup.str);  
        System.out.println("sub.str = " + sub.str);  
    }  
}
```

¿ Resultado de Ejecución ?

```
SubClase.mostrar = SubClase.str  
SubClase.mostrar = SubClase.str  
sup.str = SuperClase.str  
sub.str = SubClase.str
```


Clases extendidas: Constructores

- Al crear un objeto, el orden de ejecución es el siguiente:
 - Se invoca el constructor de la superclase (sino se especifica cuál, se usa el por omisión).
 - Se inicializan los campos usando sentencias de inicialización.
 - Se ejecuta el cuerpo del constructor.
- Si se quiere usar un constructor específico de la superclase, debe invocarse con `super(...)`.

Clases extendidas: Constructores - ejemplo

```
public class Superclass {  
    protected int X = 1;  
    protected int Y;  
  
    public Superclass() {  
        System.out.println("superclass: antes Y = " + Y);  
        Y = X;  
        System.out.println("superclass: despues Y = " + Y);  
    }  
  
    public int getY() {  
        return Y;  
    }  
}
```

Inicio de principal
superclass: antes Y = 0
superclass: despues Y = 1
subclass antes: Y = 1
subclass despues: Y = 3
Fin de principal

```
public class Subclass extends Superclass {  
    private int Z = 3;  
    public Subclass() {  
        System.out.println("subclass: antes Y = " + Y);  
        Y = Z;  
        System.out.println("subclass: despues Y = " + Y);  
    }  
}  
  
public class Principal {  
    public static void main(String[] args) {  
        System.out.println("Inicio de principal");  
        Subclass z = new Subclass();  
        System.out.println("Fin de principal");  
    }  
}
```

¿ Resultado de Ejecución ?

Clases y Métodos Abstractos (1/2)

- Permite definir clases que definen sólo parte de su implementación
 - Clases derivadas deben implementarlas
- Abstracción es útil cuando:
 - Algún comportamiento es verdad para la mayoría de los objetos de un tipo dado,
 - pero algún comportamiento sólo tiene sentido para algunos, no para toda la superclase.

Clases y Métodos Abstractos (2/2)

- Un método abstracto es uno que no está implementado (sólo define el protocolo de los mensajes).
- Una clase abstracta es aquella que incluye al menos un método abstracto.
- Una clase abstracta no permite instanciar objetos, pues su definición no está completa (parcialmente implementada).

Clases y Métodos Abstractos: Ejemplo

- Una clase abstracta no requiere declarar métodos abstractos, pero una clase que tiene un método abstracto debe ser declarada como clase abstracta.

```
abstract public class Figura {  
    int x, y;  
    ...  
    void mover(int nuevoX, int nuevoY) {  
        ...  
    }  
  
    abstract public void dibujar();  
}
```

```
public class Triangulo extends Figura {  
    void dibujar() {  
        ...  
    }  
}  
  
public class Cuadrado extends Figura {  
    void dibujar() {  
        ...  
    }  
}
```

Polimorfismo: Tipos

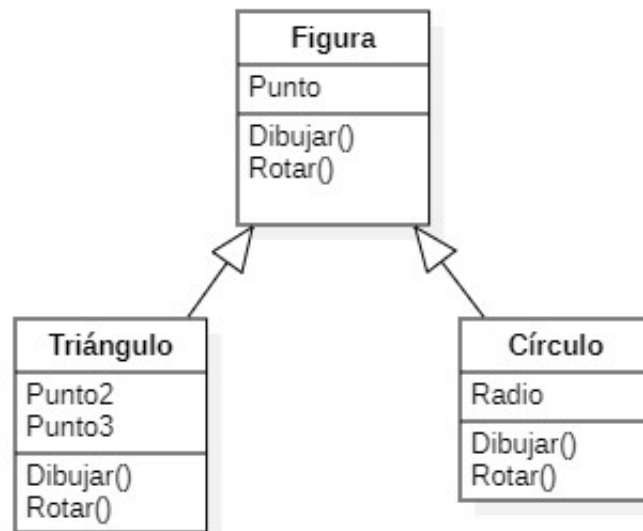
Tipos:

- Polimorfismo estático (o polimorfismo *ad hoc*): es aquél en el que los tipos a los que se aplica el polimorfismo deben ser explicitados y declarados uno por uno antes de poder ser utilizado. Ej.: sobrecarga de operadores y de subprogramas; subprogramas o clases genéricas..
- Polimorfismo dinámico (o polimorfismo paramétrico): es aquél en el que el código no incluye ningún tipo de especificación sobre el tipo de datos sobre el que se trabaja. Así, puede ser utilizado a todo tipo de datos compatible. Ejs.: ligado dinámico de métodos.

Ligado Dinámico de Métodos

- Permite crear variables del tipo base que pueden referenciar objetos de cualquier tipo de descendiente (polimorfismo).
- En una jerarquía de clases que incluye subclases que redefinen el comportamiento del método de un antecesor.
- Cuando se invoca un método redefinido, dinámicamente se debe hacer el ligado al método que corresponda.
- Permite a sistemas de software ser más fácilmente extendidos durante el desarrollo y mantenidos.

Ligado Dinámico de Métodos: Ejemplo



OBSERVACIÓN:

Una variable de tipo **Figura** también puede ser de tipo **Triángulo** y **Cuadrado**.

Ligado Dinámico de Métodos: Ejemplo en C++

