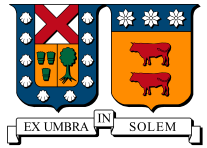


Tipos de Datos Abstractos y Orientación a Objetos

Roberto Díaz
roberto.diazu@usm.cl

Universidad Técnica Federico Santa María
Departamento de Informática – Santiago, Chile

2021-2

Introducción

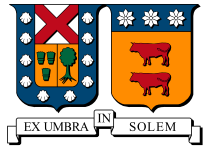
Mayoría de lenguajes modernos proveen soporte para realizar abstracciones de datos y de procesos.

Idea 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.

> **Ejemplo**: Una estructura como un registro de datos definida por el usuario no puede expresar mediante el lenguaje la asociación de operaciones especializadas para manipular esta estructura.

La Orientación a Objetos (OO) está fuertemente basada en la idea de TDA, siendo ambos conceptos tratados en este capítulo.



Tipos de Datos Abstractos

Tipo de Dato Abstracto

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.
  Por lo tanto, las únicas operaciones posibles son las provistas por la definición del tipo (interfaz).
- **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.
  Otras unidades de programa les está permitido crear variables del tipo definido, pero les está restringido el acceso a los detalles de la implementación.

Tipo de Dato Abstracto
Ventajas

- **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.
- **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

El *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 vacio
- `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 Stack 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 Stack en C++ (Interfaz)

```
// Stack.hpp - Archivo de Header de clase Stack
#include <iostream>
class Stack {
  private: /* no visibles para otros */
    int *stackPtr;
    int maxLen;
    int topPtr;
  public: /* visible para clientes */
    Stack(); /* Un constructor */
    ~Stack(); /* Un destructor */
    void push(int);
    void pop();
    int top();
    int empty();
}
```

Tipo de Dato Abstracto
Ejemplo Stack en C++ (Implementación)

```
// Stack.cpp - archivo de implementacion de clase Stack
#include <iostream>
#include "Stack.h"
Stack::Stack() { /* Un constructor */
  stackPtr = new int [100];
  maxLen = 99;
  topPtr = -1;
}
Stack::~Stack() {delete [] stackPtr;}; /* Un destructor */
void Stack::push(int number) {
  if (topPtr == maxLen)
  cerr << "Error en push-stack lleno\n";
  else stackPtr[++topPtr] = number;
}
...
```

Tipo de Dato Abstracto
Ejemplo 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 o `package`.

Tipo de Dato Abstracto
Ejemplo Java

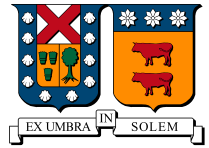
```
class StackClass {
  private int [] stackRef;
  private int maxLen, topIndex;
  public StackClass() { // un constructor
    stackRef = new int [100];
    maxLen = 99;
    topIndex = -1;
  };
  public void push (int num) {...};
  public void pop () {...};
  public int top () {...};
  public boolean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

Tipo de Dato Abstracto
Ejemplo Java (Interfaz)

```
interface StackType {
  void push (int num);
  void pop ();
  int top ();
  boolean empty ();
}
```

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

Tipo de Dato Abstracto
Ejemplo Python



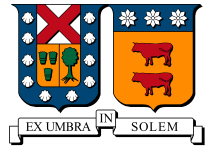
Python provee soporte para definir clases y objetos. Sin embargo, la definición es bien relajada.

- 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 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



Los Tipos de Datos Abstractos Parametrizados permiten diseñar un TDA que puede almacenar cualquier tipo de elementos, siendo este constructo de tipos sólo relevante para lenguajes con ligado de tipo estático.

- En orientación a objetos se les conoce también como ”clases genéricas”.
- Soportada por lenguajes como C++, Ada, Java 5.0 y C#.

Tipo de Dato Abstracto Parametrizado
Ejemplo C++

```
template <class Type> // Type es parametro de tipo
class Stack {
  private:
    Type *stackPtr;
    const int maxLen;
    int topPtr;
  public:
    Stack(int size) // Constructor
    {
      stackPtr = new Type[size];
      maxLen = size - 1;
      topSub = -1;
    }
    ...
}
...
Stack<int> myIntStack(5); //Instanciacion
```

Tipo de Dato Abstracto Parametrizado
Java 5.0

Los usuarios pueden definir clases genéricas, donde los parámetros genéricos deben ser clases.

- No pueden almacenar datos primitivos
- Evita tener diferente tipos en la estructura y usar casting para objetos.

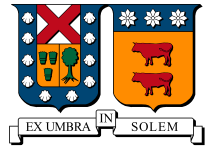
Los tipos genéricos más conocidos son de tipo colección como `LinkedList` y `ArrayList` (paquete `java.util.Collection`).

Ejemplo

```
ArrayList <Integer> miArreglo = new ArrayList <Integer> ();
miArreglo.add(0, 47); // agregar 47 en posicion 0
```

Tipo de Dato Abstracto Parametrizado
Ejemplo Java 5.0

