Programación III

Prof. Ing. Pablo Pandolfo

Contenido:

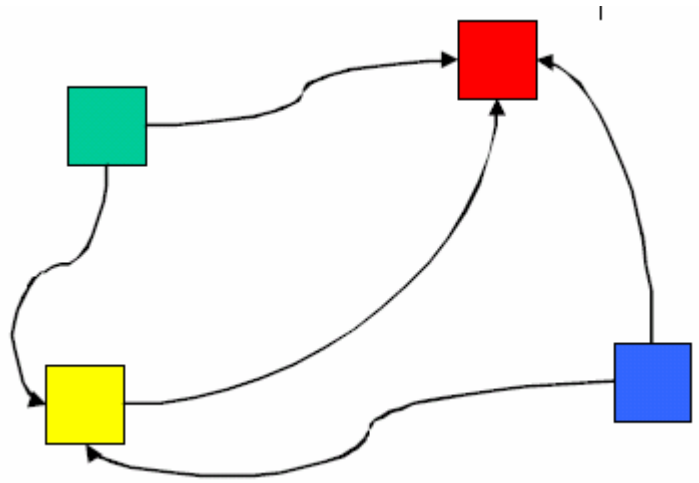
- Introducción a Programación con Objetos.
- Introducción al Lenguaje de Programación Java.
- Encapsulamiento, Herencia y Polimorfismo.
- Colecciones.
- Excepciones y Archivos.
- JDBC.
- Swing.
- Servlet.

Introducción a Programación con Objetos

- Paradigma:
  - Colección de modelos conceptuales (principios o fundamentos específicos) que juntos modelan el proceso de diseño, orientan la forma de pensar y solucionar los problemas y, por lo tanto, determinan la estructura final de un programa.
  - Clasificación
    - Operacionales: Indican el modo de construir la solución, es decir detallan paso a paso el mecanismo para obtenerla (secuencia e instrucciones de control)
    - Declarativos: Describen las características que debe tener la solución. (proposiciones, condiciones, ecuaciones, transformaciones)
- Programa:
  - Especificación formal de un algoritmo por medio de un lenguaje de programación.
- Algoritmo:
  - Secuencia de acciones elementales que transforma los datos de entrada en datos de salida con el objetivo de resolver un problema computacional.
- Lenguaje de programación:
  - Notación para escribir programas. Proporcionan estructuras (sintácticas: cómo escribir una expresión y semánticas: cómo evaluar una expresión) para organizar los cálculos.

Introducción a Programación con Objetos

- Paradigma Orientado a Objetos. Definición:
  - Dentro de los paradigmas operacionales.
  - Se fundamenta en concebir a un sistema como un conjunto de entidades que representan al mundo real, los **“objetos”**, que tienen **distribuida la funcionalidad e información** necesaria y que **cooperan entre sí** para el logro de un objetivo común.



Introducción a Programación con Objetos

- Paradigma Orientado a Objetos. Objetivos:
  - Desarrollar los sistemas con modelos **más cercanos a la realidad** que a las especificaciones computacionales.
  - Construir componentes de software que sean **reutilizables**.
  - Diseñar una implementación de manera que puedan ser **extendidos** y **codificados** con el mínimo impacto en el resto de su estructura.

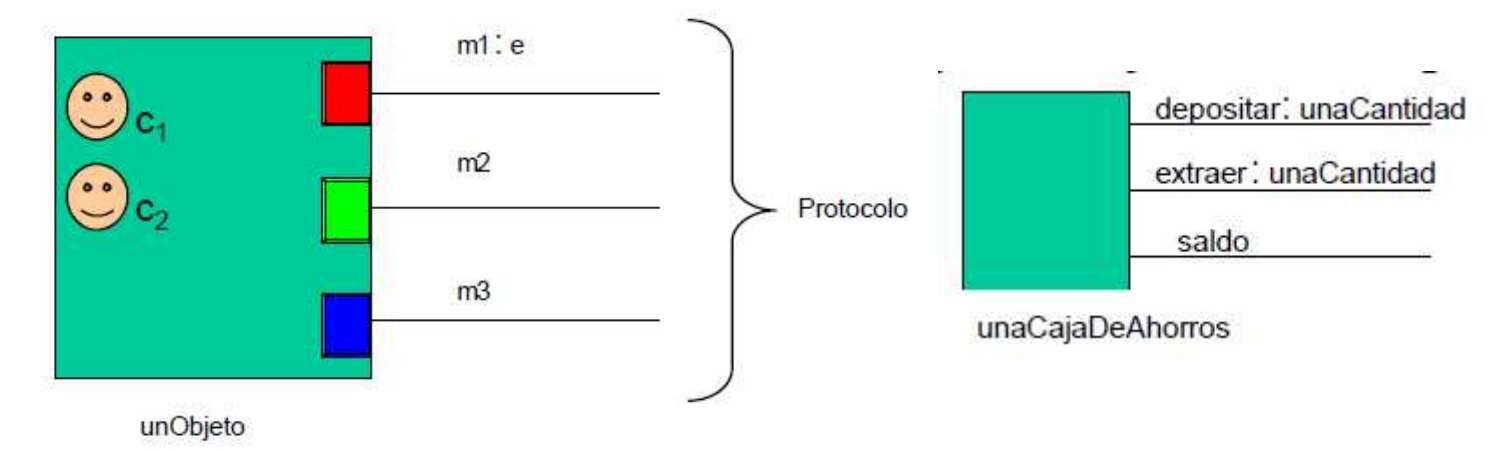
> "... en gran parte del libro me he valido del término "paradigma" en dos sentidos distintos. Por una parte, significa toda la constelación de creencias, valores, técnicas, etc., que comparten los miembros de una comunidad dada. Por otra parte, denota una especie de elemento de tal constelación, las concretas soluciones de problemas que, empleadas como modelos o ejemplos, pueden remplazar reglas explícitas como base de la solución de los restantes problemas de la ciencia normal."
>
> Thomas Kuhn, en "La estructura de las revoluciones científicas" (1969)

Introducción a Programación con Objetos

- Paradigma Orientado a Objetos. Características:
  - Estructura de desarrollo modular basada en **objetos**, que son definidos a partir de **clases**, como implementación de tipos abstractos de datos.
  - **Encapsulamiento** como forma de abstracción que separa las interfaces de las implementaciones de la funcionalidad del sistema (**métodos**) y oculta la información (**variables**).
  - Mecanismo de envío de **mensajes**, que posibilita la interacción entre los objetos y permite la **delegación** de responsabilidades de unos objetos a otros.
  - **Polimorfismo**, basado en el **enlace dinámico**, de forma que las entidades del programa puedan referenciar en tiempo de ejecución a objetos de diferentes clases.
  - **Herencia**, que permite que una clase sea definida como una extensión o modificación de otra.

Introducción a Programación con Objetos

- Objeto:
  - Los objetos son abstracciones que **representan las entidades** del mundo real que forman parte del **dominio del problema**, a los **componentes computacionales**, tanto de software como de hardware, y a toda unidad de información que sea necesaria para desarrollar un programa.
  - **“Todo” es pensado como un objeto**.
  - Al implementar los objetos mediante un lenguaje de programación, los atributos que conforman el **estado interno** se denominan **variables o colaboradores internos** y a las funcionalidades que conforman su **comportamiento** se las llama **métodos**. Estos pueden requerir de **colaboradores externos (a**yudan al objeto a responder un mensaje específico)



Introducción a Programación con Objetos

- Objetos como Abstracciones
  - Un objeto presenta:
    - Un **comportamiento** bien determinado
      - ¿Qué hace el objeto?
      - Representa la esencia del ente
    - Una **implementación** para ese comportamiento
      - ¿Cómo hace el objeto lo que hace?
      - Provee una posible implementación para esa esencia
    - Una **Identidad**
      - ¿Cómo podemos distinguir un objeto de otro?
      - Identidad vs. Igualdad.
- Ciclo de vida de un objeto:
  - Creación.
  - Uso.
  - Destrucción.

Introducción a Programación con Objetos

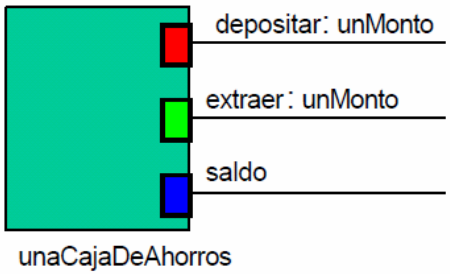
- Encapsulamiento
  - Un objeto **no conoce el funcionamiento interno** de los demás objetos y no lo necesita para poder interactuar con ellos, sino que le es suficiente con **conocer su interfase**, es decir saber la forma en que debe enviarles sus mensajes y como va a recibir la respuesta. Ante la modificación de una funcionalidad en particular del sistema, en la medida que su implementación este encapsulada en un objeto, el impacto que produce su cambio no afectara a los otros objetos que interactúan con él.

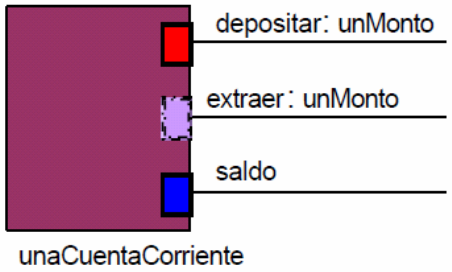
Introducción a Programación con Objetos

- Polimorfismo
  - Permite a dos o mas objetos distintos tengan definidos **métodos de igual nombre**, pero cada uno con su correspondiente implementación. Un objeto emisor puede comunicarse con cualquiera de estos objetos mandándole un mensaje donde se menciona el nombre del método, y el objeto receptor ejecutara la implementación que tenga definida, independientemente de la otras implementaciones que tengan los otros objetos. Desde el punto de vista del objeto emisor, tampoco le interesa conocer si ante la misma invocación hecha a diferentes objetos, la forma en que cada uno de ellos ejecuto el método internamente fue la misma o no.
  - Dos o más objetos son polimórficos respecto de un conjunto de mensajes, si todos pueden responder esos mensajes, aún cuando cada uno lo haga de un modo diferente.
  - Podemos decir que objetos polimórficos corresponden a un mismo “tipo” de objeto.
  - Mismo “tipo” significa mismo comportamiento esencial, independientemente de implementación.

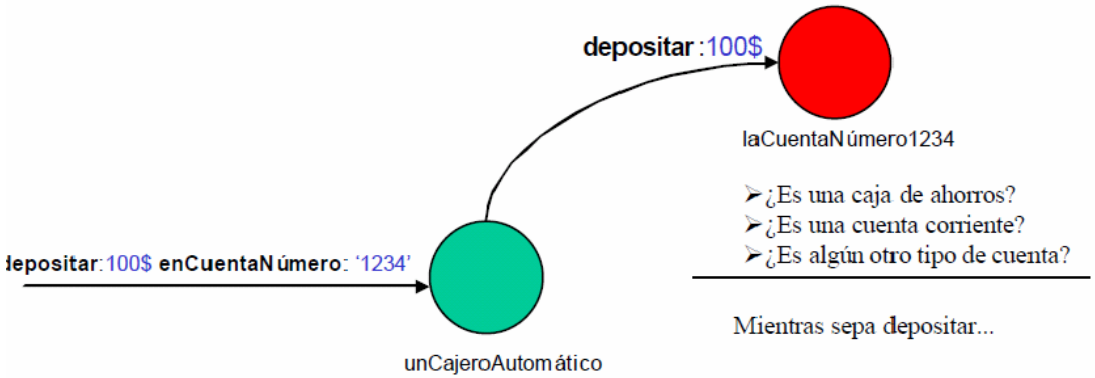
Introducción a Programación con Objetos

Dos objetos polimórficos...





Polimorfismo: ¿Para qué?



Introducción a Programación con Objetos

- Binding:
  - Binding Temprano o Estático
    - Operandos y operadores son ligados en tiempo de compilación.
    - Es el programador quién decide qué se debe ejecutar.
  - Binding Tardío o Dinámico
    - Operandos y operadores son ligados en tiempo de ejecución.
    - Es el objeto receptor del mensaje quién decide qué se debe ejecutar.
- El Polimorfismo es la clave de un buen diseño
  - “Código” genérico.
  - Objetos desacoplados.
  - Objetos intercambiables.
  - Objetos reusables.
  - Programar por protocolo, no por implementación (buscar la esencia).

Introducción a Programación con Objetos

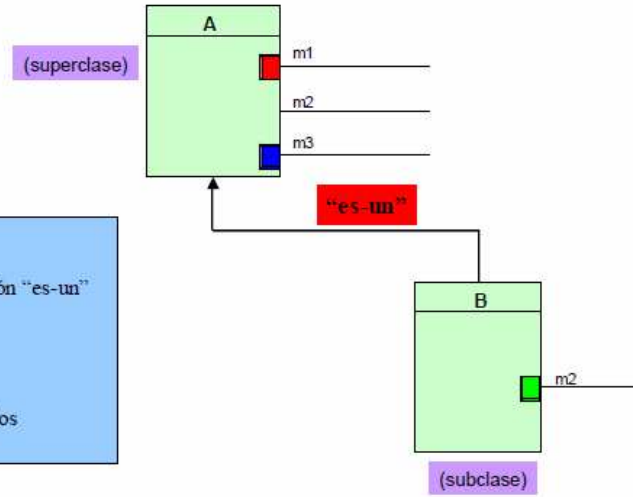
- Clase:
  - Es un objeto responsable de crear otros objetos (sus instancias)
  - **Describe** completa y detalladamente la **estructura de información** y el **comportamiento** que tendrá **todo objeto de esa clase**, o sea, define el conjunto de variables y de métodos que determinan como van a ser y como se van a comportar sus objetos.
  - ¿Qué sucede cuando las clases comparten parte del conocimiento que representan?
    - Subclasificación

Introducción a Programación con Objetos

- Relaciones entre clases:
  - Dependencia (Colaboradores Externos)
    - “utiliza un”
    - Pedido utiliza la clase Cuenta.
  - Composición (Colaboradores Internos) / Agregación
    - “tiene un”
    - Pedido contiene objetos de tipo Articulo.
  - Herencia (Compartir conocimiento)
    - “es un”
    - PedidoVigente es un Pedido.