```
public class Figura {  
    // ...  
    public:  
        public abstract void dibujar(void);  
        public abstract void rotar(int);  
        // ...  
};  
  
class Circulo: public Figura {  
    real radio;  
    public:  
        void dibujar(void);           // anula figura::dibujar  
        void rotar(int) {}           // anula figura::rotar  
        Circulo(punto p, real radio); // constructor  
};
```

```
Figura *f[100];  
...  
for (int i=0; i<n; i++)  
    f[i]->dibujar();  
...
```

Herencia y Asignación de Memoria (1/2)

- Si no existe herencia y los objetos se comportan como TDA, es simple, se puede usar indistintamente cualquier tipo de memoria:
 - Creación desde el *Stack* cuando se alcance su declaración.
 - Creación explícita desde el *Heap* con operador (ej.: *new*).
- Restringir sólo al Heap tiene la gran ventaja de tener un único método uniforme de creación dinámica de objetos.
 - Se pueden utilizar referencias con dereferenciación automática, simplificando la sintaxis de acceso.

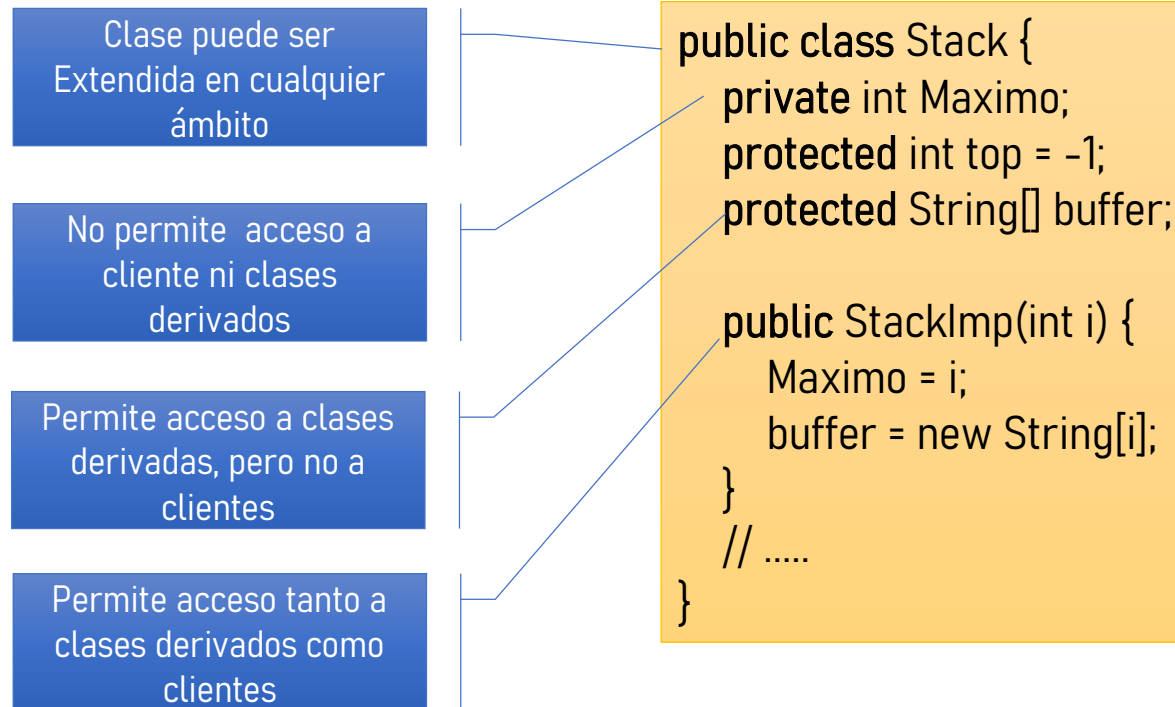
Herencia y Asignación de Memoria (2/2)

- Con herencia, la asignación dinámica de *Stack* tiene un problema:
 - A una variable de tipo base se le pueden asignar referencias de cualquier tipo derivado
 - Dificultad para analizar estáticamente el tamaño de memoria requerido en el Stack dinámico.
- ¿Liberación de memoria explícita o implícita?

Control de Acceso

- La herencia se puede complicar en el control de acceso de las entidades encapsuladas en una clase y sus descendientes:
 - Una clase puede esconder entidades a sus subclases.