```
import java.util.*;
public class CustomStack<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);
    ...
}
...
CustomStack<String> myStack = new Stack2<String>();
```



Orientación a Objetos

Introducción

La Programación Orientada a Objetos (POO) posee tres características fundamentales:

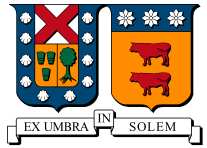
- Tipo de Datos Abstracto (TDA)
- Herencia
- Polimorfismo

Introducción

Existen muchos lenguajes para POO:

- Algunos lenguajes son OO puros (p.e. Smalltalk y Ruby)
- Soportan simultáneamente programación procedural y POO (e.g. ADA y C++); son básicamente extensiones.
- Algunos soportan programación funcional (e.g. CLOS)
- Otros lenguajes no soportan otros paradigmas, pero usan su estructura imperativa (e.g. Java y C#)
- Algunos lenguajes funcionales soportan Programación OO.

Características OO

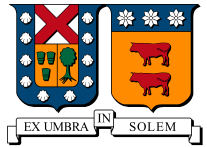


- **Herencia (*Inheritance*)**: permite extensión de datos u operaciones. Si se quiere reutilizar una componente de software a veces es necesario agregarle datos y nuevas operaciones.
- **Redefinición (*Overriding*)**: a veces se requiere redefinir el comportamiento específico de una operación (e.g.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

Características OO

- **Abstracción**: se requiere abstraer operaciones similares para diferentes componentes en un nuevo tipo (e.g. 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 (e.g. sobrecarga y tipos parametrizados).

Modelo Objetual (Básico)

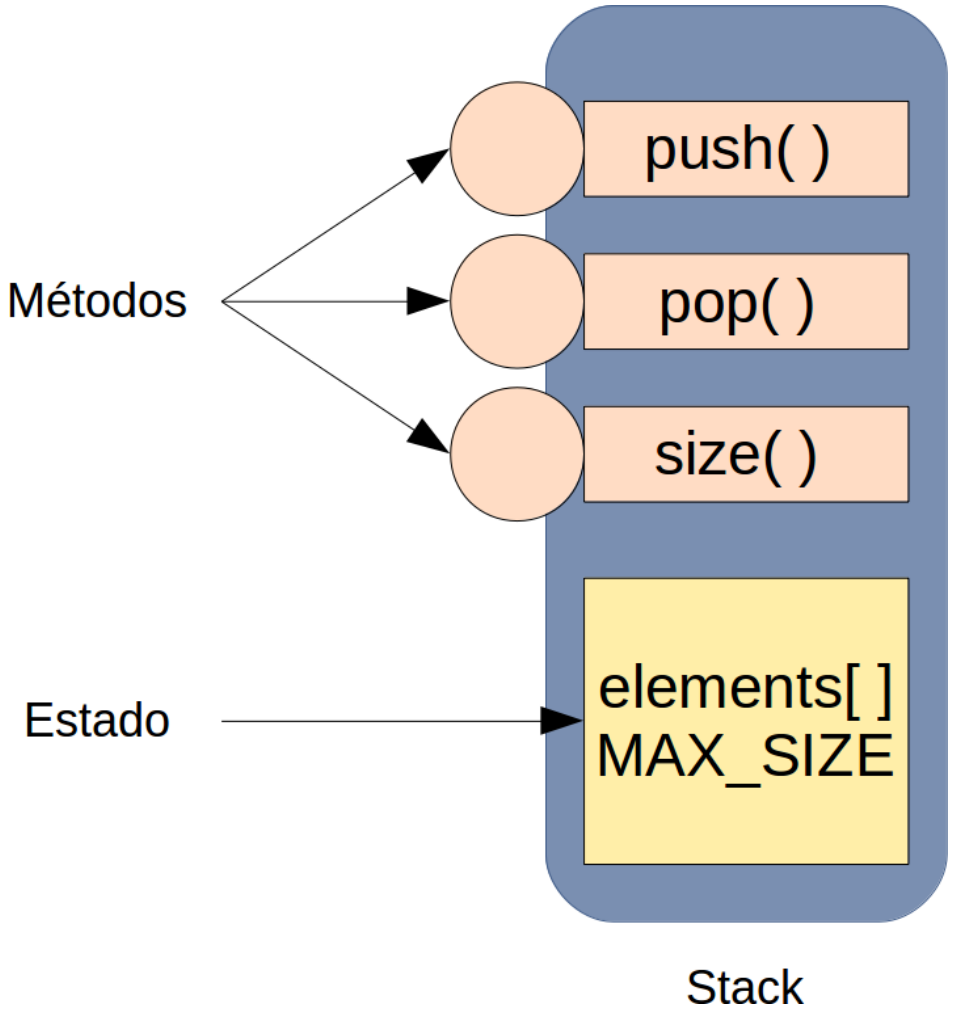


- **Programa**: conjunto de objetos que interactúan entre si.
- **Clase**: corresponde a una declaración de tipo, que especifica el estado y comportamiento que tendrán sus instancias u objetos.
  - El tipo queda definido por su interfaz, que especifica métodos y constantes.
  - La implementación del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

Modelo Objetual (Básico)

- **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.

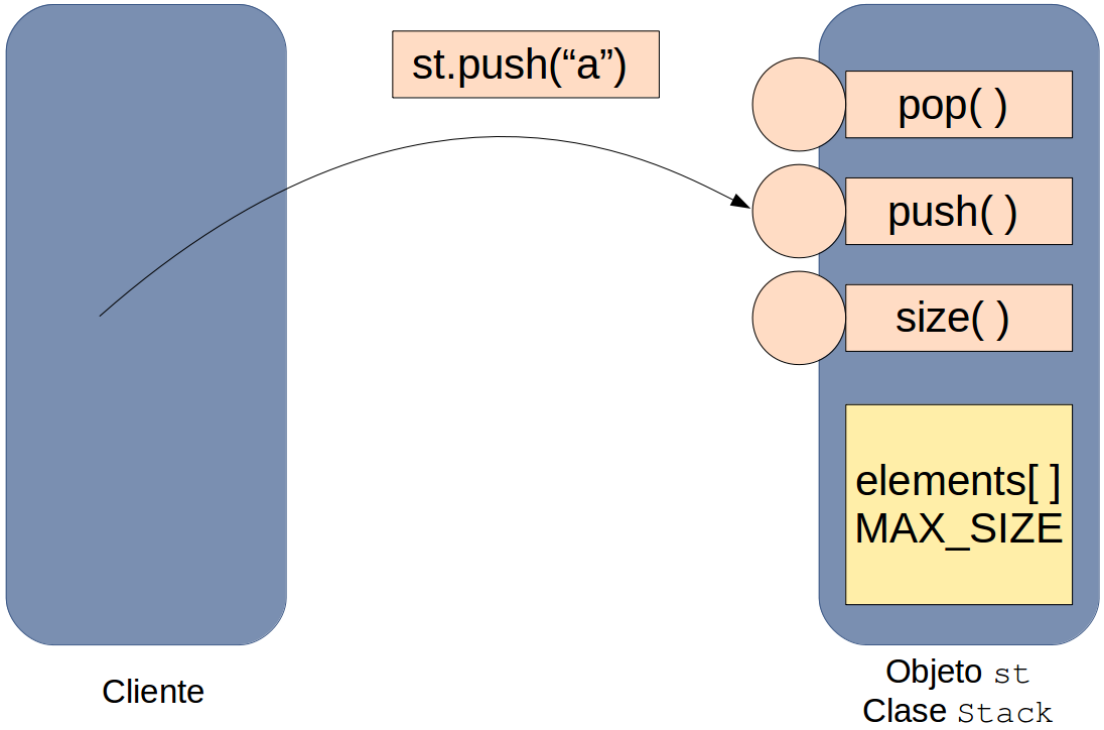
Objeto
Ejemplo



Clase
Ejemplo Java

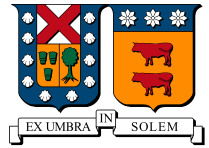
```
public class Stack {
  private int maximo;
  private int top = -1;
  private String[] buffer;
  public Stack (int i) {
    maximo = i; buffer = new String[i];
  }
  public int size() {
    return top+1;
  }
  public void push(String nuevo) {
    if (top < maximo-1) buffer[++top] = nuevo;
    else 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;
  }
}
```

Mensaje
Ejemplo



Mensaje
Ejemplo

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



Clases y Objetos

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 un 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