Introducción a Programación con Objetos

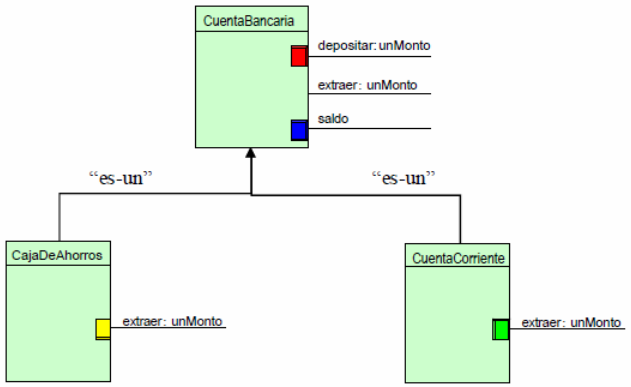
Subclasificación



Conceptos clave

- El significado de la relación "es-un"
- Clases Abstractas
- Clases Concretas
- Jerarquías Polimórficas
- Tipos
- Re-definición de protocolos
- Herencia

Una jerarquía de clases



Introducción a Programación con Objetos

- Lenguajes Orientados a Objetos. Características.
  - Básicas:
    - Todo es un objeto.
    - Todo objeto es instancia de una clase.
    - Todos los objetos de la misma clase pueden recibir los mismos mensajes.
    - Las clases se organizan en una estructura arbórea de raíz única, llamada jerarquía de herencia.
    - Un programa es un conjunto de objetos que se comunican mediante el envío de mensajes.
  - Opcionales:
    - Genericidad.
    - Gestión de errores.
    - Aserciones.
    - Tipado estático.
    - Recogida de basura.
    - Reflexión.

Introducción a Programación con Objetos

- Lenguajes Orientados a Objetos. Historia.
  - 1967. **Simula67**. Ole-Johan Dahl y Kristen Nygaard de Norwegian Computer Center (Noruega). Modelado y simulación de procesos industriales y científicos.
  - 1980. **Smalltalk**. Alan Kay. LOO puro.
  - 1980-1986:
    - Extensiones de Smalltalk: **Smalltalk/V**
    - Extensiones de C: **Objective C, C++** (Bjarne Stroustrup)
    - Extensiones de Pascal: **Object Pascal, Delphi**
    - Extensiones de LISP: **CLOS**
    - Extensiones de Simula: **Eiffel** (Bertrand Meyer) (puro)
  - 1986. Primera conferencia internacional sobre LOO.
  - A partir de los ‘90 proliferan con gran éxito la tecnología y LOO.
  - 1995. Java. James Gosling de Sun Microsystems. "*write once, run anywhere*"
  - Los mas implantados en la actualidad: **Java**, **C++** y **PHP**.
  - **C#, Python, Ruby, Delphi** son otros LOO muy utilizados.

Introducción a Programación con Objetos

- Caso de estudio: Envío de Flores
  - Ejemplo: Supongamos que Luis quiere enviar flores a Alba, que vive en otra ciudad.
    - Luis va a la floristería más cercana, regentada por un florista llamado Pedro.
    - Luis le dice a Pedro que tipo de flores enviar a Alba y la dirección de recepción.
  - El mecanismo utilizado para resolver el problema es encontrar un agente apropiado (Pedro)
  - Enviarle un mensaje conteniendo la petición (envía flores a Alba).
  - Es la responsabilidad de Pedro satisfacer esa petición.
  - Para ello, es posible que Pedro disponga de algún método (algoritmo o conjunto de operaciones) para realizar la tarea.
  - Luis no necesita (ni le interesa) conocer el método particular que Pedro utilizará para satisfacer la petición: esa información está OCULTA

Introducción a Programación con Objetos

- **Ejercicio 1:** Realizar un mapa conceptual con los conceptos vistos en la Unidad 1.
- **Ejercicio 2:** Identificar objetos y responsabilidades:
  - Supongamos que Luís quiere enviar flores a Alba, que vive en otra ciudad.
  - Luís va a la floristería más cercana, regentada por un florista llamado Pedro.
  - Luís le dice a Pedro que tipo de flores enviar a Alba y la dirección de recepción.
- **Ejercicio 3:** Identificar objetos y responsabilidades:
  - Cada usuario tiene una casilla de correos.
  - La correspondencia puede ser texto o imagen.
  - Queremos mostrar el contenido de una casilla.
  - Comparar:
    - El paradigma procedural o imperativo (C/Fortran/BASIC/Pascal)
    - El paradigma orientado a objetos (Java/C++/Smalltalk/Eiffel)
  - Agreguemos correspondencia de audio.

Introducción al Lenguaje de Programación Java

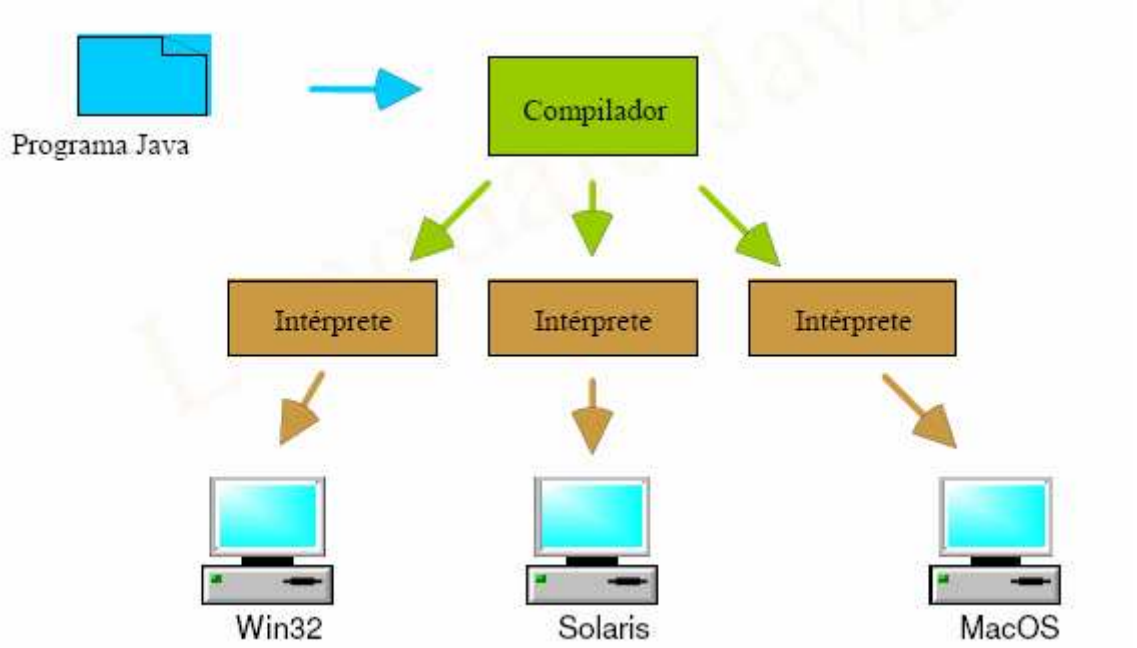
- Java abarca dos aspectos:
  - Una plataforma: es un ambiente de software y/o hardware sobre el que se ejecuta un programa.
    - Enorme biblioteca.
    - Código reutilizable.
    - Entorno de ejecución: seguridad, adaptabilidad, Garbage Collector
  - Un lenguaje: posibilita el desarrollo de aplicaciones seguras, robustas sobre múltiples plataformas en redes heterogéneas y distribuidas.
    - Sintaxis agradable.
    - Semántica comprensible.

Introducción al Lenguaje de Programación Java

- La plataforma Java se compone de:
  - La Java Virtual Machine (JVM): es la base de la Plataforma Java y puede ser incorporada en la mayoría de las plataformas (hardware y sistema operativo). Contiene el intérprete Java.
  - La Java Application Programming Interface (Java API): es una colección de componentes de software que proveen una amplia gama de funcionalidades, como GUIs, I/O, Seguridad, etc. Está dividida en paquetes o librerías de componentes relacionadas.

Introducción al Lenguaje de Programación Java

- La Plataforma Java (la API y la JVM) independiza al programa Java del hardware.



Introducción al Lenguaje de Programación Java

- Principales características del Lenguaje:
  - Simple y Familiar
  - Orientado a Objetos
  - Distribuido
  - Robusto
  - Seguro
  - Arquitectura neutral
  - Multithread
  - Interpretado
  - Dinámico

Introducción al Lenguaje de Programación Java

- Simple y Familiar
  - El objetivo de los diseñadores de Java fue crear un lenguaje de programación que fuera fácil de aprender. Java adoptó una sintaxis similar a la de C/C++ teniendo en cuenta su popularidad. Y eliminó aquellas características que son fuente de confusión.
  - Características de C/C++ eliminadas en Java
    - No más preprocesador
    - No más estructuras de datos ni uniones
    - No más tipos enumerativos
    - No más funciones
    - No más herencia múltiple de clases
    - No más GOTO
    - No más punteros

Introducción al Lenguaje de Programación Java

- Orientado a Objetos:
  - Java posee todas las características de un lenguaje orientado a objetos
    - Polimorfismo.
    - Encapsulamiento.
    - Herencia.
  - Java implementa herencia simple de clases. Una clase puede ser subclase de una única clase. Todas las clases heredan de la clase Object. (Todo es un objeto)
  - Java enriquece el modelo de herencia simple de clases, implementando interfaces.

Introducción al Lenguaje de Programación Java

- Distribuido:
  - Con Java es posible usar protocolos como HTTP y FTP para leer o copiar archivos remotos de manera tan simple como si estuviesen en el File System local.
  - El comportamiento distribuido de Java posibilita la colaboración y la distribución de carga de trabajo del sistema. Ejemplo: Applets.
  - RMI (Java Remote Method Invocation) provee una forma simple y directa de manejar objetos distribuidos.

Introducción al Lenguaje de Programación Java

- Seguro:
  - Mapa de memoria y alocación de memoria:
    - En Java no existe el concepto de punteros a memoria al estilo C y C++.
    - En Java el modelo de alocación de memoria es transparente al programador, ya que es controlado íntegramente por JVM.
  - Chequeos de seguridad en el Class Loader:
    - Las clases de la API Java no pueden ser sobrescritas por clases importadas desde la red.
    - Las clases importadas desde la red, se ubican en espacios de nombres privados.
  - Verificación del ByteCode: el intérprete Java chequea los archivos .class que vienen de la red, evaluando:
    - Que el código no falsifique punteros.
    - Que el código no viole restricciones de acceso.
    - Que el código no viole el acceso a los objetos usando casting.

Introducción al Lenguaje de Programación Java

- Arquitectura neutral:
  - Java fue diseñado para soportar aplicaciones que se ejecutan en ambientes de redes heterogéneos, independientemente de la plataforma de hardware y del sistema operativo.
  - La arquitectura neutral dada por los ByteCodes es el paso más importante hacia la portabilidad de los programas.
  - “Write once, run anywhere”.

Introducción al Lenguaje de Programación Java

- Multithreaded:
  - Un Thread es un flujo de control secuencial dentro de un programa. Java provee múltiples threads en un programa, ejecutándose concurrentemente y llevando a cabo tareas distintas.
  - La API Java contiene primitivas de sincronización.
  - Los múltiples hilos (threads) de ejecución permiten mejorar la interactividad y la performance del sistema. (Mejor respuesta interactiva y comportamiento de tiempo real)

Introducción al Lenguaje de Programación Java

- Interpretado y dinámico:
  - El compilador Java genera ByteCodes para la JVM. El intérprete, incorporado en la JVM es el que permite ejecutar el programa.
  - Los ByteCodes de Java pueden ejecutarse en cualquier plataforma que tenga la JVM implementada.
  - Java es dinámicamente extensible ya que las clases se linkean a medida que se necesitan y pueden ser cargadas dinámicamente a través de la red.

Introducción al Lenguaje de Programación Java

- Tipos de programas:
  - Applets: son pequeños programas que se ejecutan dentro de un web browser Java-compatible. (p.e. Netscape Navigator, Microsoft Internet Explorer, HotJava).
  - Aplicaciones: son programas comunes que se ejecutan utilizando solo a JVM como plataforma.
  - Servlets: son programas que corren dentro de un ambiente provisto por un “contenedor”. Los contenedores son servidores de aplicaciones. Su ejecución se desencadena escribiendo su URL en un browser web.