 - Una clase puede esconder entidades a sus clientes.
 - Una clase puede esconder entidades a sus clientes, pero haciéndolas visibles a sus subclases.
- Una subclase podría modificar el comportamiento de un método heredado de la clase base.
- En Java se usan los calificativos *public*, *private* y *protected* para definir la semántica de acceso a entidades de una clase en una relación de herencia y con sus clientes.

Control de Acceso

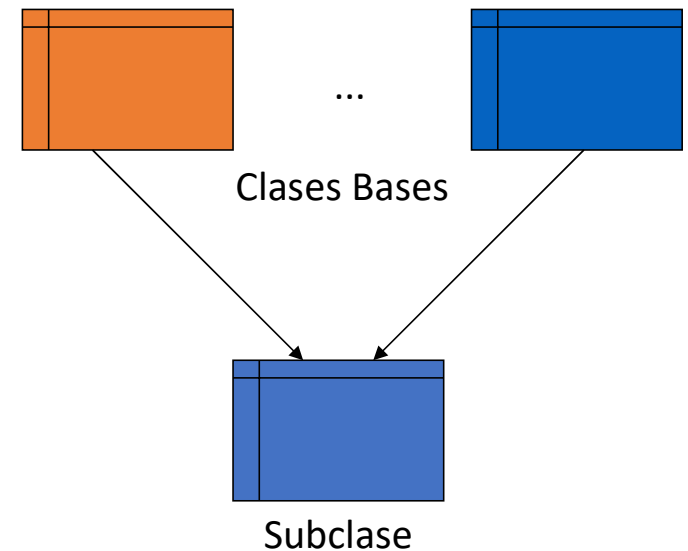


Modificadores

- **public**: por defecto u 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 (implícitamente, todos sus métodos son *final* también).
 - Obs.: también se puede usar con variables para no modificar su valor (constante) y en métodos para que no sean redefinidos.

Herencia Múltiple

- Se presenta cuando una clase extiende a más de una clase.
- Problemas que presenta:
 - Colisión de nombres introduce una complicación.
 - Pérdida de eficiencia por su mayor complejidad (en el ligado de los métodos).
 - No está claro que su uso mejore el diseño y mantención de sistemas, considerando la mayor complejidad de organización.
- Obs.: a nivel de clases disponible en C++, no en Java; éste la aplica con las interfaces.



Clase Object

- Todas las clases tienen a *Object* como raíz, por lo tanto heredan sus métodos.
- Considera dos tipos de métodos: de utilidad general y para soporte de hebras (threads) – en el curso, sólo veremos el primer grupo.

Clase Object: Métodos de Utilidad General (1/3)

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