```java
/* Clase con nombre "Persona" con acceso "public" */
public class Persona {
  /* Miembro con nombre "ID" con acceso "private" */
  private int ID;
  private String Nombre;

  /* Constructor */
  public Persona (String s, int id) {
    Nombre = s; ID = id;
  }

  /* Protocolo de metodo "miNombre" */
  public String miNombre(int id) {
    /* Implementacion metodo "miNombre" */
    if (id == ID)
      return Nombre;
    else return null;
  }
}
```

Variables miembros

Las variables de clases se denominan campos. 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 similares (de la misma clase).

Variables miembros se pueden compartir entre todos los objetos de una clase con el modificador `static`.

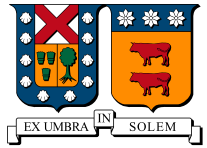
- Todos los objetos de una clase comparten una única copia de un campo declarado como static

En general, cuando se habla de variables y métodos miembros se refiere a aquellos no estáticos.

Variable miembro estática
Ejemplo

```
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;
  }
}
```

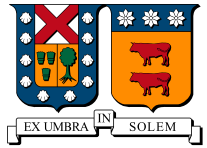
Control de Acceso de Miembros



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



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 (*garbage collector*); y si aun no existe suficiente memoria, se lanza una excepción del tipo `OutOfMemoryError`.

Terminada la inicialización, el *runtime* retorna la referencia al nuevo objeto.

Creación de objetos
Ejemplo

```
//Declaracion, p1 es igual a null
Persona p1;
//Instanciacion
Persona p2 = new Persona("Pedro");
//Instanciacion de nuevo objeto
p2 = new Persona("Jose");
```

Constructores

Un objeto recién creado debe inicializar las variables miembro.
Las variables miembro pueden ser inicializadas en la declaración de forma explícita.

- Muchas veces se requiere algo más.
- **Ejemplo**: ejecutar código para abrir un archivo.

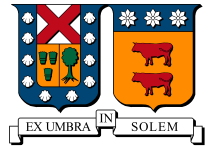
Los constructores cumplen ese propósito; tienen el mismo nombre de la clase y pueden recibir parámetros. Los constructores no son un método (no retornan un valor).
Una clase puede tener varios constructores.

Constructor
Ejemplo

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

Constructor
Razones

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



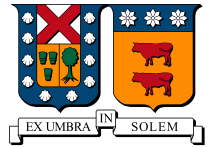
`private`
Ninguna otra clase puede instanciar la clase (e.g. Método fábrica).

`protected`
Sólo la clase, subclases de las 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.

Constructor por Omisión (*Default*)

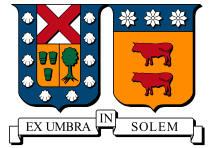


- Si no se define constructor, se asume un constructor sin argumentos, que no hace nada.
- Este constructor por omisión se asume que siempre existe, en el caso de no haberse definido un constructor
- El constructor por omisión es público si la clase lo es, sino no lo es.

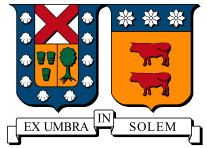
Constructores
Ejemplo múltiples constructores

```
class Persona {
  private static int nextID = 0;
  private int ID;
  private String Nombre;
  private int edad = -1;

  public Persona (String s) {
    ID = ++nextID;
    Nombre = s;
  }

  public Persona (String s, int ed) {
    this(s); //invoca al otro constructor
    edad = ed;
  }
  //...
}
```

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 esta sintaxis: `referencia.metodo(args)`
- Java no permite un número variable de argumentos como C o C++ (no permite elipsis)



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.

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 accesores para proteger el estado de los objetos.

Métodos
Ejemplo Accesor

```
class Persona {
  private static int nextID = 0;
  private int ID;
  private String Nombre;
  private int edad = -1;

  public Persona (String s) {
    ID = ++nextID;
    Nombre = s;
  }

  /* Accesor */
  public int getID(){
    return ID;
  }

  //...
}
```

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.

En el ejemplo anterior

```
public int getID(){
  return ID; //equivalente a: return this.ID;
}
```

Referencia `this`
Usos

Se usa usualmente para pasar una referencia del objeto actual como parámetro a otro método

```
Timer.wakeup(this,30);
```

Otro uso es cuando existe ocultamiento de un identificador debido a colisión de nombres

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

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

Cada método tiene un **protocolo** o **firma** (*signature*):

- Nombre del método
- Número y tipos de parámetros
- El tipo de retorno

Java permite tener varios métodos con un mismo nombre, pero con diferentes parámetros. Esto se denomina **sobrecarga** (*overload*).

Cuando se invoca un método sobrecargado, se calza por número y tipo los parámetros usados (reales), para seleccionar el adecuado.

Miembros y Métodos 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. Se deben inicializar antes de que se use cualquier variable estática o método de la clase.

Método estático: se invoca en nombre de toda la clase

- Sólo puede acceder a miembros estáticos (e.g. no permite usar `this`).
- Se invocan típicamente usando nombre de la clase.
- Se denominan también métodos de clase.

Método Estático
Ejemplo