Introducción al Lenguaje de Programación Java

- Java es un lenguaje de la empresa SUN
- Primera versión 1.0 (1996)

Versión	Características	Nº Clases e Interfaces
1.0	El lenguaje en sí	211
1.1	Clases internas	477
1.2	Ninguna (Java 2) J2SE/EE/ME	1524
1.3	Ninguna	1840
1.4	Aserciones	2723
5.0	Clases genéricas, for each, argumentos variables, enumeraciones	3270

Introducción al Lenguaje de Programación Java

- Entornos de programación Java
  - JDK: Java Development Kit
  - Java SDK: Software Development Kit (versiones 1.2 a 1.4)
- Entornos integrado de desarrollo (IDE)
  - Forte
  - Sun ONE Studio
  - Sun Java Studio
  - Netbeans (IDE de SUN, principal competencia de Eclipse)
  - Eclipse (escrito en Java)
- Opciones de entorno:
  - JDK + editor de texto
  - IDE
- Uso de herramientas de la línea de comandos
  - Abrir ventana de shell
  - Ejecutar los siguientes comandos:
    - Javac Bienvenido.java
    - Java Bienvenido

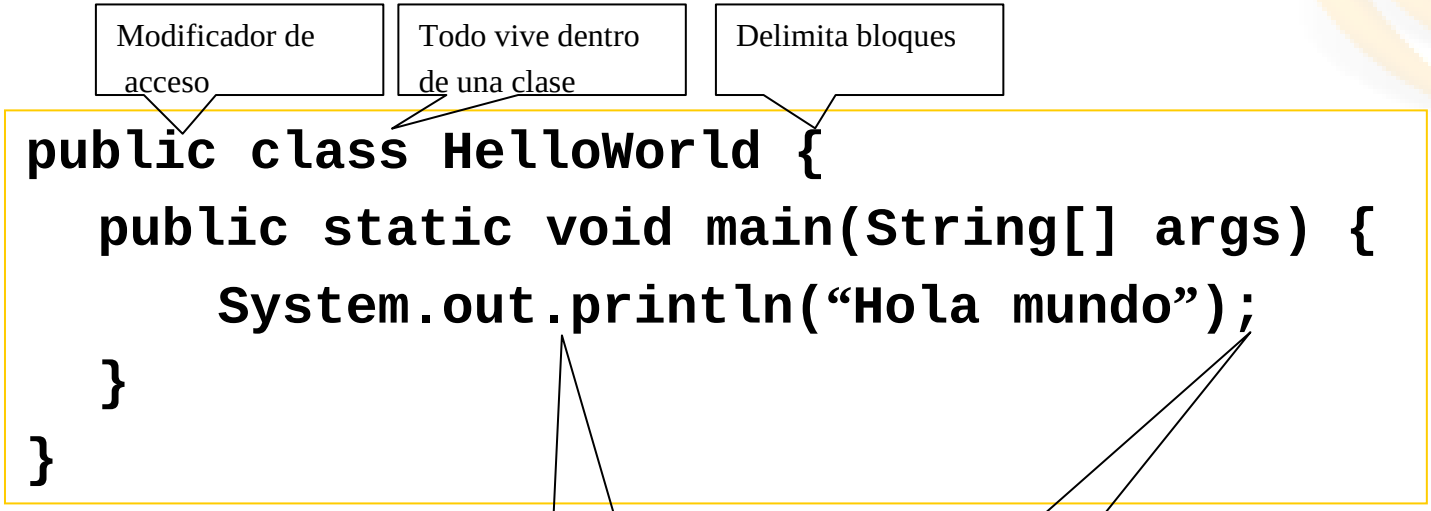
Introducción al Lenguaje de Programación Java

- Eclipse
  - Es un IDE abierto y extensible (plugins)
  - Un IDE es un programa compuesto por un conjunto de herramientas útiles para un desarrollador.
  - Software libre.
  - Gran parte de la programación Eclipse hecha por IBM.
  - Antecesor de Eclipse: VisualAge
  - En 2001 IBM y Borland crearon la Fundación Eclipse
  - Sitio web: www.eclipse.org

Versión	Año	Nombre
3.0	2004	Eclipse 3.0
3.1	2005	Eclipse 3.1
3.2	2006	Callisto
3.3	2007	Europa
3.4	2008	Ganymede
3.5	2009	Galileo
3.6	2010	Helio
3.7	2011	Indigo
4.2	2012	Juno
4.3	2013	Kepler
4.4	2014	Luna
?	2015	Mars

Introducción al Lenguaje de Programación Java

- La aplicación Hola Mundo



>>> javac HelloWorld.java

>>> java HelloWorld

>>> Hola mundo

JVM empieza la ejecución en el código que se encuentra en el método main
Eclipse → se ejecuta <CTRL>+<F11>

Introducción al Lenguaje de Programación Java

- En Java existen dos categorías de tipos de datos:
  - Tipos primitivos:
    - int — 32-bit complemento a dos
    - boolean — true o false
    - char — 16-bit caracteres Unicode
    - byte — 8-bit complemento a dos
    - short — 16-bit complemento a dos
    - long — 64-bit complemento a dos
    - float — 32-bit IEEE 754
    - double — 64-bit IEEE 754
  - Clases o interfaces
- Java es un lenguaje fuertemente tipado:
  - Comprobación estricta de tipos: todas las variables tienen que tener un tipo declarado.

Introducción al Lenguaje de Programación Java

- Para los tipos primitivos existen clases “wrappers”:
  - **int Integer**
  - **boolean Boolean**
  - **char Character**
  - **...**
- ¿Para qué utilizar las clases wrappers?
- Java no es un lenguaje orientado a objetos “puro”.

Introducción al Lenguaje de Programación Java

- Clase String:
  - Es una secuencia de caracteres.
  - No es un tipo primitivo.
  - Los String son instancias de la clase **java.lang.String**.
  - El compilador trata a los String como si fuesen tipos del lenguaje.
  - La clase tiene varios métodos para trabajar con ellos.
  - Son inmutables.
  - Como crear uno:

```
String saludo = “Hola”;
String otroSaludo = new String(“Como andás?”);
saludo.length();
saludo.charAt(0);
saludo.substring(0,2);
saludo.compareTo(“adios”);
saludo.equals(“Hol”);
saludo.split(“o”);
saludo.trim();
saludo.indexOf(“a”);
saludo.indexOf(“a”, 2);
saludo.toLowerCase();
saludo.toUpperCase();
saludo.equalsIgnoreCase(“h”);
saludo.endsWith(“la”);
```

Introducción al Lenguaje de Programación Java

Operadores:

- Asignación:
  i = i + 1;
  j += k;
- Aritméticos:
  i + 4 * f
  j – (8 / b)
- Lógicos:
  a && b
  d || c
  !a
- Relacionales:
  2 >= p
  a == 5
  b != 8
- Otros:
  i++
  j--
  [] . () (refundición) new instanceof
- Condicional:
  (x<y)?x:y

Introducción al Lenguaje de Programación Java

- Arreglos:
  - Todo arreglo es de un tipo o una clase.
  - Se los instancia con el operador **new**
  - Son “zero-based”
  - Ejemplos:

```
char[] carácteres;
String[] palabras = new String[10];
float[] arreglo = {2, 4, 6};
int[][] matriz = {{1}, {3, 4}};
carácteres = new char[]{‘a’, ‘1’};
palabras[2] = “chango”;
```

Introducción al Lenguaje de Programación Java

- Parámetros de la línea de comandos

```
public class Mensaje {
   public static void main(String[] args) {
       if (args[0].equals(“-h”))
        System.out.println(“Hola ”);
       if (args[0].equals(“-a”))
        System.out.println(“Adiós ”);
       for(int i=1; i<args.length; i++)
        System.out.println(“ ” + args[i]);
       System.out.println(“!”);
   }
}
```

Introducción al Lenguaje de Programación Java

- Parámetros de la línea de comandos:

>>> java Mensaje –h mundo cruel

>>> Hola, mundo cruel!

>>> java Mensaje –a mundo cruel

>>> Adiós, mundo cruel!

- Operaciones de arreglos:

```
int [] a = new int [10000];
…
Arrays.sort(a);
Arrays.binarySearch(a, 3);
Arrays.fill(a, 3);
```

Introducción al Lenguaje de Programación Java

Estructuras de control:

- if:

```
if (condición) {
        sentencias;
}
```

- while:

```
while (condición) {
        sentencias;
}
```

- for:

```
for (int i=0; i < 8; i++) {
        sentencias;
}
for (tipo elemento: colección) {
        sentencias; // por c/ elemento
}
```

- if else:

```
if (condición) {
        sentencias
} else {
        sentencias
}
```

- switch:

```
switch (i) {
        case 1:{sentencias;[break;]}
        case 2:{sentencias;[break;]}
        case 3:{sentencias;[break;]}
}
```

enteros, caracteres o constantes enumeradas

Rompe el flujo de control.

Introducción al Lenguaje de Programación Java

- Comentarios:
  - No aparecen en el programa ejecutable.
  - Existen tres tipos:
    - Por línea: //
    - Bloque de código: /* */
    - JavaDoc: /** */ Genera automáticamente la documentación.
      - En HTML a partir del Programa Fuente.
      - Vista en Eclipse.
      - Marcadores:
        - @param
        - @return
        - @throws
        - @exception
        - @author
        - @version
        - @deprecated

Introducción al Lenguaje de Programación Java

- Entrada y Salida
  - Entrada de datos por teclado:

```
Scanner in = new Scanner(System.in);
in.nextLine(); //lee una línea de entrada
in.next();     //lee una sola palabra
in.nextInt();  //lee un entero
in.nextDouble(); //lee número de coma
                    flotante
in.hasNext();    //si hay o no otra palabra
in.hasNextInt(); //si hay o no otro entero
in.hasNextDouble(); //si hay o no otro
                       número de coma flotante
```

Introducción al Lenguaje de Programación Java

- Entrada y Salida
  - Salida de datos a consola:

```
System.out.print(“Hola mundo”);
System.out.println(“Hola mundo”);

Double x = 10000.0/3.0;
System.out.println(x); //3333.3333333
System.out.printf(“%8.2f”, x); //3333.33
System.out.printf(“%,.2f”, x); //3,333.33

String.format(“Hola, %s. El año que viene
   tendrás %d”, nombre, edad);
```

Introducción al Lenguaje de Programación Java

- **Ejercicio 1:** Recorrer un **arreglo** de números enteros de dimensión N sumando en una variable los números **pares** y en otra los **impares**. Recorrerlo utilizando un **for** y luego utilizando un **while**.
- **Ejercicio 2:** Imprimir los 20 primeros números enteros positivos.
- **Ejercicio 3:** Sumar los 1000 primeros números naturales (1+2+3+4+...+1000), imprimiendo por cada suma el resultado parcial obtenido.

Encapsulamiento - Clase

- Clase
  - Cada clase, excepto la clase Object, es una extensión (subclase), de una sola clase ya existente (herencia simple).
  - En Java, una clase se compone de:
    - Declaración
    - Cuerpo

Encapsulamiento - Clase

- Forma general de declaración de una Clase:

```
package nombrePaquete;
{importaciones}
[modificadores] class nombreClase
[extends nombreSuperClase]
[implements nombresInterfaces] {
}
```

- Si una clase no declara explícitamente su superclase, entonces se asume que extiende a la clase Object.

Encapsulamiento - Paquete

- **nombrePaquete**:
  - Todo nombre de paquete debe por convención comenzar con una letra minúscula.
  - Java utiliza caracteres Unicode: **fácil**, **araña**, **vistaCliente2** son nombres de paquetes válidos.
  - El alcance de un identificador de paquete es todo el paquete en donde se declara, por lo tanto no puede haber dos paquetes con el mismo nombre dentro de un mismo paquete.

Encapsulamiento - Clase

- **Modificadores**:
  - Existen tres modificadores:
    - **public**
    - **abstract:** clases que no se pueden instanciar. Toda clase que tenga 1 o mas métodos abstractos tiene que declararse abstracta a su vez. Puede tener atributos y métodos concretos.
    - **final:** clases que no se pueden extender. Ejemplo String. Todos los métodos de una clase final son final automáticamente NO así los atributos.
  - Son opcionales
  - Se pueden combinar, salvo **abstract** y **final**.

Encapsulamiento - Clase

- **nombreClase**:
  - Todo nombre de clase debe por convención comenzar con una letra mayúscula.
  - Java utiliza caracteres Unicode: **Inútil**, **Pequeña**, **NegroEl8** son nombres de clases válidos.
  - El alcance de un identificador de clase es todo el paquete en donde se declara la clase, por lo tanto no puede haber dos clases con el mismo nombre dentro de un mismo paquete.