```
public class Igualdad {  
  
    public static void main(String[] args) {  
        Integer Un = new Integer(1), otroUn = new Integer(1);  
  
        if (Un.equals(otroUn))  
            System.out.println("objetos son iguales");  
        else  
            System.out.println("objetos son diferentes");  
    }  
}
```

Clase Object: Métodos de Utilidad General (2/3)

- `protected Object clone()`: retorna un clon del objeto (una copia).
 - En principio, cambios posteriores en el clon no afectan al estado del objeto original.
 - Una clase que permite clonar objetos normalmente implementa la interfaz *Cloneable*.
 - Aquellas clases que tienen variables miembros que son referencias deben redefinir el método *clone()*.

```

public class Stack implements Cloneable {
    private Vector items;

    // código de los métodos y constructor de Stack

    protected Object clone() {
        try {
            Stack s = (Stack) super.clone();           // clona el stack
            s.items = (Vector) items.clone();          // clona el vector
            return s;                                  // return the clone
        } catch (CloneNotSupportedException e) { // esto no debiera suceder dado que el Stack es clonable
            throw new InternalError();
        }
    }
}

```

Clase Object: Métodos de Utilidad General (3/3)

- `public int hashCode()`: retorna código *hash* del objeto, que es usualmente único para cada objeto. Se usa en tablas de *hash*.
- `public final Class getClass()`: retorna un objeto de tipo `Class` que representa la clase del objeto *this*.
- `public void finalize()`: finaliza un objeto durante la recolección de basura.
- `public String toString()`: retorna una representación del objeto en un `String`, que la mayor parte de las clases redefinen.

5.5 Interfaces

Interfaz

- **DEFINICIÓN:** define un protocolo de comunicación para la interacción entre objetos, definiendo por lo tanto un tipo de datos sin necesidad de conocer la clase que lo implementa (especie de polimorfismo).
 - Una o más clases pueden implementar una misma interfaz.
 - Una clase que implementa una interfaz debe necesariamente implementar cada método de la interfaz.

Interfaz

- Es una manera de declarar tipos consistentes sólo de métodos abstractos y constantes.
- Las interfaces son útiles para el diseño.
 - Clases deciden cómo implementarlas.
 - Una interfaz puede tener muchas implementaciones.
 - Importante: una vez declarada una interfaz, no debiera ser modificada.

Interfaz

Ventajas:

- Captura similitudes entre clases no relacionadas, sin forzar artificialmente una relación entre ellas.
- Declara métodos que una o más clases esperan implementar.
- Revela una interfaz de programación sin revelar las clases que la implementan.
- Útil para definir una API.

Cuerpo de la Interfaz

- Todas las constantes son implícitamente:
 - públicas,
 - estáticas y
 - finales.
- Todos los métodos son implícitamente:
 - públicos y
 - abstractos
- No se aceptan otros modificadores (ej.: *private*, *protected* y *synchronized*)

Interfaz: Ejemplo de declaración en Java

```
public interface <nombre interfaz> extends <lista de super-interfaces> {  
    <cuerpo de la interfaz>  
}
```

- **public** hace pública la interfaz fuera del paquete (opcional).
- La interfaz puede extender varias interfaces (opcional), lo que define una herencia múltiple.

Interfaz: Ejemplo de Uso en Java

```
public interface StackInterface {  
    void push(String s);  
    String pop();  
    int size();  
}
```



```
public class StackImp implements StackInterface {  
    private int Maximo;  
    protected int top = -1;  
    protected String[] buffer;  
  
    public StackImp(int i) {  
        Maximo = i;  
        buffer = new String[i];  
    }  
    // ..... al menos: push(), pop(), size() !!!!  
}
```

Interfaz: otro Ejemplo de Uso en Java

```
public class Demolterador implements java.util.Iterator {  
    private int actual = 0;  
    private String[] Cartas = {  
        "2", "3", "4", "5", "6", "7", "8", "9",  
        "10", "Sota", "Reina", "Rey", "As" };  
  
    public boolean hasNext() {  
        if (actual == Cartas.length) {  
            return false;  
        } else {  
            return true;  
        }  
    }  
}
```

```
    public Object next() {  
        return (Object) Cartas[actual++];  
    }  
    public void remove() {  
        throw new UnsupportedOperationException();  
    }  
  
    public static void main(String[] args) {  
        Demolterador i = new Demolterador();  
        while (i.hasNext()) {  
            System.out.println(i.next());  
        }  
    }  
}
```

Interfaz: vs. Clase Abstracta

- Interfaz es una simple lista de métodos abstractos (no implementados)
- Una interfaz se diferencia de una clase abstracta en:
 - Una interfaz no puede implementar ningún método, una clase abstracta si lo puede.
 - Una clase puede implementar varias interfaces, pero puede tener una única superclase.
 - Una interfaz no puede ser parte de una jerarquía de clases.
- Clases no relacionadas pueden implementar una misma interfaz.

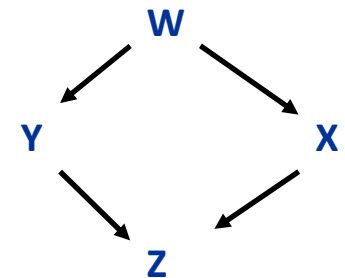
Interfaz: Principio de Sustitución

- Donde se espera una referencia a una interfaz A, en su lugar se “debe” utilizar una referencia a una clase B que la implementa directa o indirectamente – esto no funciona en sentido contrario.
- Ej.: si B es una clase que implementa la interfaz A, entonces es válido en la invocación de un método *f* con la firma *void f(A a)*; considerar:

```
B refB = new B(...);  
...f(refB);
```

Herencia Simple vs. Múltiple

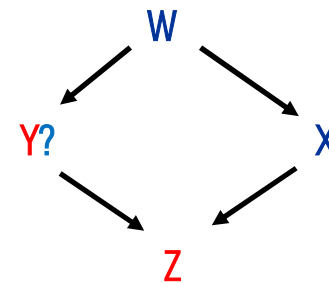
- Una **clase** puede extender exactamente una sola **superclase**, una **interfaz** puede extender múltiples interfaces.
- Herencia múltiple con clases puede producir problemas debido a diferentes implementaciones.
- Ejemplo:
 - **W** tiene un campo **wVar**
 - Si **Z** hace referencia a **wVar**, ¿cuál es? ...existen dos **wVar**?
- Java evita el problema permitiendo sólo herencia simple para clases.



Herencia Simple vs. Múltiple