```
public class PruebaUsoStack {
  /* Metodo estatico */
  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();
      //Clase "System" miembro estatico "out(...)"
      System.out.println(s);
    }
  }
}
```

Método Estático
Ejemplo

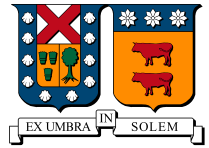
```
public class Matematica {
  /* Metodo estatico */
  public static float abs(float a) {
      return a < 0 ? -a : a;
  }
}

public class Prueba {
  public static void main(String[] args) {
    float a=1,b=-3,c=0;
    System.out.println(Matematica.abs(a));
    System.out.println(Matematica.abs(b));
    System.out.println(Matematica.abs(c));
  }
}
```

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.

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

Clases Anidadas



Java permite definir una clase como miembro de otra clase.

Útil cuando una clase sólo tiene sentido en el contexto específico de una clase.

Clase anidada tiene acceso total a todos los miembros de la clase a la que pertenece (como lo haría cualquier otro método miembro de la clase superior).

Clases Anidadas

Si clase anidada se declara estática, se llama **clase anidada estática**

- No tiene acceso a miembros no estáticos.
- 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 en la clase a que pertenece.
- Permite relacionar objetos de las clases relacionadas.

Clase Interna
Ejemplo

```
public class Stack {
  private Vector items;
  //codigo de metodos 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--);
    }
  }
}
```

Recolección de Basura

Java realiza recolección automática de basura. 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 `main`

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

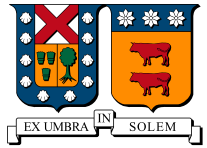
Método `main`
Ejemplo

Archivo fuente Echo.java

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

Ejecución programa

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



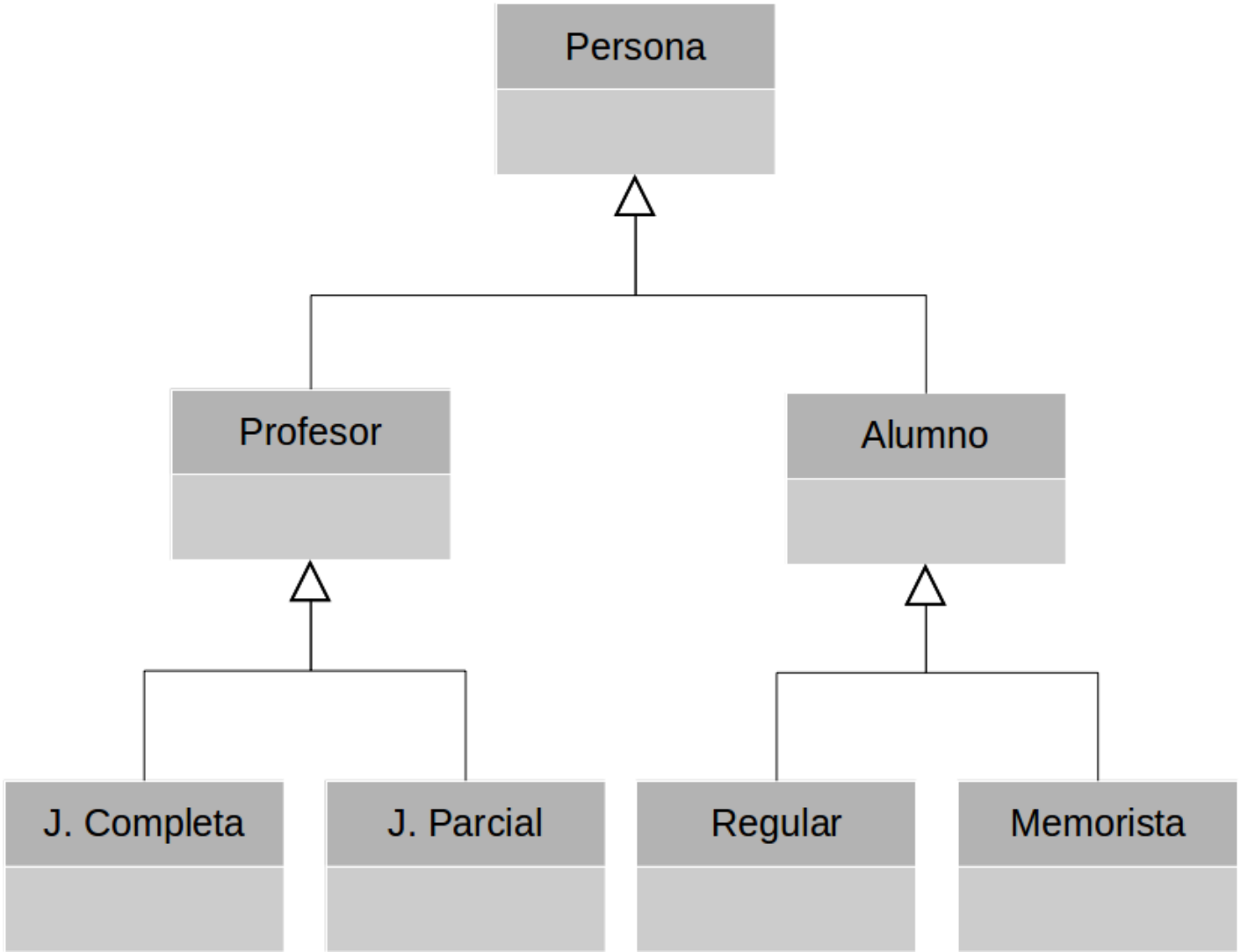
Herencia

Herencia

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**.
- Existe herencia simple y múltiple.

Herencia
Ejemplo



Conceptos preliminares

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** (o **subclase**) agrega funcionalidad de la otra clase (**superclase**), 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

- El mecanismo de herencia permite extender una clase (agregar variables o métodos).
- Si B extiende a A, entonces B hereda todo lo de A (la superclase), siendo la clase heredera B una subclase de A
- Una subclase heredera puede implementar a su manera un método heredado, redefiniéndolo (override).

Clase Extendida
Consideraciones

Una clase que se extiende sigue siendo del tipo de la superclase (`isA`), no viceversa
Ejemplo: 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`)
Ejemplo: Un círculo tiene un punto, pero no es un punto.

En casos que la extensión permite tener diferentes roles es más conveniente usar agregación
Ejemplo: Una persona que tiene roles diferentes no es simple extenderla por derivación

Clase Extendida

En Java todas las clases son extendidas, aún cuando no se hayan declarado específicamente de esa forma.

- Toda clase se deriva (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.

Java sólo permite tener una única superclase (**herencia simple**).

Herencia
Ejemplo

```
/*  Definicion */
public class StackPrint extends Stack {
  public StackPrint(int i) {
    super(i);
  }
  public void print() {
    System.out.println("size: " + (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();
  }
}
```

Reglas de Herencia

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 sobreescrito (*overriden*)

Reglas de 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`

Reglas de Herencia
Ejemplo Redefinición de Variables Miembros