Encapsulamiento - Clase

- Cuerpo
  - Se considera como cuerpo de una clase todo lo encerrado entre “{” y “}”.
  - En el cuerpo se declaran:
    - Atributos
    - Constructores
    - Métodos
  - Otras clases

Encapsulamiento - Clase

```
package muebles;

public class MesaAlgarrobo
        extends Mesa {
...
}
```

```
package muebles;

abstract class Mesa {
...
}
```

```
package cliente;

public final class Conexión
   implements ErrorListener
   {
...
}
```

```
package servidor;

public abstract class Estado
   {
...
}
```

Lenguaje Java

Encapsulamiento - Constructor

- Constructores, declaración

```
package nombrePaquete;
public class NombreClase {
[accesibilidad] NombreClase
   ([parámetros])[throws excepciones] {
   ...
   }
}
```

- Los constructores:
  - No son métodos
  - Tienen el mismo nombre que su clase
  - No se heredan de las superclases

Encapsulamiento - Constructor

- Constructores, accesibilidad:

Accesibilidad	Clase	Paquete	Subclase	Mundo
private	X			
package	X	X		
protected	X	X	X	
public	X	X	X	X

```
public class MesaAlgarrobo extends Mesa {
   public MesaAlgarrobo() {}
}
abstract class Mesa {
   Mesa() {}
}
```

Encapsulamiento - Constructor

- Constructores :
  - Si una clase no declara ningún constructor, entonces tiene el constructor por defecto.
  - Números se inicializan en 0.
  - Booleanos se inicializan en false.
  - Objetos se inicializan en null.

```
public class Punto {
}
```

=

```
public class Punto {
  public Punto() {
    super();
  }
}
```

Encapsulamiento - Constructor

- Constructores:
  - Una clase puede tener varios constructores (overloading).
  - Los diferentes constructores se diferencian por el número y tipo de los parámetros.
  - Los constructores tienen el mismo manejo de excepciones que los métodos.

```
public class Círculo {
 protected Círculo(Punto punto) {...}
 public Círculo(Punto punto, Color color) {
  this(punto); ...
  }
}
```

Encapsulamiento - Atributo

- Variables:
  - Todas las variables tienen un tipo. El tipo puede ser:
    - Tipo primitivo
    - Clase
    - Interfaz

- Declaración en una clase:

```
[accesibilidad] [modificadores] tipo nombre
   [= valor];
```

- Declaración en un método:

```
[final] tipo nombre [= valor];
```

Encapsulamiento - Atributo

- Variables, accesibilidad:

Accesibilidad	Clase	Paquete	Subclase	Mundo
private	X			
package	X	X		
protected	X	X	X	
public	X	X	X	X

```
public class MesaAlgarrobo extends Mesa {
   private Color color;
   bool redonda;
}
```

Encapsulamiento - Atributo

- Variables, modificadores:
  - **static**: variables de clase. Existen desde que el ClassLoader carga la clase. Pertenecen a la clase y no a ningún objeto individual (variable compartida)
  - **transient**: no persistentes.
  - **volatile**: indica a la JVM que la variable puede ser modificada en forma asincrónica por cualquier thread.
  - **final**: constantes. Deben recibir valor inicial cuando se construye el objeto. En lo sucesivo, el campo o atributo no podrá ser modificado. Ejemplo: Math.PI
    - public static final double PI = 3.14….;

Encapsulamiento - Atributo

- Variables:
  - Nombre:
    - Compuesto de caracteres Unicode.
    - Por convención los nombres de las variables empiezan con minúscula.
    - Como en las clases, si se juntan varias palabras, al principio de cada una se coloca mayúscula.
    - Case Sensitive.

Encapsulamiento - Atributo

- Variables:
  - Visibilidad: bloque de código en donde es accesible la variable
  - Hay 3 categorías:

```
public class MyClass {
   String s;
   public void miMétodo(boolean b) {
      int s;
      if (b) {
         long s;
      }
   }
}
```

Encapsulamiento - Método

- Métodos, definición:
  - Consta de dos partes:
    - Declaración: se declaran los modificadores, tipo de retorno, nombre, lista de excepciones.
    - Cuerpo: se declaran las variables locales y el código del método.
      - Las variables locales de un método siempre deben recibir un valor inicial explícito.

```
[accesibilidad] [modificadores] tipoRetorno
  nombreMétodo([parámetros]) [throws
  excepciones] {
  ...
}
```

- Ejemplo: **public boolean isEmpty() {...}**

Encapsulamiento - Método

- Métodos, modificadores:

Accesibilidad	Clase	Paquete	Subclase	Mundo
private	X			
package	X	X		
protected	X	X	X	
public	X	X	X	X

  - Se declaran métodos privados cuando:
    - Demasiado cerca de la implementación
    - Requieren un determinado orden de llamada
    - Se utilizan en las operaciones de la propia clase

Encapsulamiento - Método

- Métodos, modificadores:
  - **abstract**: Un método abstracto no tiene implementación. Debe ser miembro de una clase abstracta. Actúan como reservas de espacio para los métodos que se implementan en las subclases.
  - **static**: Declara al método como método de clase. No necesita de un objeto que lo controle. Ejemplo: Math.pow
    - Métodos factoría: Métodos estáticos que retornan objetos de su propia clase. Ejemplo: NumberFormat.getCurrencyInstance()
  - **final**:El método no puede ser redefinido por las subclases.
  - **native**: El método está implementado en otro lenguaje.
  - **synchronized**: Permite que múltiples objetos invoquen el mismo método con exclusión mutua.

Encapsulamiento - Método

- Métodos, tipo de retorno:
  - Todo método debe tener un tipo de retorno o **void** si el método no devuelve nada.
  - Ejemplos:
    - **public void beOn()**
    - **public boolean isEmpty()**
    - **public Object[] getElements()**
    - **public Enumeration elements()**
    - **public Vector getElements()**

Encapsulamiento - Método

- Métodos, nombre:
  - Cualquier identificador válido puede ser nombre de método. Caracteres Unicode.
  - Los nombres de los métodos empiezan con minúscula.
  - Pueden existir múltiples métodos con el mismo nombre. Los métodos se diferencian por el número y tipo de los parámetros. (Sobrecarga de métodos)
  - Se recomienda el uso de getters y setters:
    - getX() Método de acceso o consulta. No escribir métodos de acceso que devuelvan referencias de objetos que se pueden modificar. Solución: clonarlo (.clone())
    - setX() Método de modificación

Encapsulamiento - Método

- Métodos, lista de parámetros:
  - Es una lista delimitada por coma de la forma "**tipo parámetro**".
  - Los tipos primitivos se pasan por valor.
  - Los objetos usan pasaje de referencias por valor.
  - No se puede declarar una variable dentro de un método con el mismo nombre que un parámetro.
  - Ejemplo:
    - **void unMétodo(int x, int y, String s)**

Encapsulamiento - Método

- Métodos, lista de parámetros variables:
  - Número variable de argumentos (varargs)
  - Ejemplo:
    - **void unMetodo(Object…args) {**

      //args es manipulado como un arreglo de Objects

      }
    - Invocaciones al método:
      - objeto.unMetodo(new Punto());
      - objeto.unMetodo(new Punto(), new Integer(3));
      - objeto.unMetodo(new Integer(3), “Hola”, “Mundo”);

Encapsulamiento - Método

- Métodos, lista de excepciones:
  - Java plantea un esquema de excepciones estricto.
  - Contiene todas las excepciones que se pueden levantar a raíz de la ejecución del método.
  - Cuando se invoca un método que puede levantar una excepción, hay que incluirlo dentro de una cláusula try – catch.

Encapsulamiento - Método

- Métodos, cuerpo:
  - Se considera como cuerpo todo lo encerrado entre “**{**” y “**}**”.
  - Las variables locales enmascaran a las variables miembro de la clase.
  - Las variables declaradas dentro de los métodos duran lo que dura el método.
  - **this**:
    - Se refiere al objeto actual.
    - Sólo puede aparecer en el cuerpo de un método de instancia, en un constructor (invoca a otro constructor de la misma clase) o en la inicialización de una variable de instancia.
  - **super**:
    - Se refiere a la superclase del objeto actual.
  - **return**:
    - Se utiliza para devolver un valor en los métodos que devuelven algo (no **void**).

Encapsulamiento – Clases Predefinidas

- Clases Predefinidas
  - Math

```
Math.sqrt(double):double
Math.pow(double, double):double
Math.max(int, int):int
Math.min(int, int):int
Math.random():double [0,1)
```

  - Date

```
Date fecha = new Date();
```

  - GregorianCalendar

```
GregorianCalendar hoy = new GregorianCalendar();
hoy.get(Calendar.MONTH)
```

Encapsulamiento - Objeto

- Ciclo de vida de un objeto. Fases:
  - Creación
  - Utilización
  - Finalización

Encapsulamiento - Objeto

- Creación:
  - Se lleva a cabo utilizando los constructores.

    Rectangle rect = new Rectangle();
  - Esta sentencia realiza:
    - Declaración
    - Instanciación
    - Inicialización

Encapsulamiento - Objeto

- Creación:
  - **new** es el operador de Java que aloca espacio para un nuevo objeto.
  - Luego del **new** se coloca un constructor con sus parámetros si los tuviera.
  - Ejemplos:

    new Rectangle(10,10);

    new Rectangle(new Point(10, 10), 10, 10);

Encapsulamiento - Objeto

- Utilización de objetos:
  - Invocación de métodos, acceso a variables y constantes:
  - En clases:
    - **Movible.ORIGEN;**
    - **Math.abs(-1);**
  - En instancias:
    - **rect.width();**
    - **new Point(10,10).x;**

Encapsulamiento - Objeto

- Finalización:
  - Conceptualmente existen dos variantes:
    - El usuario se encarga de liberar los recursos (Delphi).
    - El sistema es el encargado de liberar los recursos (Smalltalk, Java).

Encapsulamiento - Objeto

- Finalización:
  - Las referencias mantenidas en variables locales son liberadas cuando salen del alcance o cuando se les asigna **null**.
  - Los objetos son recolectados cuando no existen más referencias a ellos.
  - Antes de destruir un objeto el Garbage Collector llama al método **finalize()** de dicho objeto. Se utiliza para liberar recursos.

Encapsulamiento - Paquete

- Paquetes, descripción:
  - Un paquete es un conjunto de clases e interfaces relacionadas que proveen acceso protegido y administración de nombres.
  - Las clases e interfaces que son parte del lenguaje están agrupadas en paquetes de acuerdo a su función:
    - **java.lang**: Clases del lenguaje. Se importa por defecto.
    - **java.io**: para manejo de Entrada/Salida
  - El programador agrupa sus clases e interfaces en paquetes, anteponiendo la cláusula **package** NombreDelPaquete**;** a las declaraciones de todas las clases e interfaces agrupadas.
  - Se recomienda utilizar el nombre de dominio de Internet al revés. Ejemplo: ar.edu.uno.programacion.uno.xxx.
  - No puede haber dos paquetes con el mismo nombre dentro de un mismo paquete.

Encapsulamiento - Paquete

- Paquetes, Creación:
  - Un paquete es creado simplemente incorporando una clase o una interfaz.
  - Se requiere escribir la sentencia **package** como primer sentencia del archivo fuente en donde se está definiendo la clase o la interfaz.

```
package graphics;
class Circle extends Graphics
        implementes Draggable {
  ...
}
```

```
package graphics;
class Rectangle extends Graphics
          implementes Draggable {
  ...
}
```

Encapsulamiento - Paquete

- Paquetes, beneficios:
  - Reconocer y encontrar fácilmente las clases e interfaces relacionadas.
  - Evitar conflictos de nombres, ya que cada clase pertenece a un único paquete (no puede haber dos clases con el mismo nombre dentro de un mismo paquete).
  - Controlar el acceso a las clases del paquete.
  - Si no se usa la sentencia **package**, las clases e interfaces se ubican en el paquete por defecto (default package), que es un paquete sin nombre. No es recomendable utilizar este paquete.

Encapsulamiento - Paquete

- Paquetes, accesos:
  - Las clases e interfaces miembros de un paquete declarados públicos pueden ser accedidos desde afuera del paquete, de alguna de las siguientes formas:
    - Refiriéndose a su nombre largo:

      graphics.Rectangle miRect = new graphics.Rectangle();
      - Es la solución al conflicto del mismo nombre de clase en paquetes importados.
    - Importándolo:

      import graphics.Circle;

      Circle miCir = new Circle();
    - Importando el paquete íntegro:

      import graphics.*;

      Circle miCir = new Circle();

      Rectangle miRec = new Rectangle();
      - *: no tiene efectos negativos sobre el tamaño del código.

Encapsulamiento - Paquete

- Paquetes, accesos estáticos:
  - Permiten llamar a un método o propiedad estática sin necesidad de hacer referencia al nombre de su clase.
    - La sintaxis general, es:
      - import static paquete.Clase.metodo_o_propiedad_static; //Para un sólo método o propiedad.
      - import static paquete.Clase.*; //para todos los elementos estáticos de la clase
    - Ejemplo:

      import static java.lang.System.*;

      out.println(“Hola”);

      exit(0);

Encapsulamiento - Enum

- Clases de enumeración:
  - Para declarar variables con un conjunto restringido de valores.
  - Enum es una clase.
  - El enum posee objetos.
  - Ejemplo:

```
enum Porte {MINI, MEDIANO, GRANDE,
   EXTRA_GRANDE};
```

```
Porte s = Porte.MEDIANO;
```

Encapsulamiento - Enum

- Clases de enumeración:
  - Ejemplo:

```
public enum Talla
{ MINI(“S”), MEDIANO(“M”), GRANDE(“L”),
  EXTRA_GRANDE(“XL”);
  private Talla(String abrev){
      this.abrev = abrev;
  }
  public String getAbreviatura() {
      return abrev;
  }
  private String abrev;
}
```

Encapsulamiento - Reflection

- Reflection, definición:
  - Es la habilidad que tiene un programa para realizar algún tipo de computación sobre sí mismo.
  - También se conoce como introspección.
  - Para esto es necesario representar los elementos que forman parte de los sistemas de objetos:
    - Clases
    - Mensajes
  - Es un mecanismo potente y complejo.
  - Interés especial para los constructores de herramientas y no para los programadores de aplicaciones.

Encapsulamiento - Reflection

- Reflection, tipos:
  - Structural Reflection:
    - Referida a las propiedades estructurales de los sistemas, como jerarquías de clases, tipos, etc. (versión provista por Java)
  - Behavioral Reflection:
    - Que ataca el comportamiento de los objetos en el sistema. Utilizada para monitorear y modificar dicho comportamiento.