```
interface W {...}  
interface X extends W {...}  
class Y implements W {...}  
class Z extends Y implements X {...}
```

```
interface W {...}  
interface X extends W {...}  
interface Y extends W {...}  
class Z implements X, Y {...}
```



Herencia Múltiple: Resolución de Conflictos

- ¿Qué pasa si un mismo nombre se encuentra en más de una interfaz de los supertipos?
 - Si las firmas no son idénticas, es trivial (se sobrecargarán diferentes métodos).
 - Si tienen firma idéntica, entonces la clase tendrá un solo método con esa firma.
 - Si las firmas sólo difieren en el tipo de retorno, no se puede implementar la interfaz.
- Para constantes es simple: se selecciona cada una usando nombre de la interfaz
 - Ejemplo: X.var, Y.var

5.6 Manejo de Excepciones

Motivación

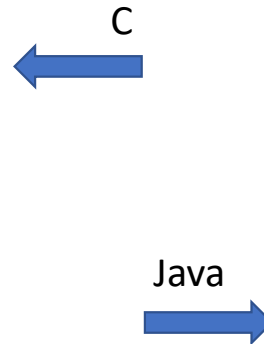
- Cuando ocurren errores es importante que un programa sea capaz de “capturar” el evento.
- Ejemplos de acciones posibles:
 - Notificar al usuario de un error
 - Guardar el trabajo hecho
 - Volver a un estado seguro anterior
 - Terminar limpiamente el programa
- Ejemplos de errores:
 - Error en la entrada de datos
 - Error en un dispositivo (ej.: Impresora apagada, disco lleno)
 - Error en el código

Excepciones: Concepto

- Una *excepción* corresponde a un evento que interrumpe el flujo normal de ejecución.
- Cuando ocurre tal tipo de evento en un método, se *lanza* (**throw**) una excepción, creando un objeto especial cuya referencia se pasa al *runtime* para manejar la excepción.
- El *runtime* busca en el stack el método que maneje el tipo de excepción:
 - Si se encuentra, se captura la excepción invocando al *manejador de la excepción* (**catch**)
 - Si no se encuentra se termina el programa
- Observación: aunque se está estudiando en la unidad de OO, mencionar que las excepciones no son exclusivas de este paradigma.

Excepciones: “comparación” entre C y Java

```
TipoError leerArchivo {  
    int errorCode = 0;  
    int fd;  
    int n;;  
  
    if ((fd=open("miArchivo") < 0) {  
        return ERROR_ABRIR_ARCHIVO;  
  
    int n= largo de archivo;  
    if (n < 0) {  
        close(fd);  
        return ERROR_LARGO_ARCHIVO;  
    }  
  
    while ((n = read(fd, ...) ) > 0) {  
        procesar datos leídos,  
    }  
  
    if (n != EOF)  
        return ERROR_LECTURA_ARCHIVO  
}
```

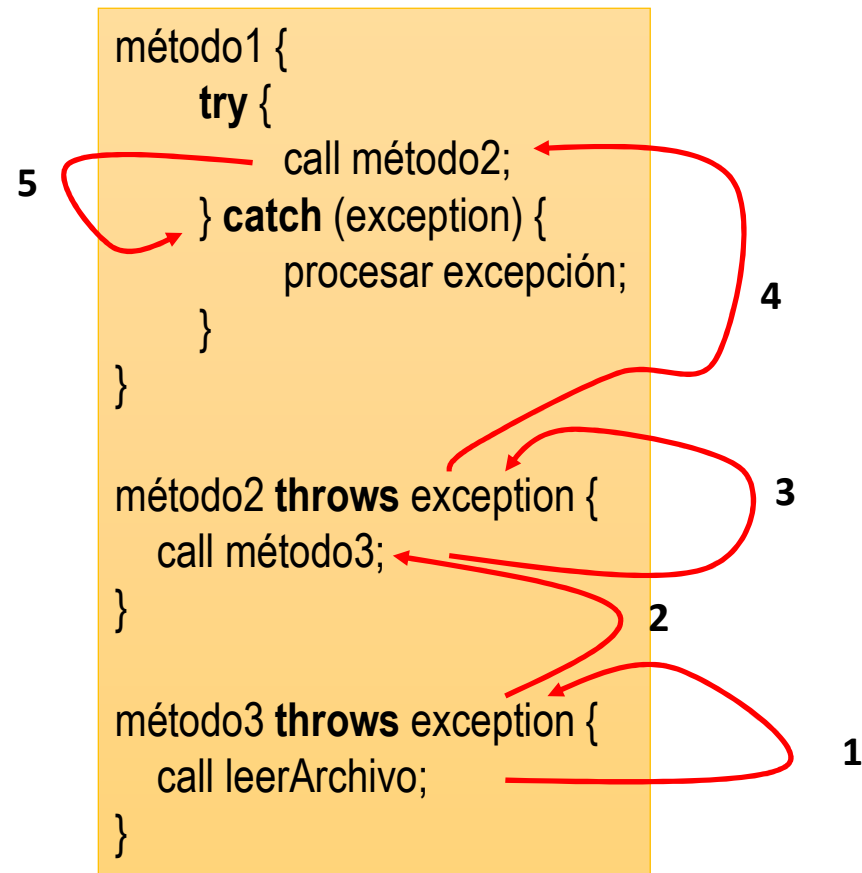