```
class SuperClase {
  public Integer numero;
  public void print() {
    System.out.println(numero);
  }
}

class SubClase extends SuperClase {
  public Float numero;
  public void printBoth(){
    System.out.println(numero+" "+super.numero);
  }
}
```

```
SuperClase sup = new SuperClase();
SubClase sub = new SubClase();
sup.numero = 1; sub.numero = 2.0f;
sup.print(); //1
sub.print(); //null
sub.printBoth(); //2.0 null
```

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`.

Un método redefinido puede usar modificadores de acceso, pero sólo puede dar más acceso, no menos.
Ejemplo: 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:

- Métodos que han sido declarados abstractos en la superclase.

Reglas de Herencia
Redefinición de Métodos

```
class SuperClase {
  public Integer numeroInt;
  public void print() {
    System.out.println(numero);
  }
}

class SubClase extends SuperClase {
  public Float numeroFloat;
  public void print(){
    System.out.println(numeroFloat);
  }
}
```

```
SuperClase sup = new SuperClase();
SubClase sub = new SubClase();
sup.numeroInt = 1; sub.numeroFloat = 2.0f;
sup.print(); //1
sub.print(); //2.0
```

Reglas de Herencia
Constructores

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(...)`. Si la superclase no tiene un constructor sin argumentos se debe específicar un constructor.

Reglas de Herencia
Ejemplo Constructores

```
class SuperClase {
  protected int x = 0, y = 1;
  public SuperClase(){
    System.out.format("Sup() x:%d y:%d\n",x,y); x = y;
    System.out.format("Sup() x:%d y:%d\n",x,y);
  }
}
class SubClase extends SuperClase {
  protected int z = 2;
  public SubClase(){
    System.out.format("Sub() x:%d y:%d z:%d\n",x,y,z);
    y = z;
    System.out.format("Sub() x:%d y:%d z:%d\n",x,y,z);
  }
}
```

```
SubClase sub = new SubClase(); /* Output:
Sup() x:0 y:1
Sup() x:1 y:1
Sub() x:1 y:1 z:2
Sub() x:1 y:2 z:2 */
```

Clases y Métodos Abstractos

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

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).

Clase Abstracta
Ejemplo

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

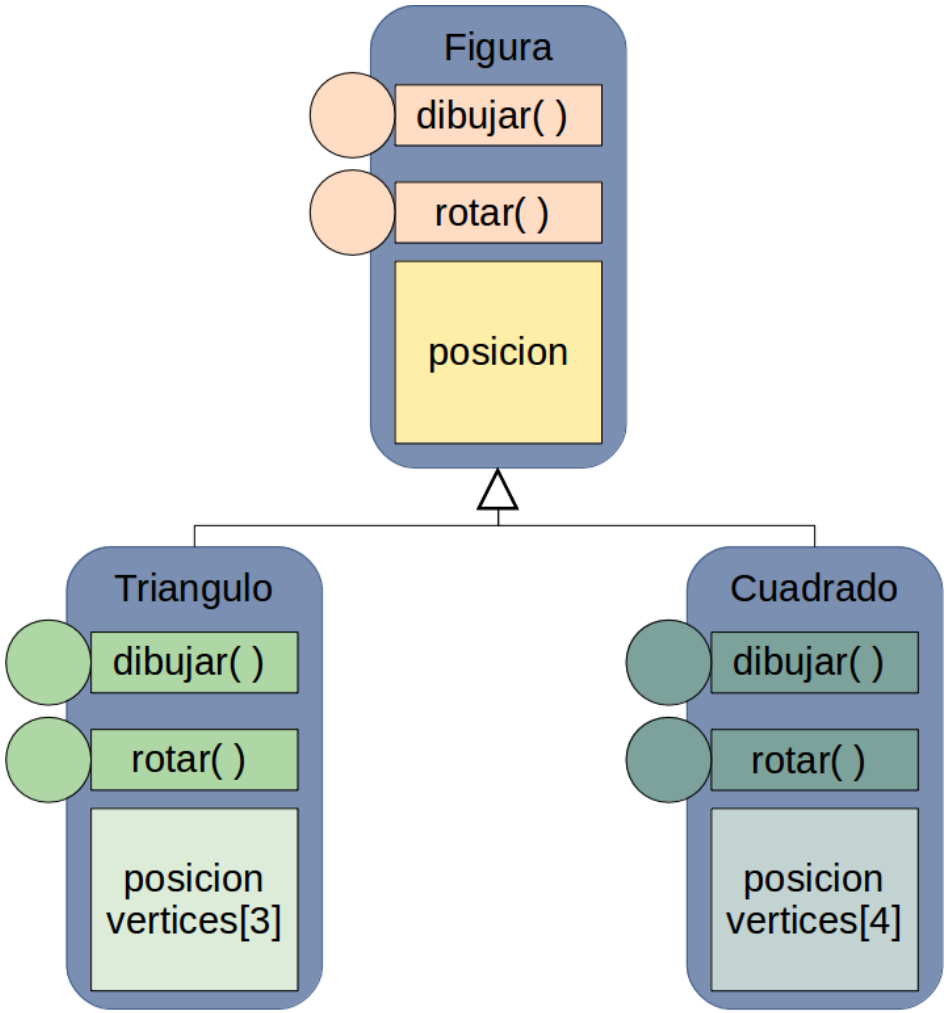
```
abstract class ObjectoGrafico {
  int x, y;
  //...
  void mover(int nuevoX, int nuevoY) {
    //...
  }
  abstract void dibujar();
}
```

```
class Circulo extends ObjectoGrafico {
  void dibujar() {
    //...
  }
}
class Rectangulo extends ObjectoGrafico {
  void dibujar() {
    //...
  }
}
```

Ligado Dinámico de Métodos

- Se 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 (***override***) el comportamiento del método de un antecesor.
- Cuando se invoca un método redefinido, dinámicamente se debe hacer el ligado al método que corresponda (***dynamic binding***).
- Permite a sistemas de software ser más fácilmente extendidos durante el desarrollo y mantenidos.

Ligado Dinámico de Métodos
Ejemplo



Asignación de Memoria

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 (e.g. *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.

Asignación de Memoria

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

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.

Además, 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
Ejemplo Java