Encapsulamiento - Reflection

- Reflection, funcionalidad:
  - Determinar la clase de un objeto.
  - Obtener información acerca de las clases: modificadores, constructores, métodos, variables, superclases e interfaces que implementan.
  - Obtener información acerca de las interfaces: declaración de métodos y constantes.
  - Invocar métodos que no son conocidos hasta el tiempo de ejecución.
  - Crear instancias de clases que son desconocidas hasta el tiempo de ejecución.

Encapsulamiento - Reflection

- ¿Cómo obtener instancias de la clase Class?
  - Class c = mystery.getClass();
  - TextField t = new TextField();
  - Class c = t.getClass();
  - Class s = c.getSuperclass();
  - Class c = java.awt.Button.class;
  - Class c = Class.forName(strg);
- Creación de un objeto:
  - c.newInstance() //Llama al constructor predeterminado.

Encapsulamiento - Reflection

- Descubriendo los modificadores:

```
public void printModifiers(Object o) {
        Class c = o.getClass();
        int m = c.getModifiers();
        if (Modifier.isPublic(m))
                System.out.println("public");
        if (Modifier.isAbstract(m))
                System.out.println("abstract");
        if (Modifier.isFinal(m))
                System.out.println("final");
}
```

Encapsulamiento - Reflection

- Recuperando las variables:

```
public void printPublicFieldNames(Object o) {
        Class c = o.getClass();
        Field[] publicFields = c.getFields();
        for (int i = 0; i < publicFields.length; i++) {
                String fieldName = publicFields[i].getName();
                Class typeClass = publicFields[i].getType();
                String fieldType = typeClass.getName();
                System.out.println("Name: " + fieldName +
                ", Type: " + fieldType);
        }
}
```

Encapsulamiento - Reflection

- Recuperando los métodos:

```
public void showMethods(Object o) {
        Class c = o.getClass();
        Method[] methods = c.getMethods();
        for (int i = 0; i < methods.length; i++) {
                String name = methods[i].getName();
                System.out.println("Name: " + name);
                String return = methods[i].getReturnType().getName();
                System.out.println("Return Type: " + return);
                Class[] paramTypes = methods[i].getParameterTypes();
                System.out.print("Parameter Types:");
                for (int k = 0; k < paramTypes.length; k ++) {
                        String parameterString = paramTypes[k].getName();
                        System.out.print(" " + parameterString);
                }
                System.out.println();
        }
}
```

Encapsulamiento - Reflection

- Invocación de métodos:

```
Method metodo = Empleado.class.getMethod("getNombre");
System.out.println(metodo.invoke(new Empleado()));
metodo = Math.class.getMethod("sqrt", Double.class);
(Double) metodo.invoke(null, 9);
```

Encapsulamiento - Genericidad

- Consiste en escribir código que se puede reutilizar para objetos de muy distintos tipos.
- Antes de JDK 5.0, la programación genérica en Java siempre se realizaba por herencia (Object). Problemas:
  - Se necesita una refundición siempre que se recupera un valor.
  - No hay comprobación de errores. Un error de compilación es mucho mejor que una excepción de refundición (ClassCastException) durante la ejecución.
- JDK 5.0 ofrece una solución mejor: los parámetros de tipo.
- Ventajas:
  - Programas mas fáciles de leer y mas seguros (no se necesita refundir).
- Desventajas:
  - No es tan fácil implementarlas.

Encapsulamiento - Genericidad

- Convención de nombres de las variables de tipo:

E	Tipo de elemento de las colecciones. Ejemplo: ArrayList<E>
K	Tipo de clave. Ejemplo: Hashtable<K,V>
V	Valor de las tablas. Ejemplo: Dictionary<K,V>
T,U,V	Absolutamente cualquier tipo. Ejemplo: Comparable<T>

Encapsulamiento - Genericidad

- Definición de una clase genérica:
  - Es una clase que tiene una o mas variables de tipo.
  - Ejemplo:

```
public class Pareja <T> {
   private T prim;
   private T seg;
   public Pareja() {prim=null; seg=null;}
   public Pareja(T p, T s) {prim=p; seg=s;}
   public T getPrim() {return prim;}
   public T getSeg() {return seg;}
}
```

Encapsulamiento - Genericidad

- Los tipos genéricos se particularizan reemplazando las variables de tipo por tipos, de la forma:

```
Pareja<String> obj  = new Pareja<String>();
Pareja<String> obj2 = new Pareja<String>("1", "2");
String primero = obj2.getPrim();
```

Encapsulamiento - Genericidad

- Métodos genéricos:
  - Se puede definir un único método con parámetros de tipo.

```
public class AlgMatrices {
   public static <T> T getInicial(T[] a) {
       return a[a.length/2];
   }
}
```

```
String [] vocales = {“a”, “e”, “i”};
String inicial =
   AlgMatrices.<String>getInicial(vocales);
```

Encapsulamiento - Genericidad

- Límites para las variables de tipo:
  - Restringir T a una clase que implemente una interfaz o a una subclase. Ejemplo:

```
public static <T extends Comparable> T min(T[] a){
   ...
}
```

  - Las variables o comodines de tipo pueden tener múltiples límites, por ejemplo:

```
<T extends Comparable & Serializable>
```

Encapsulamiento - Genericidad

- Límites para las variables de tipo:
  - No es posible reemplazar un parámetro de tipo por un tipo primitivo.
  - No es posible lanzar ni capturar objetos de clases genéricas.
  - No se permite que una clase genérica extienda a Throwable.
  - No es posible declarar vectores de tipos parametrizados. Ejemplo:
    - Pareja<String> [] v = new Pareja<String>(10); //Error
  - No se pueden crear objetos de tipos genéricos. Ejemplo:
    - new T(); //Error
  - No es posible hacer referencias a variables de tipo en atributos o métodos estáticos. Ejemplo:
    - private static T ejemplarUnico; //Error

Encapsulamiento - Genericidad

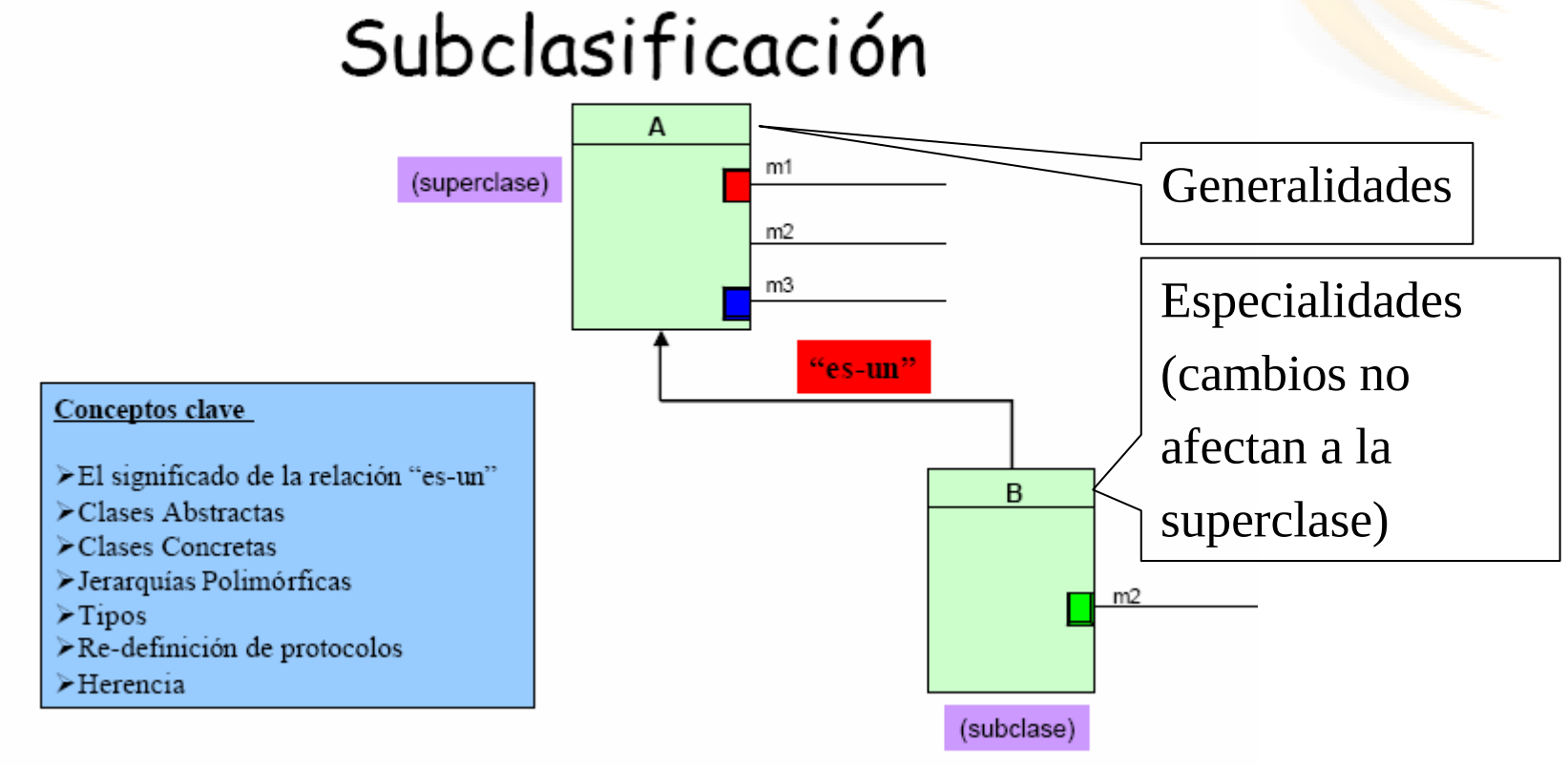
- Tipo comodín:
  - Denota cualquier tipo genérico de Pareja cuyo parámetro de tipo sea una subclase CuentaBancaria, tal como Pareja<CajaDeAhorro>, pero no Pareja<String>

```
Pareja<? extends CuentaBancaria>
```

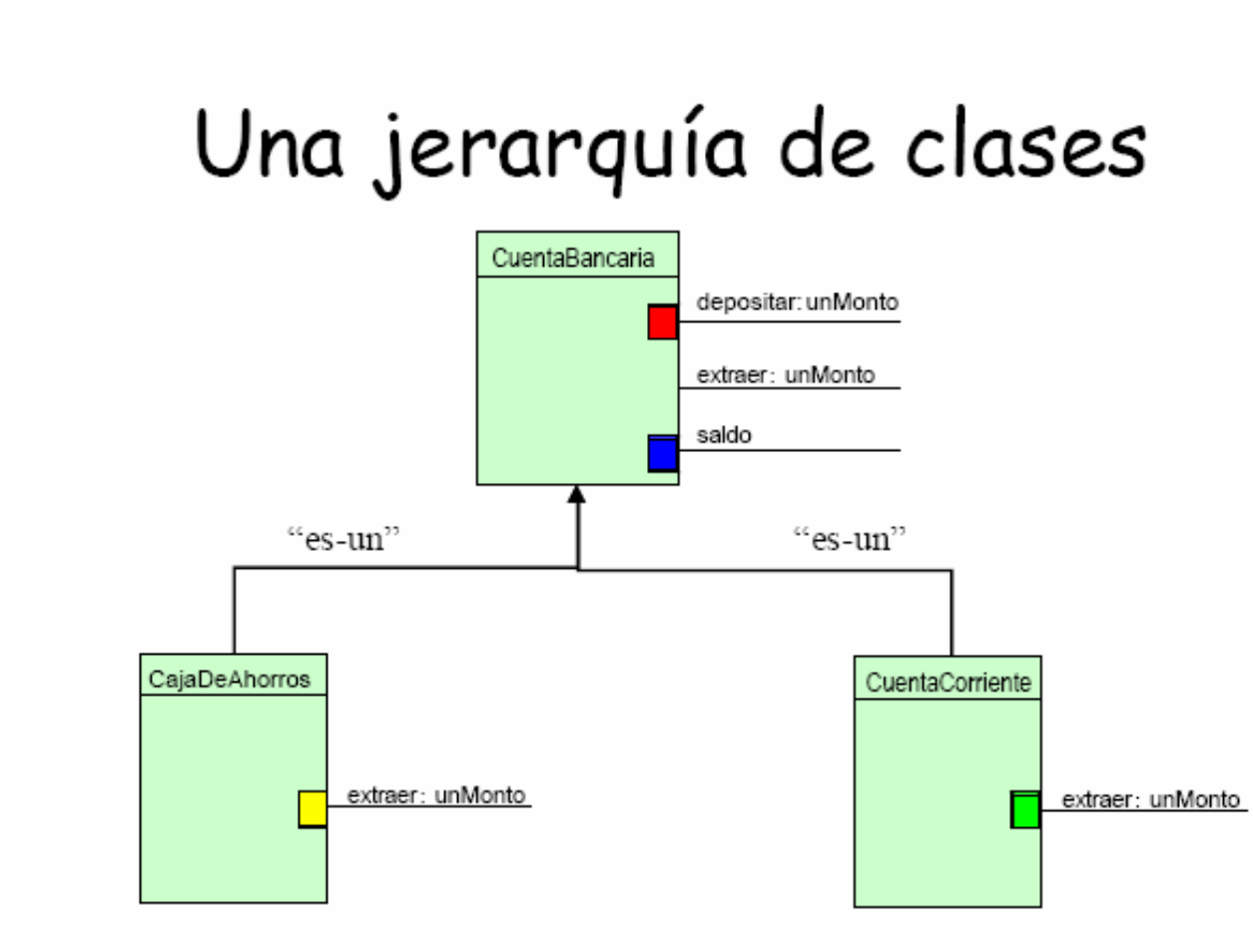
Herencia y Polimorfismo - Herencia

- Usamos las clases para representar parte del conocimiento que adquirimos del dominio de problema
- ¿Qué sucede cuando las clases comparten parte del conocimiento que representan?
  - – Subclasificación
- Crear clases nuevas que se construyan tomando como base clases ya existentes.
- Cuando se hereda, se reutilizan métodos y atributos (pueden ser statics)
- Permite extender la funcionalidad de un objeto.