```
import java.io.*;  
  
class ejemplo {  
    ....  
    public void leerArchivo() throws Exception {  
        try {  
            File arch= new File ("miArchivo");  
            FileReader in = new FileReader(arch);  
            int c;  
  
            while ((c = in.read()) != -1)  
                procesar c;  
  
            in.close();  
        } catch (IOException e) { hacer algo;}  
    }  
}
```

Excepciones: Ventajas

- Separa el código de manejo de errores del código normal → hace más legibles los programas.
- Propaga errores a través de los métodos que se encuentran activos en el *stack* → transfiere el control en el anidamiento de invocaciones al lugar adecuado para su manejo.
- Permite agrupar errores y diferenciar errores → una excepción es una clase que se puede derivar.

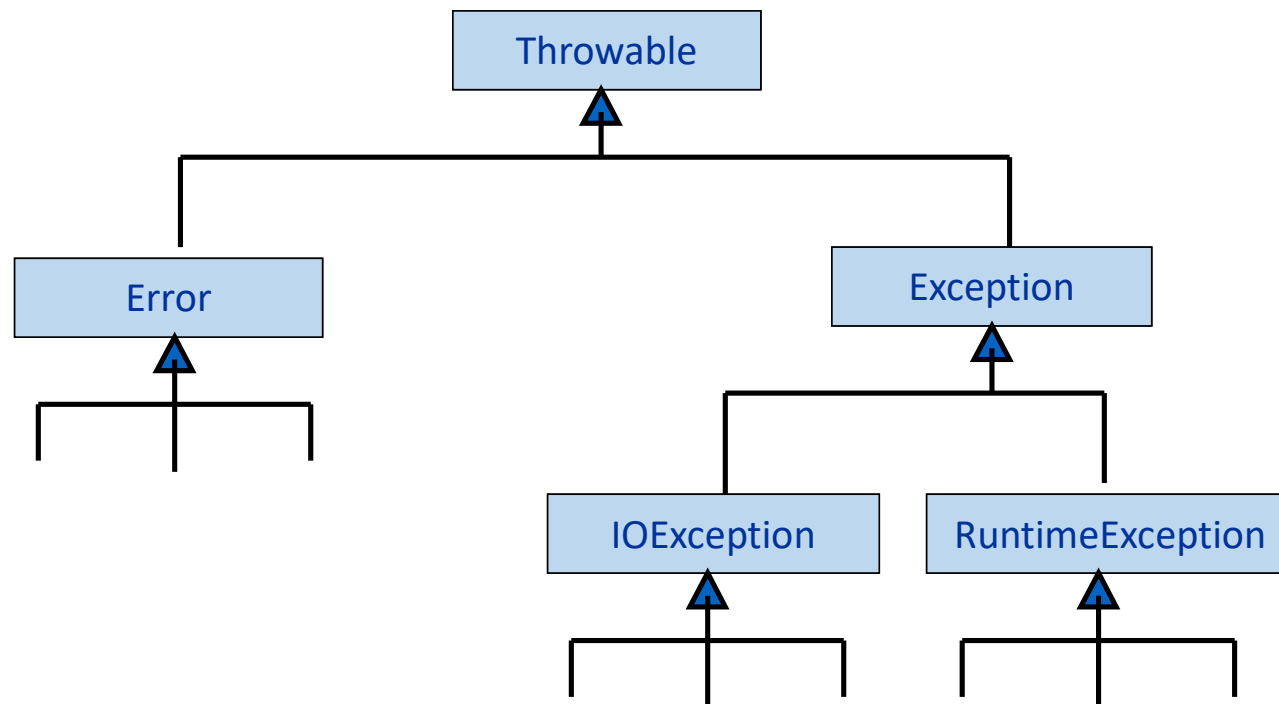
Excepciones: Anidamiento



Excepciones: Jerarquía de Clases

- En Java toda excepción se deriva de la clase `Throwable`.
- Existen dos subclases:
 - **Error**: representa un error interno o agotamiento de recursos en el sistema del runtime de Java
 - **Exception**: representa un error en el programa. Tiene dos subclases:
 - `IOException` y
 - `RuntimeException`

Excepciones: Jerarquía de Clases



Si sucede una **RuntimeException** es porque el programador ha hecho algo mal, no así **IOException**.

Excepciones: Subclases

- **RuntimeException** se debe a un error de programación, tales como:
 - mal uso de *cast*
 - Acceso a arreglo fuera de límite
 - Acceso con referencia nula
- **IOException** son por algún otro problema, tales como:
 - Leer más allá del final del archivo
 - Abrir una URL mal formada

Excepciones: Tipos

- Excepciones no verificadas (*unchecked*)
 - Excepciones derivadas de la clase `Error` y `RuntimeException`
- Excepciones verificadas (*checked*)
 - Otras clases de excepciones (`IOException`)

Excepciones: Manejo

- Un método advierte al compilador sobre excepciones que no puede manejar, lanzándolas con `throws`.
- Ejemplo: `public String leerLinea() throws IOException`
- Un método debe declarar todas las excepciones verificadas.
 - Si no declara todas, el compilador reclama
 - Al declarar una clase de excepciones, entonces puede lanzar cualquier excepción de alguna subclase
 - Aquellas excepciones que son capturadas (*catch*) no salen del método y no debieran ser declaradas

Excepciones: Manejo

- Los pasos necesarios para lanzar una excepción son:
 - Se debe elegir una clase apropiada de excepción
 - Crear un objeto de excepción de esa clase
 - Lanzar la excepción con **throw**