```
//public la clase puede ser usada en cualquier ambito
public class Stack{

  //private no permite acceso a clientes ni subclases
  private int maximo;

  //protected permite acceso a subclases pero no a clientes
  protected int top = -1;
  protected String[] buffer;

  //permite acceso tanto a subclases como clientes
  public Stack(int i){
    maximo = i;
    buffer = new String[i];
  }
  ...
}
```

Modificador `final`

Se puede usar el modificador `final` en tres situaciones diferentes:

- **Variable**: la variable no puede ser modificada (constante).
- **Método**: el método no puede ser redefinido.
- **Clase**: la clase no se puede derivar (implícitamente todos sus métodos son finales).

Uso de `final` en métodos y clases restringe el reuso, por lo que debe usarse con reserva.

Herencia Múltiple

Se tiene **Herencia Múltiple** cuando una clase extiende más de una clase base.

Ventaja:

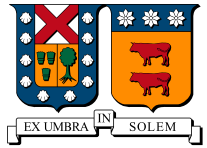
- A veces permite reutilización.

Problemas:

- **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**.

Esta disponible en C++ y Python. No está disponible en Java para clases, pero sí para interfaces

Clase `Object`



Todas las clases tienen a `Object` como raíz. En consecuencia heredan sus métodos.
Existen dos tipos de métodos:

- Métodos de utilidad general
- Métodos para soportar hebras (*threads*)

El último tipo no se estudia en el contexto de este curso.

Clase `Object`
Métodos de Utilidad General

```
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 necesariamente es igual a verificar que tienen la misma referencia (se puede hacer con ==).

```
public int hashCode()
```

- Retorna código hash del objeto, que es usualmente único para cada objeto.
- Sirve para ser usados en tablas de hash.

Clase Object
Métodos de Utilidad General

```
protected Object clone()
```

- Retorna un clon del objeto (una copia).

```
public final Class getClass()
```

- Retorna un objeto de tipo Class que representa la clase del objeto this.

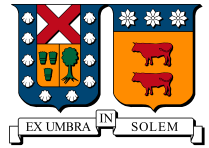
```
public void finalize()
```

- Finaliza un objeto durante la recolección de basura.

```
public String toString()
```

- Retorna una representación del objeto en un String
- Mayor parte de las clases lo redefinen.

Métodos `equals` y `hashCode`



Los métodos `equals` y `hashCode` debieran ser redefinidos conjuntamente.

Dos objetos que son iguales según `equals` debieran retornar lo mismo en la función `hashCode`.

Esto para garantizar el correcto funcionamiento en estructuras basadas en Hashing (`HashMap`, `Hashtable`, `HashSet`, etc)

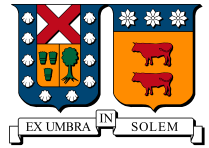
Método Equals
Ejemplo

```
String a = new String("eso mismo");
String b = new String("eso mismo");

System.out.println(a == b);         //false
System.out.println(a.equals(b));    //true
a = b;
System.out.println(a == b);         //true
System.out.println(a.equals(b));    //true
```

Método `toString()`

Si un objeto soporta el método `public String toString()`, el cual se invoca en cualquier expresión en que el objeto aparezca con los operadores + y +=.

```
class Persona {
  private int ID;
  private String Nombre;
  public String toString() {
    return ID + " : " + Nombre;
  }
  //... Otros metodos
}
//...
Persona p; //...
System.out.println("La persona" + p);
```



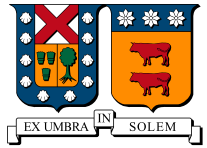
Interfaces

Interfaz

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.

Interfaces



Mecanismo de **Interfaz** es una manera de declarar tipos consistentes sólo de métodos abstractos y constantes.
Clases deben implementar los métodos de las interfaces.
Las interfaces son útiles para el diseño.

- Clases deciden cómo implementarlas.
- Una interfaz puede tener muchas implementaciones.

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.

Interfaz
Ejemplo 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];
  }
  // .....
}
```

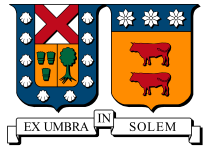
Interfaces
Declaración

La declaración de la interfaz debe incluir un nombre y un cuerpo de la interfaz que defina métodos y constantes relacionados.

La interfaz puede extender una o más interfaces. Las interfaces permiten herencia múltiple

`public` hace la interfaz visible fuera del paquete (opcional)

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



Todas las **constantes** son implícitamente **públicas**, **estáticas** y **finales**.

Todos los **métodos** son implícitamente **públicos** y **abstractos**.

No se aceptan otros modificadores (`private`, `protected` y `synchronized`)

Interfaces
Ejemplo

```
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];
  }
  void push(String s){
    //...
  }
  //...
}
```

Interfaces
Comparación con 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 Extendida
Ejemplo

La raíz de una jerarquía de interfaces no necesariamente es única:

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

No es el caso para las clases

```
class W {...}
class X extends W {...}
class Y extends W {...}
class Z extends Y, X {...} // Error
```

Conflictos de Nombre

¿Qué pasa si un mismo nombre se encuentra en más de una interfaz de los supertipos?

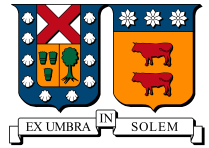
- Si las firmas no son idénticas, es trivial (se redefiniran diferentes métodos).
- Si tienen firma idéntica, entonces la clase tendrá un solo método con esa firma.
- Si las firmas sólo difieren en el tipo de retorno, no se puede implementar la interfaz.

Para constantes es simple: se selecciona cada una usando nombre de la interfaz
Ejemplo: X.var, Y.var

Modificación de Interfaces

Una interfaz, una vez declarada, no debiera ser modificada
Ejemplo: si se modifica una interfaz agregando un nuevo método, todas las clases que implementaban la especificación anterior ya no son válidas. La definición de una interfaz requiere diseño cuidadoso.

Una solución alternativa al problema anterior es definir una sub-interfaz con el nuevo método.



Paquetes y Encapsulación

Importación de clases y paquetes

La palabra `import` permite importar clases de un paquete.

- Se puede importar una clase:
  `import package.subpackage.class`
- Se pueden importar todas las clases de un paquete:
  `import package.subpackage.*`
- A pesar de la aparente jerarquía, importar un paquete no importa su ”subpaquete” (importar todo de `package` no importa las clases en `package.subpackage`)
- No es necesario importar una clase, pero en cada uso se deberá específicar el nombre completo de la clase si esta en un paquete distinto
- Las clases del paquete `java.lang` son importadas automáticamente

Paquetes
Concepto

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.

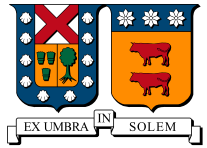
Paquetes
Mecanismo de definición

En la primera línea de cada archivo fuente de una clase o interfaz debe aparecer: `package nombrePaquete;`

El nombre del paquete implícitamente se antepone al nombre de la clase o interfaz.
Ejemplo: `unPaquete.unaClase;`
Código externo al paquete puede referenciar a tipos internos del paquete de acuerdo a las reglas de control de acceso.

Paquetes
Ejemplo de Uso