Herencia y Polimorfismo - Herencia



Herencia y Polimorfismo - Herencia



Herencia y Polimorfismo - Herencia

```
public class CajaDeAhorro extends
   CuentaBancaria {
}
```

- CajaDeAhorro es subclase/clase derivada / clase hija
  - Especialidades
  - Tiene mas funcionalidad
  - Cambios no afectan a la superclase.
- CuentaBancaria es superclase / clase base / clase padre
  - Generalidades

Herencia y Polimorfismo - Herencia

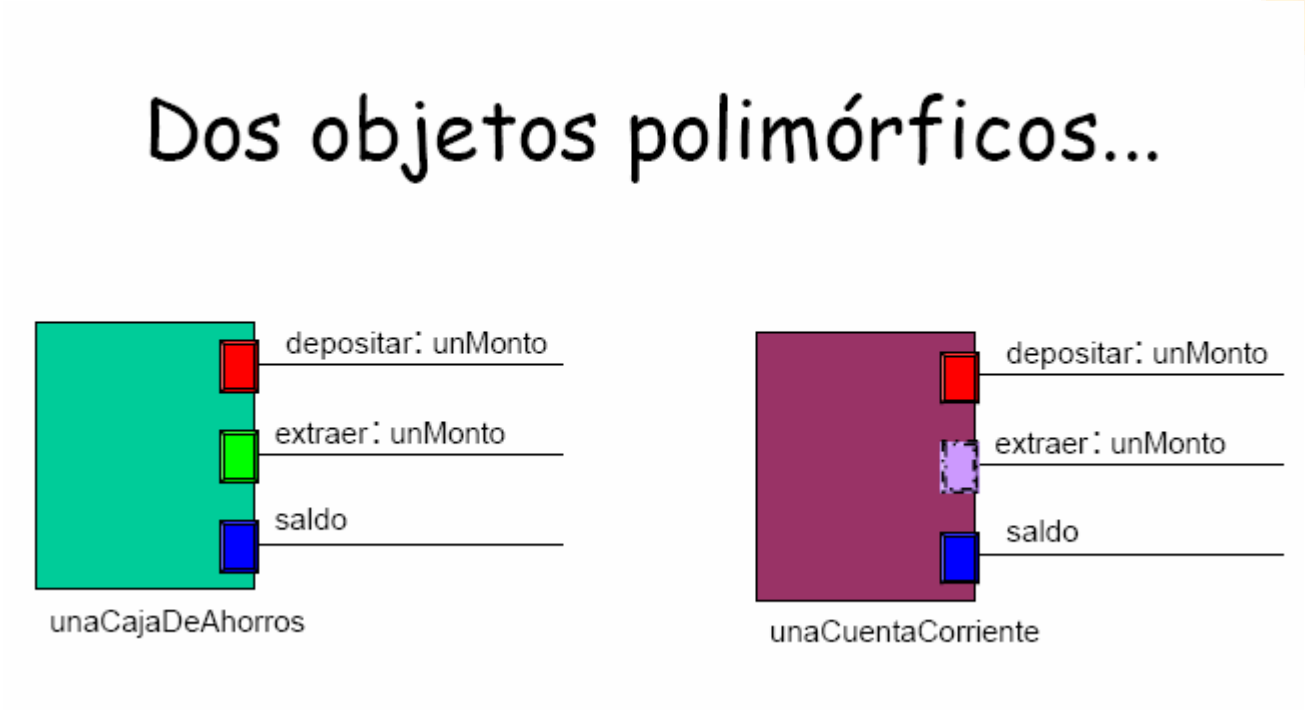
- Los constructores NO se heredan de las superclases. Si se puede invocar del constructor de la subclase a la superclase con super(). Debe ser primera sentencia del cuerpo del constructor.

```
public class CajaDeAhorro extends
   CuentaBancaria {

   public CajaDeAhorro() {
       super();
   }
}
```

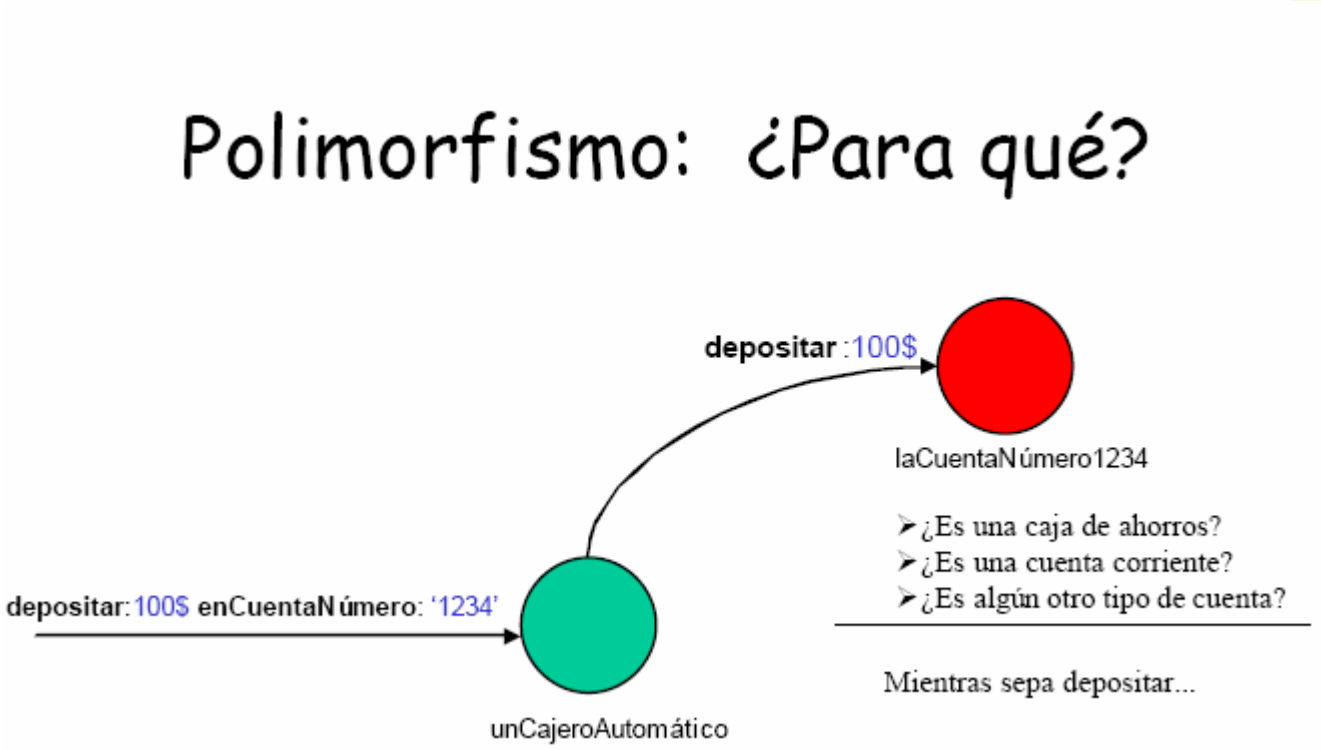
Herencia y Polimorfismo - Polimorfismo

- Polimorfismo:
  - Dos o más objetos son polimórficos respecto de un conjunto de mensajes, si todos pueden responder esos mensajes, aún cuando cada uno lo haga de un modo diferente.
  - Podemos decir que objetos polimórficos corresponden a un mismo “tipo” de objeto.
  - Mismo “tipo” significa mismo comportamiento esencial, independientemente de implementación.
  - Permite que clases de distintos tipos puedan ser referenciadas por una misma variable
    - CuentaBancaria cta;
    - cta = new CajaDeAhorro();
    - cta = new CuentaCorriente();

Herencia y Polimorfismo - Polimorfismo



Herencia y Polimorfismo - Polimorfismo



Herencia y Polimorfismo - Polimorfismo

- El Polimorfismo es la clave de un buen diseño
  - “Código” genérico
  - Objetos desacoplados
  - Objetos intercambiables
  - Objetos reusables
  - Programar por protocolo, no por implementación (buscar la esencia)

Herencia y Polimorfismo - Enlaces

- Binding
  - Binding Temprano o Estático
    - Operandos y operadores son ligados en tiempo de compilación.
    - Es el programador quién decide qué se debe ejecutar.
  - Binding Tardío o Dinámico
    - Operandos y operadores son ligados en tiempo de ejecución.
    - Es el objeto receptor del mensaje quién decide qué se debe ejecutar.
    - La Máquina Virtual tiene que llamar a la versión del método que sea la adecuada para el tipo real del objeto al que se refiere.

Herencia y Polimorfismo – Clase Object

- Clase Object
  - Todas las clases de Java extienden a Object.
  - Es la clase raíz de todo el árbol de la jerarquía de clases de Java.
  - Se puede utilizar una variable de tipo Object como referencia de objetos de cualquier tipo.
  - Métodos de Object
    - **public boolean equals(Object obj)**
      - Define que significa que dos objetos sean iguales.
      - La implementación en **Object** retorna **true** si y solo si se trata del mismo objeto (equivale a una comparación usando ==)
    - **public String toString()**
      - Utilizado para obtener una representación textual conveniente.
      - Los wrappers de tipos primitivos (Integer, Character, Boolean) lo sobrescriben.

Herencia y Polimorfismo - Interface

- Interface
  - Es una forma de describir lo que deberían hacer las clases sin especificar como deben hacerlo (protocolo de comportamiento)
  - Es una colección de declaraciones de constantes y definiciones de métodos sin implementación, agrupados bajo un nombre.
  - La razón de usar interfaces es que Java tiene comprobación estricta de tipos. Cuando se hace una llamada a un método, el compilador necesita ser capaz de averiguar si el método existe realmente.
  - Puede extender múltiples interfaces. Por lo tanto, se tiene herencia múltiple de interfaces.
  - Una clase que implementa una interfaz debe implementar cada uno de los métodos que están definidos en ésta. Una clase puede implementar una o mas interfaces.

Herencia y Polimorfismo - Interface

- Forma general de declaración:

```
package nombrePaquete;
{importaciones}
[public] interface NombreInterfaz [extends
  SuperInterfaces] {
  [Constantes]
  [Encabezados de métodos]
}
```

  - SuperInterfaces es una lista de nombres de interfaces separados por coma.
  - Una interfaz hereda todas las constantes y métodos de sus SuperInterfaces.

Herencia y Polimorfismo - Interface

```
public interface Comparable<T> {
   int compareTo(T otro);
}
```

```
public class CuentaBancaria implements
   Comparable<CuentaBancaria> {
   public int compareTo(CuentaBancaria otro){
   if (saldo < otro.saldo) return -1;
   if (saldo > otro.saldo) return 1;
   return 0;
}
```

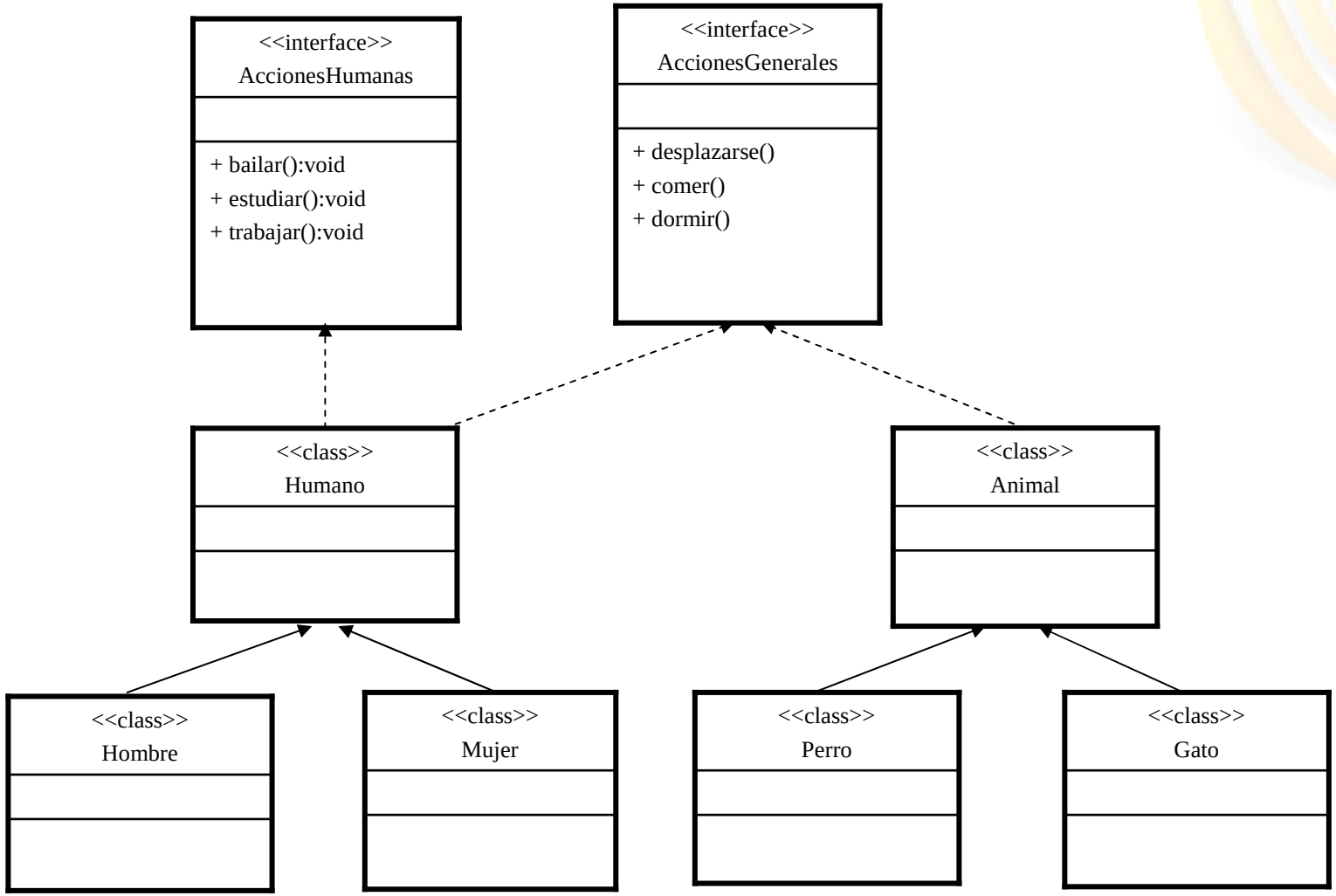
```
Arrays.sort(Object []);
Algoritmo de ordenamiento por fusión, ordena invocando
   a compareTo.
```