```
if (ch == -1) {           // se encuentra EOF
    if (leídos < tamaño)
        throw new EOFException();
}
```

Excepciones: Manejo (ejemplo)

```
public Object pop() throws EmptyStackException {  
    Object obj;  
  
    if (size == 0)  
        throw new EmptyStackException();  
  
    obj = objectAt(size - 1);  
    setObjectAt(size - 1, null);  
    size--;  
    return obj;  
}
```

Creación de Nuevas Clases de Excepciones

- Si las excepciones estándares no son adecuadas, el usuario puede definir las propias, derivando alguna clase existente.
- Es práctica común definir un constructor de omisión y otro con un mensaje detallado.
- Ejemplo:

```
public class FileFormatException extends IOException {  
    public FileFormatException () {}  
  
    public FileFormatException (String msg) {  
        super(msg);  
    }  
}
```

Creación de Nuevas Clases de Excepciones

```
public class FileFormatException extends IOException {  
    public FileFormatException () {}  
  
    public FileFormatException (String msg) {  
        super(msg);  
    }  
}
```

```
public String leer(BufferLectura lector) throws FileFormatException {  
    ...  
    while ( ... ) {  
        if (ch == -1) { // se encuentra EOF  
            if (leidos < tamaño)  
                throw new FileFormatException();  
        }  
        ...  
        return str;  
    }  
}
```


Capturando Excepciones

- Sentencia **try** permite definir un bloque de sentencias para las cuales se quiere definir un manejador de excepciones
- Para cada clase de excepción se define un manejador diferente con **catch**
 - Usando una superclase común, varias subclases diferentes de excepciones pueden tener un único manejador
- Formato:

```
try {  
    // grupo de sentencias  
} catch (ClaseExcepción1 e) { ...}  
  
} catch (ClaseExcepciónn e) { ...}
```

Capturando Excepciones: Ejemplo

```
try {  
    ...  
} catch (Exception e) {  
    System.err.println("Capturó ArrayIndexOutOfBoundsException: " + e.getMessage());  
} catch (IOException e) {  
    System.err.println("Capturó IOException: " + e.getMessage());  
}
```

Relanzando una Excepción

```
Graphics g = imagen.getGraphics();  
try {  
    // código que lanza excepciones  
} catch (MalformedURLException e) {  
    g.dispose();  
    throw e;  
}
```

Relanza la excepción
capturada por el manejador
de `MalformedURLException`

¡Se podría lanzar también una excepción diferente!

Cláusula finally

- Cuando se lanza una excepción, se detiene el procesamiento normal del método.
- Dicha situación puede ser un problema si el método ha adquirido recursos, por lo que se hace necesario disponer de un código de limpieza.
- La solución es usar la cláusula `finally`, que se ejecuta haya o no ocurrido una excepción antes de retornar.
- En caso de capturar una excepción, el código de manejo se ejecuta antes que el de `finally`.

Cláusula finally: Ejemplo