```
package personas;

class Persona {
  private static int nextID = 0;
  private int ID;
  private String nombre;
  private int edad = -1;

  //... Metodos
}
```



Para referenciar una clase que esta en un paquete distinto existen varias alternativas

Para referenciar una clase que esta en un paquete distinto existen varias alternativas

- Anteponer el nombre del paquete a cada tipo en cada uso. Razonable si hay pocas referencias.

```
personas.Persona objetoPersona = new personas.
    Persona();
```

Paquetes
Referencias Externas

Para referenciar una clase que esta en un paquete distinto existen varias alternativas

- Anteponer el nombre del paquete a cada tipo en cada uso. Razonable si hay pocas referencias.

```
personas.Persona objetoPersona = new personas.
    Persona();
```

- Importar desde el paquete la clase

```
import personas.Persona;
...
Persona objetoPersona = new Persona();
```

Paquetes
Referencias Externas

Para referenciar una clase que esta en un paquete distinto existen varias alternativas

- Anteponer el nombre del paquete a cada tipo en cada uso. Razonable si hay pocas referencias.

```
personas.Persona objetoPersona = new personas.
    Persona();
```

- Importar desde el paquete la clase

```
import personas.Persona;
...
Persona objetoPersona = new Persona();
```

- Importar todas las clases del paquete. Notar que así es más fácil generar colisión de nombres.

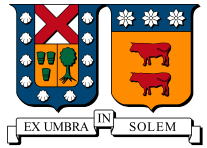
```
import personas.*;
...
Persona objetoPersona = new Persona();
```

Importación de clases y paquetes
Ejemplo

```
import Personas.Persona;

public class Main
{
  public static void main(String[] args)
  {
    Persona p = new Persona();
    // ...
  }
}
```

Importación de clases y paquetes
Ejemplo

```
import java.util.Calendar;

public class Main
{
  public static void main(String[] args)
  {
    Calendar calendar = Calendar.getInstance();
    java.util.Date firstDate = calendar.getTime();
    //String pertenece a java.lang
    String s = firstDate.toString();

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



Si dos paquetes tienen el mismo nombre y requieren ser usados, existe un problema.

Una solución es usar nombres anidados:
Ejemplo: `unProyecto.unPaquete.*`

En grandes organizaciones se acostumbra a usar nombres de dominio de Internet en orden inverso:
Ejemplo: `cl.utfsm.inf.unPaquete.*`

Clases e **interfaces** tienen dos accesos:

- **Público**. Si se califican con `public` está permitido su acceso fuera del paquete.
- **Privado al paquete**. Sino se declara `public` (sin calificador), 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 (`package`, `protected` y `public`) son accesibles por cualquier código del paquete (son amistosos).

Paquetes
Recomendaciones

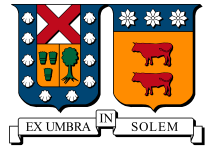
Los paquetes se deben definir para clases e interfaces relacionadas:

- Tal agrupación permite una mejor reutilización.
- Permite usar nombres adecuados sin provocar colisiones.

Los paquetes se pueden anidar:

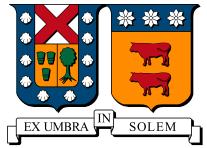
- Permite agrupar paquetes relacionados (e.g. `java.lang`).
- Es un asunto organizacional, no de control de acceso.
- Jerarquía típicamente se refleja en la estructura del directorio.

Paquetes de Java



- `java.lang`
- `java.io`
- `java.util`
- `java.math`
- `java.awt`
- `java.net`
- `java.rmi`
- `java.applet`
- `java.sql`
- `java.beans`
- `java.security`
- etc.

- `javax.swing`
- `javax.servlet`
- `javax.crypto`
- `javax.accessibility`
- `javax.naming`
- `javax.transaction`
- `javax.xml`
- `javax.sound`
- `javax.print`
- etc.



Manejo de Excepciones

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 (e.g. 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

Excepciones
Ejemplo

Tradicionalmente se debían usar códigos de error

```
TipoError leerArchivo(){
  TipoError errorCode = NO_ERROR;
  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 leidos;
  }
  if (n != EOF)
    return ERROR_LECTURA_ARCHIVO;
}
```

Excepciones
Ejemplo

Excepciones simplifican el manejo de errores

```
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) {
      //manejar excepcion
    }
  }
}
```

Excepciones
Ventajas

Separa el código de manejo de errores del código normal haciendo 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 y diferenciar errores. Una excepción es una clase que se puede derivar.

Excepciones
Ejemplo de Anidamiento

```
public void metodo1(){
  try{
    metodo2();
  }catch(Exception e){
    procesarExcepcion(e);
  }
}

private void metodo2() throws Exception {
  metodo3();
}

private void metodo3() throws Exception {
  leerArchivo();
}
```