Comparable<Object> pivot = (Comparable) a[start];

...

if (pivot.compareTo(a[mid]) < 0)

Herencia y Polimorfismo - Interface



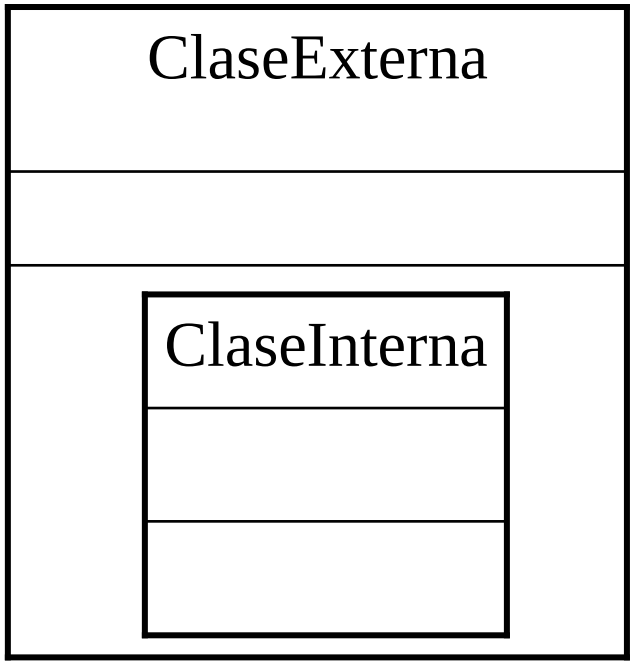
Herencia y Polimorfismo - Interface

- Interfaces, propiedades:
  - No son clases.
  - Se pueden declarar variables de tipo interfaz.
    - Ejemplo: Comparable comparable;
  - Las variables de tipo interfaz tienen que referirse a un objeto de tipo de una clase que implemente la interfaz.
    - Ejemplo: comparable = new Integer(4);
  - Se puede usar instanceof para comprobar si un objeto implementa o no la interfaz.
    - Ejemplo: “Ejemplo” instanceof Comparable;
  - Se pueden extender las interfaces.
  - No se pueden poner atributos ni métodos estáticos, si es posible aportar constantes en ellas.
    - Los atributos son public static final automáticamente.
  - Las clases pueden implementar múltiples interfaces.
  - No debe crecer, si se cambia el comportamiento de una interfaz, todas las clases que la implementen fallaran.

Herencia y Polimorfismo - Clases Internas

- Clases internas (Inner classes):
  - Definición: una clase anidada es una clase que se encuentra dentro de otra.

```
public class UnaClase {
   . . .
   class UnaClaseAnidada
   {
       . . .
   }
}
```



Herencia y Polimorfismo - Clases Internas

- Razones de porque podrían ser necesario hacer esto:
  - Los métodos de las clases internas pueden acceder a los datos que se encuentren en el mismo ámbito en que hayan sido definidos.
  - Se pueden ocultar de otras clases del mismo paquete.
  - Las clases internas anónimas son útiles cuando se intentan definir retrollamadas sin escribir mucho código.
  - Las clases internas son un fenómeno del compilador, no de la máquina virtual (no posee conocimiento respecto a ellas)
  - Las clases se traducen a archivos
    - UnaClase$UnaClaseAnidada.class

Herencia y Polimorfismo - Clases Internas

- Clases internas, características:
  - Solo pueden instanciarse dentro de su clase contenedora.
  - Se instancian de la misma forma que una clase común.
  - Puede haber múltiples niveles de clases anidadas.
  - Las clases anidadas pueden tener las mismas características que las clases comunes, salvo que no pueden declarar ni métodos ni variables **static**.
  - Pueden acceder a todas las variables y métodos de la clase contenedora, incluso los **private**.
  - La clase contenedora puede acceder a todas las variables y métodos de la clase anidada, incluso los **private**.
  - Se utilizan principalmente para estructurar mejor el código dentro de una clase.

Herencia y Polimorfismo - Clases Internas

- Clases internas, clases anónimas:
  - Son clases que se declaran en el código para ser utilizadas en un solo lugar.
  - Sintaxis críptica.
  - Deben ser simples, sino utilizar clases anidadas comunes.
  - No tienen modificadores de acceso ni constructores.

```
JButton jButton = new JButton(new
  AbstractAction() {
  public void actionPerformed(ActionEvent e) {
      ...
  }
  });
```

Encapsulamiento, Herencia y Polimorfismo

- **Ejercicio 1:** Para cada uno de los siguientes TDA, defina en Java la clase correspondiente con los métodos asociados. Cada clase debe tener un método miembro main, donde debe implementarse el test para cada método.
- **Complejo:**
  - Implementar los métodos que permitan realizar las siguientes operaciones entre objetos de la clase: sumar, restar, multiplicar, comparar (equals), obtener el modulo (modulo), toString().
  - Sobrecargue las funciones de manera que se pueda trabajar con objetos no complejos, por ejemplo, sumar un complejo con un número entero.
  - Observe que z * ž da un número Real, llamado módulo.
  - Crear una clase OrdenadoraDeComplejos que ordene los números complejos según los siguientes criterios: por su módulo, por la parte real, por la parte imaginaria.
- **VectorMath:**
  - Implementar los métodos que permitan realizar las siguientes operaciones entre objetos de la clase:
  - suma de vectores
  - resta de vectores
  - producto de un vectorMath por otro vectorMath
  - producto de un vectorMath por una MatrizMath
  - producto de un vectorMath por un Real
  - normaUno(); normaDos; normaInfinito()
  - equals().
- **MatrizMath:**
  - Implementar los métodos que permitan realizar las siguientes operaciones entre objetos de la clase:
  - suma de matrices
  - resta de matrices
  - producto de matrices
  - producto de un matriz por un vector
  - producto de un matriz por una constante tipo float
  - matrizInversa()
  - determinante()
  - normaUno(), normaDos(); normaInfinito();
  - equals().
- **SEL (Sistemas de ecuaciones lineales)**
  - constructores
  - static bool test()
  - resolver ()
  - mostrarResultado()

Encapsulamiento, Herencia y Polimorfismo

- **Ejercicio 2:** Modele una Empresa con empleados. Una empresa conoce a todos sus empleados. Los empleados pueden ser de planta permanente o temporaria, además hay gerentes, que también son empleados de planta permanente, pero siguen un régimen salarial particular.
- Cuando un empleado es de planta permanente cobra la cantidad de horas trabajadas por $30, más antigüedad ($10 por año de antigüedad), más salario familiar. Cuando es de planta temporaria, no cobra antigüedad y cobra la cantidad de horas trabajadas por $20, más salario familiar. El salario familiar es $20 por cada hijo, los empleados casados además cobran $10 por su esposa/o.
- Un gerente cobra de manera similar a un empleado de planta permanente pero su hora trabajada vale $40, por antigüedad se le pagan $15 por año, mientras que el salario familiar es el mismo que el de los empleados de planta permanente y temporal.
- Defina e implemente el mensaje montoTotal() en la clase Empresa, que retorna el monto total que la empresa debe pagar en concepto de sueldos a sus empleados.
- Realice la implementación del sistema completo en Java.
- Provea una clase TestEmpresa para instanciar y testear su sistema. En el método main de esa clase cree el siguiente escenario y envíe a la empresa el mensaje montoTotal() para obtener la liquidación total.
  - Una empresa, con el CUIT y Razón Social que desee, y con los cuatro empleados que se describen a continuación.
  - Un empleado de Planta Temporaria con 80 horas trabajadas, con esposa y sin hijos.
  - Un empleado de Planta Permanente (que no sea gerente) con 80 horas trabajadas, con esposa, 2 hijos y 6 años de antigüedad.
  - Un empleado de Planta Permanente (que no sea gerente) con 160 horas trabajadas, sin esposa, sin hijos y con 4 años de antigüedad.
  - Un Gerente con 160 horas trabajadas, con esposa, un hijo y 10 años de antigüedad

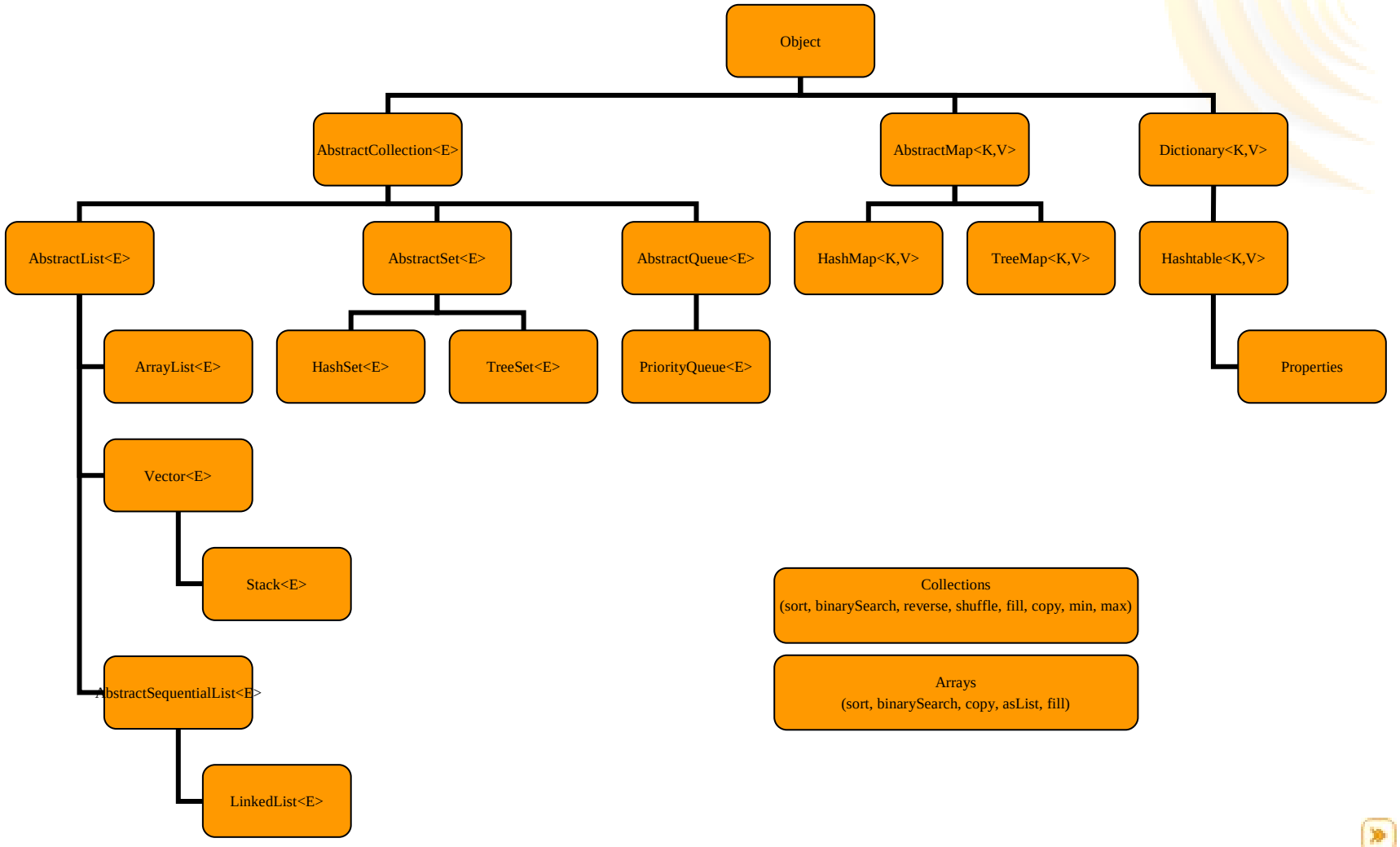
Encapsulamiento, Herencia y Polimorfismo

- **Ejercicio 3:** Un negocio se dedica a la reparación de PCs. Cada PC que recoge contiene información sobre el código de la reparación, el código del cliente, la descripción de la avería y el precio de la reparación. Además contiene un campo entero que recoge el estado de la reparación con la siguiente información:
  - Cuando llega una PC al taller se queda pendiente de aprobar el presupuesto y el estado es 0
  - Cuando el cliente acepta el presupuesto el campo estado se pone a 1
  - Cuando el ordenador esta reparado el campo estado se pone a 2
- Algunas PCs están en garantía, en cuyo caso se añadirá un campo con la fecha de compra (AAAAMMDD)
  - a) Codifique la clase PC. Tendrá un constructor que permita inicializar el código de la reparación, el código del cliente y la descripción de la avería. También tendrá un método que permita convertir a cadena una representación de la PC.
  - b) Codifique una subclase para las pcs en garantía. En esta clase, el método que convierte la PC en cadena añadirá también la fecha de la compra.
  - c) Codifique una clase Taller. El constructor de dicha clase permitirá la creación de una colección de donde se almacenarán las PCs que están en el taller; dicho constructor tomará como argumento el número de PCs máximo que puede tener el taller. La clase Taller contendrá métodos para:
- La entrada de una PC en el taller.
- La aceptación del presupuesto de una reparación por el cliente. El método deberá buscar la PC y poner el estado de la reparación a 1.
- La comunicación del final de la reparación de la PC. Este método recibirá el código de la reparación y pondrá el estado a 2. En el caso de ordenadores que no estén en garantía recibirá también el precio y actualizará el campo correspondiente.
- La entrega de una PC, dejando el elemento que ocupaba a nulo.