```
public void writeList() {  
    PrintWriter out = null;  
  
    try {  
        System.out.println("Entering try statement");  
        out = new PrintWriter(new FileWriter("OutFile.txt"));  
        for (int i = 0; i < size; i++)  
            out.println("Value at: " + i + " = " + vector.elementAt(i));  
    } catch (ArrayIndexOutOfBoundsException e) {  
        System.err.println("Caught ArrayIndexOutOfBoundsException: " + e.getMessage());  
    } catch (IOException e) {  
        System.err.println("Caught IOException: " + e.getMessage());  
    } finally {  
        if (out != null) {  
            System.out.println("Closing PrintWriter");  
            out.close();  
        } else {  
            System.out.println("PrintWriter not open");  
        }  
    }  
}
```

Recordar

- El mecanismo de excepciones permite transferir el control cuando suceden errores en el programa.
- El código de manejo de los errores se puede separar del código de desarrollo normal.
- Cada método debe declarar las excepciones verificadas que no puede manejar, de manera que los programadores sepan al usar la clase que excepciones pueden esperar.

5.7 Paquetes

Concepto de Paquete

- Contiene clases, interfaces y subpaquetes que están relacionados.
- Razones para definirlos:
 - Permiten agrupar interfaces y clases relacionadas.
 - Interfaces y clases pueden usar nombres públicos populares sin tener conflictos con otros paquetes.
 - Paquetes pueden tener componentes que sólo están disponibles dentro del paquete.
 - Facilita la distribución de software.

Paquete: Mecanismo de Definición

- En la primera línea de cada archivo fuente de una clase o interfaz debe aparecer: `package <nombre>;`
- El nombre del paquete implícitamente se antepone al nombre de la clase o interfaz. Ejemplo: `miPaquete.miClase`
- Código externo al paquete puede referenciar a tipos internos del paquete de acuerdo a las reglas de control de acceso

Paquete: Ejemplo de Uso

```
package Personas;  
  
class Persona {  
    private static int    nextID = 0;  
    private      int      ID;  
    private      String   Nombre;  
    private      int      edad = -1;  
  
    //... Otros métodos  
}
```

Se puede referenciar la clase `Persona` por: `Personas.Persona`

Referencias Externas

- Una forma es anteponer el nombre del paquete a cada tipo (si existen subpaquetes se puede definir una jerarquía).
 - Esta forma es razonable si se usan pocas referencias a tipos del paquete.
- La otra forma es mediante `import`, lo que permite usar referencias en el código sin anteponer nombre del paquete.
 - Ejemplo:
`import Personas.*`
`// ahora se puede usar la clase Personas sin anteponer nombre del paquete`

Colisiones de Nombres

- Si se usa más de un paquete, se pueden producir colisiones de nombres.
- Para controlar el uso correcto se puede usar:
 1. Anteponer el nombre del paquete en aquellos casos de colisión (ej.: `Personas.Persona`)
 2. Importar en forma completa un solo paquete y usar nombres completos para los nombres que colisionan.
- Ejemplo:
 - `import Personas.*`
 - `// OtrasPersonas.Persona se refiere al otro paquete`
 - `// Persona se refiere ahora a Personas.Persona`

Colisiones de Nombres

- Si dos paquetes tienen el mismo nombre y requieren ser usados, existe un problema.
- Una solución es usar nombres anidados:
 - Ejemplo: `MiProyecto.MiPaquete.*`
- En grandes organizaciones se acostumbra a usar nombres de dominio de Internet en orden inverso:
 - Ejemplo: `cl.utfsm.inf.MiPaquete.*`

Control de Acceso

- Clases e Interfaces tienen dos accesos:
 - **Público**: se permite su acceso fuera del paquete.
 - **Privado al paquete**: sino se declara **public** (sin calificador), el acceso se restringe dentro del paquete.
- Miembros de clase declarados:
 - Sin modificador de acceso son accesibles dentro del paquete, pero no fuera de él.
 - Miembros no privados (paquete, protegido y público) son accesibles por cualquier código del paquete (son amistosos).

Recomendaciones

- Los paquetes se deben definir para clases e interfaces relacionadas:
 - Tal agrupación permite una mejor reutilización.
 - Permite usar nombres adecuados sin provocar colisiones.
- Los paquetes se pueden anidar.
 - Permite agrupar paquetes relacionados (ej.: `java.lang`).
 - Es un asunto organizacional, no de control de acceso.
 - Jerarquía típicamente se refleja en la estructura del directorio.

Ejemplos en Java

- java.lang
- java.io
- java.util
- java.math
- java.awt
- java.net
- java.rmi
- java.applet
- java.sql
- java.beans
- java.security
- javax.swing
- javax.servlet
- javax.crypto
- javax.accessibility
- javax.naming
- javax.transaction
- javax.xml
- javax.sound
- javax.print

Unidad 5

Tipos de Datos Abstractos y Programación Orientada al Objeto

FIN

5.1 Tipos de Datos Abstractos

5.2 Orientación a Objetos (visión general)

5.3 Clases y Objetos

5.4 Herencia

5.5 Interfaces

5.6 Manejo de Excepciones

5.7 Paquetes