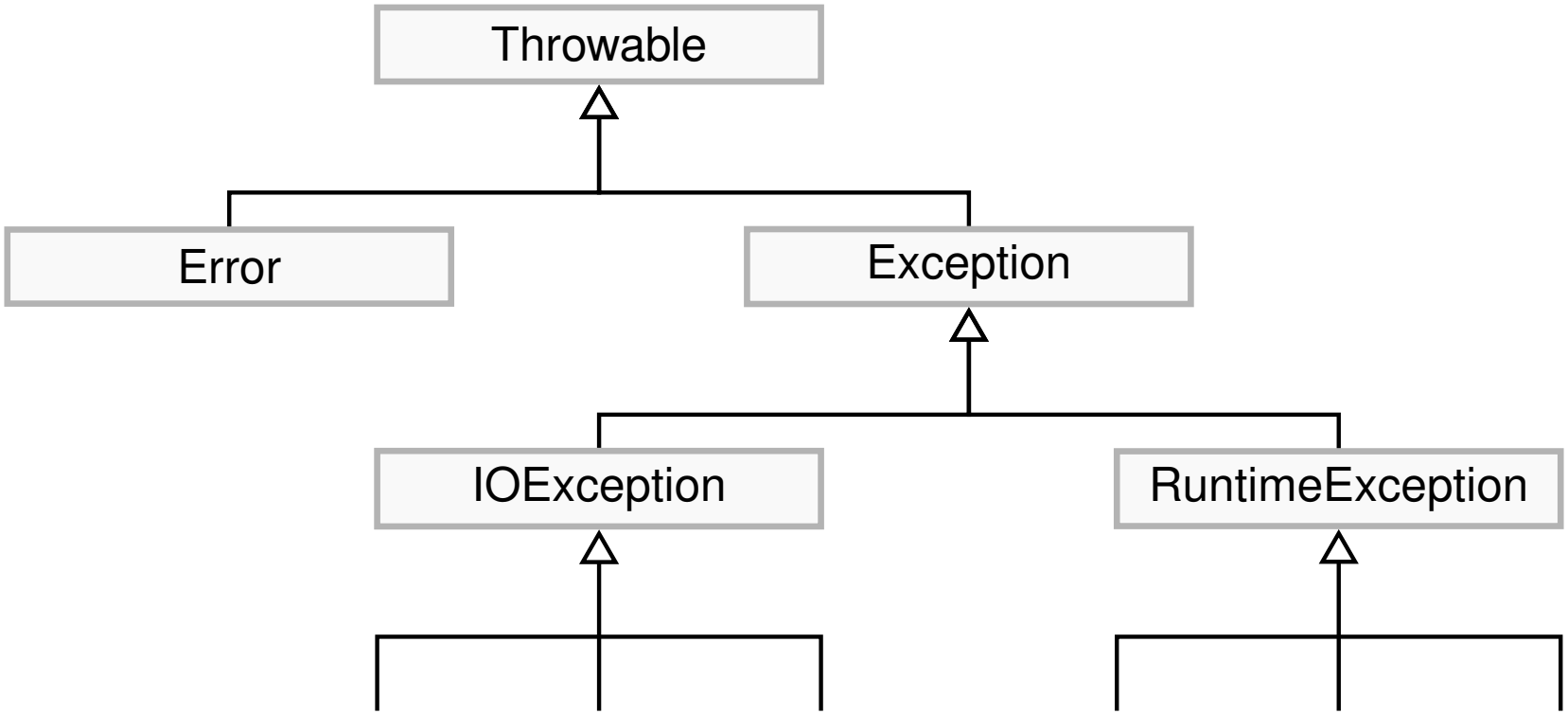
Excepciones
Jerarquía de clases

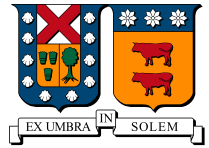
En Java toda excepción se deriva de la clase `Throwable`.

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 subclase:
  - `IOException`
  - `RuntimeException`

Excepciones
Jerarquía de Clases



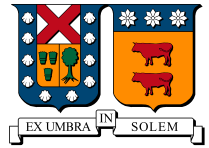


`RuntimeException` se debe a un error de programación:

- Mal uso de cast
- Acceso a arreglo fuera de límite
- Acceso con referencia nula

`IOException` son por algún otro problema:

- Leer más allá del final del archivo
- Abrir una URL mal formada
- etc.



Excepciones no verificadas (`unchecked`)
Excepciones derivadas de la clase `Error` y `RuntimeException`

Excepciones verificadas (`checked`)
La clase `Exception` y subclases derivadas de ésta (como `IOException`) distintas de `RuntimeException`.

Un método advierte al compilador sobre excepciones que no puede manejar con la palabra clave `throws`.
Ejemplo:

```
public String leerLinea() throws IOException
```

Un método debe declarar todas las excepciones verificadas (*checked*)

- 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

Para lanzar una excepción se debe usar la sentencia `throw` creando una nueva excepción de esa clase.

Ejemplo:

```
//...
if (ch == -1) { // se encuentra EOF
  if (leidos < tamano)
    throw new EOFException();
}
//...
```

Excepciones
Ejemplo Lanzamientos

```
public Object pop() throws EmptyStackException {
  Object obj;

  if (size == 0)
    throw new EmptyStackException();

  obj = objectAt(size - 1);
  setObjectAt(size - 1, null);
  size--;
  return obj;
}
```

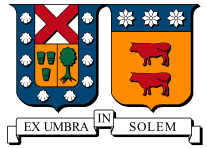
Si las excepciones estándares no son adecuadas, el usuario puede definir las propias, derivando alguna clase existente.
Es práctica común definir un constructor de omisión y otro con un mensaje detallado.

Ejemplo:

```
class FileFormatException extends IOException {
  public FileFormatException () {
  }

  public FileFormatException (String msg) {
    super(msg);
  }
}
```

Excepciones
Ejemplo de Uso

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



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

Excepciones
Ejemplo Captura de Excepciones

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

Dentro del bloque `catch` se puede relanzar la excepción. Se puede incluso lanzar una excepción diferente.

```
try {
// codigo que lanza excepciones
} catch (MalformedURLException e) {
  g.dispose();
  throw e;
}
```

Excepciones
Cláusula `finally`

Cuando se lanza una excepción, se detiene el procesamiento normal del método:

- Esta situación puede ser un problema si el método ha adquirido recursos
- 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`

Excepciones
Ejemplo Cláusula `finally`

```
try {
  if( size == 0 ) return;
  out = new PrintWriter(new FileWriter("output.txt"));
  for (int i = 0; i < size; i++)
    out.println(i + " = " + vector.elementAt(i));
} 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");
  }
}
```

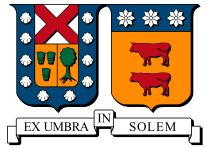
Resumen

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 (*checked*) que no puede manejar, de manera que los programadores sepan al usar la clase que excepciones pueden esperar.

Referencias



- Capítulos XII de [Sebesta, 2011]
- Capítulos V de [Louden, 2011]
- Capítulo XIII de [Tucker, 2006]