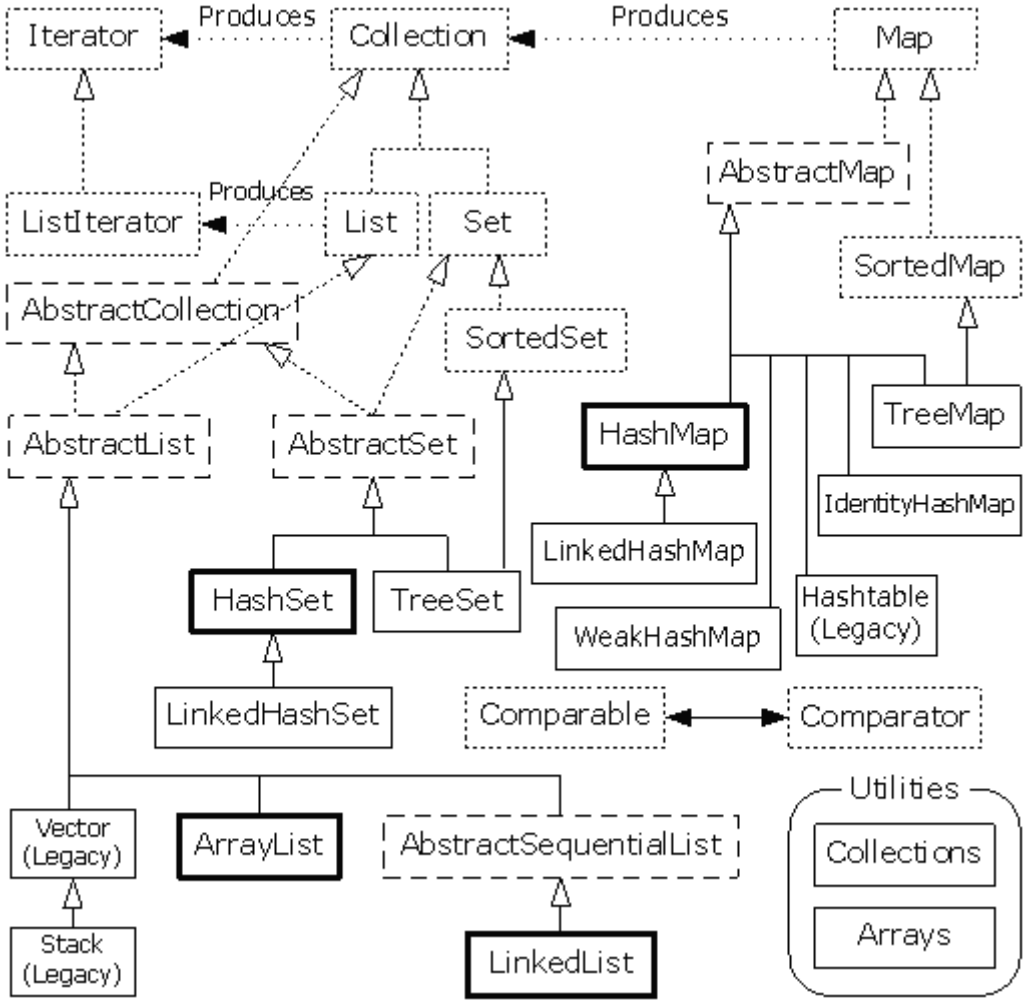
Encapsulamiento, Herencia y Polimorfismo

- **Ejercicio 4:** Modele una Empresa de Correo con envíos a diferentes lugares del país. Una empresa conoce todos sus envíos. Los envíos pueden ser cartas o encomiendas, además hay telegramas, que también son cartas, pero siguen un cálculo de precio particular. Un envío implementa la interfaz Calculable, la misma describe la funcionalidad precio que devuelve un double.
- Cuando un envío es una encomienda el precio se calcula sobre el peso de la misma, siendo la cantidad de gramos de peso por $0.01 más precio por distancia. Cuando es una carta solo se cobra por distancia. El precio por distancia es de $1 si no supera los 100 km., $1.5 si la distancia está entre los 100 km. y los 500 km. y $2 si supera los 500 Km.
- Un telegrama se cobra de manera similar a una carta pero se le adiciona $0.01 por carácter (no se incluyen los espacios en blanco).
- Defina e implemente el mensaje liquidaciónTotal() en la clase Correo, que retorna el monto total que la empresa debe cobrar en concepto de precios de envío.
- Realice la implementación del sistema completo en Java.
- Provea una clase TestCorreo para instanciar y testear su sistema. En el método main de esa clase cree el siguiente escenario y envíe a la empresa de correo el mensaje liquidaciónTotal() para obtener la liquidación total.
  - Una empresa, con el CUIT y Razón Social que desee, y con los cuatro envíos que se describen a continuación.
  - Una encomienda con 120 km. de recorrido y con 3.5 kg. de peso
  - Una carta con 650 km. de recorrido.
  - Una carta con 5 km. de recorrido.
  - Un telegrama con 405 Km. de recorrido y un texto “Te deseo hoy y siempre felicidad completa.”
- Las clases se organizan en el paquete, cuyo nombre es su apellido.

Colecciones - Jerarquía



Colecciones - Jerarquía



Colecciones - ArrayList

- ArrayList
  - java.util.ArrayList<T>
  - Ajusta automáticamente su capacidad a medida que se ingresan y eliminan elementos.
  - Es una clase genérica que admite un tipo como parámetro.
  - Mas eficiente que la clase Vector.
  - Construtores:
    - ArrayList<T>()
    - ArrayList<T>(int capacidadInicial)

Colecciones - ArrayList

- ArrayList
  - Métodos:
    - size(): int //número de elementos almacenados actualmente.
    - ensureCapacity(int capacidad): void
    - trimToSize(): void //reduce capacidad de almacenamiento a su tamaño actual
    - add(T obj): boolean
    - add(int indice, T obj): void //desplaza hacia adelante
    - set(int indice, T obj): void // sobreescribe
    - get(int indice): T
    - remove(int indice): T
    - remove(T obj): boolean

Colecciones - Stack

- Stack
  - java.util.Stack<E>
  - Extiende a Vector.
  - Lógica LIFO
  - Constructores:
    - Stack()
  - Métodos:
    - push(E obj): E
    - pop(): E
    - peek(): E
    - empty(): boolean

Colecciones - Varias

Usos:

ArrayList<E>

```
ArrayList<Integer> vector = new ArrayList<Integer>();
System.out.println("Esta vacio?: " + vector.isEmpty());
vector.add(2);
vector.add(5);
vector.add(3);
System.out.println("toString: " + vector);
vector.remove(2);
System.out.println("toString: " + vector);
System.out.println("Esta vacio?: " + vector.isEmpty());
System.out.println("Posición del elemento 5: " + vector.indexOf(5));
System.out.println("Tamaño del vector: " + vector.size());
```

Stack<E>

```
Stack<Integer> pila = new Stack<Integer>();
System.out.println("Esta vacia?: " + pila.empty());
pila.push(2);
pila.push(5);
pila.push(3);
System.out.println("toString: " + pila);
pila.pop();
System.out.println("toString: " + pila);
System.out.println("Esta vacio?: " + pila.empty());
System.out.println("Elemento en el tope: " + pila.peek());
```

PriorityQueue<E>

```
Queue<Integer> cola = new PriorityQueue<Integer>();
cola.add(2);
cola.add(5);
cola.add(3);
System.out.println("toString: " + cola);
cola.poll();
System.out.println("toString: " + cola);
System.out.println("Elemento en a salir: " + cola.peek());
```

LinkedList<E>

```
List<Integer> lista = new LinkedList<Integer>();
System.out.println("Esta vacia?: " + lista.isEmpty());
lista.add(2);
lista.add(1, 5);
lista.add(3);
System.out.println("toString: " + lista);
lista.remove(1);
System.out.println("toString: " + lista);
System.out.println("Esta vacia?: " + lista.isEmpty());
System.out.println("Elemento en pos 1?: " + lista.get(1));
System.out.println("Tamaño de la lista: " + lista.size());
```

Colecciones - Mapas

- Mapa
  - Es una estructura de Java que permite almacenar pares clave-valor.
  - Mapas importantes:
    - Properties:
      - Extiende a Hashtable (ordena por valor y no por clave)
      - Útil para almacenar y recuperar archivos de propiedades (opciones de configuración para programas)
    - HashMap
      - Extiende a AbstractMap
      - Implementación basada en una tabla hash

Colecciones - Mapas

Properties

```
Properties prop = new Properties();
prop.put(“user”, “ppando”);
prop.get(“user”);
prop.load(new FileInputStream(new
   File("propiedades/prop.properties")));
```

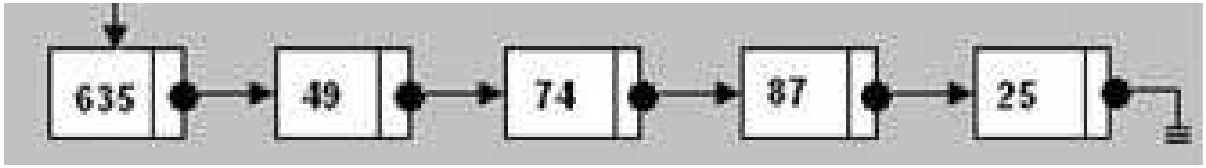
HashMap

```
HashMap<String, Object> map = new HashMap<String,
   Object>();
map.put("user", "ppando");
map.get("user");
```

Colecciones - Dato Recursivo

- Dato recursivo:
  - Si el nombre del tipo aparece en su propia definición (una o más veces)
  - Ejemplo:

```
public class Nodo<T> {
    private T elemento;
    private Nodo<T> sig;
}
```

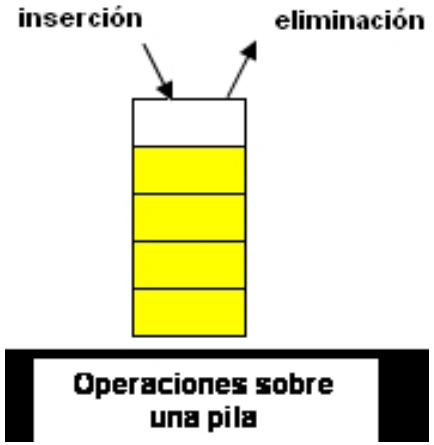


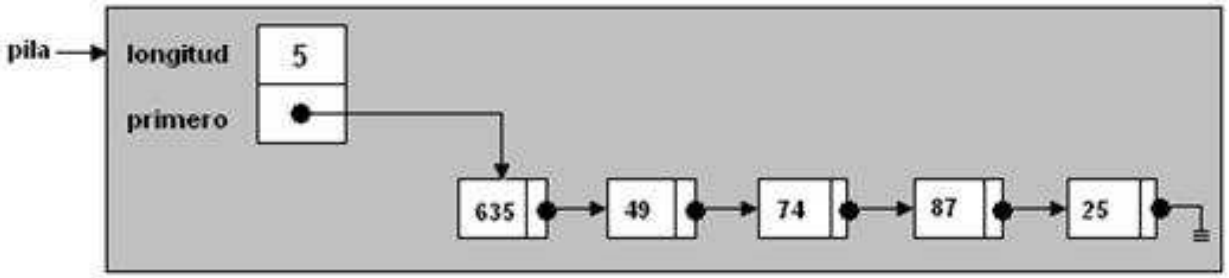
Colecciones - Pila

- Pila:
  - Una pila es una estructura de datos en la cual sus elementos sólo pueden ser sacados y agregados por uno de sus extremos. Este tipos de estructuras suele llamarse LIFO (last-in-first-out) ya que el último elemento en llegar es el primero en salir.
  - En todo momento el único elemento visible es el último que se insertó. La parte de la pila donde éste se encuentra se denomina tope.
  - Las pilas son utilizadas en el reconocimiento de lenguajes, en la evaluación de expresiones, en la implementación de recursividad y en el recorrido de árboles.

Colecciones - Pila

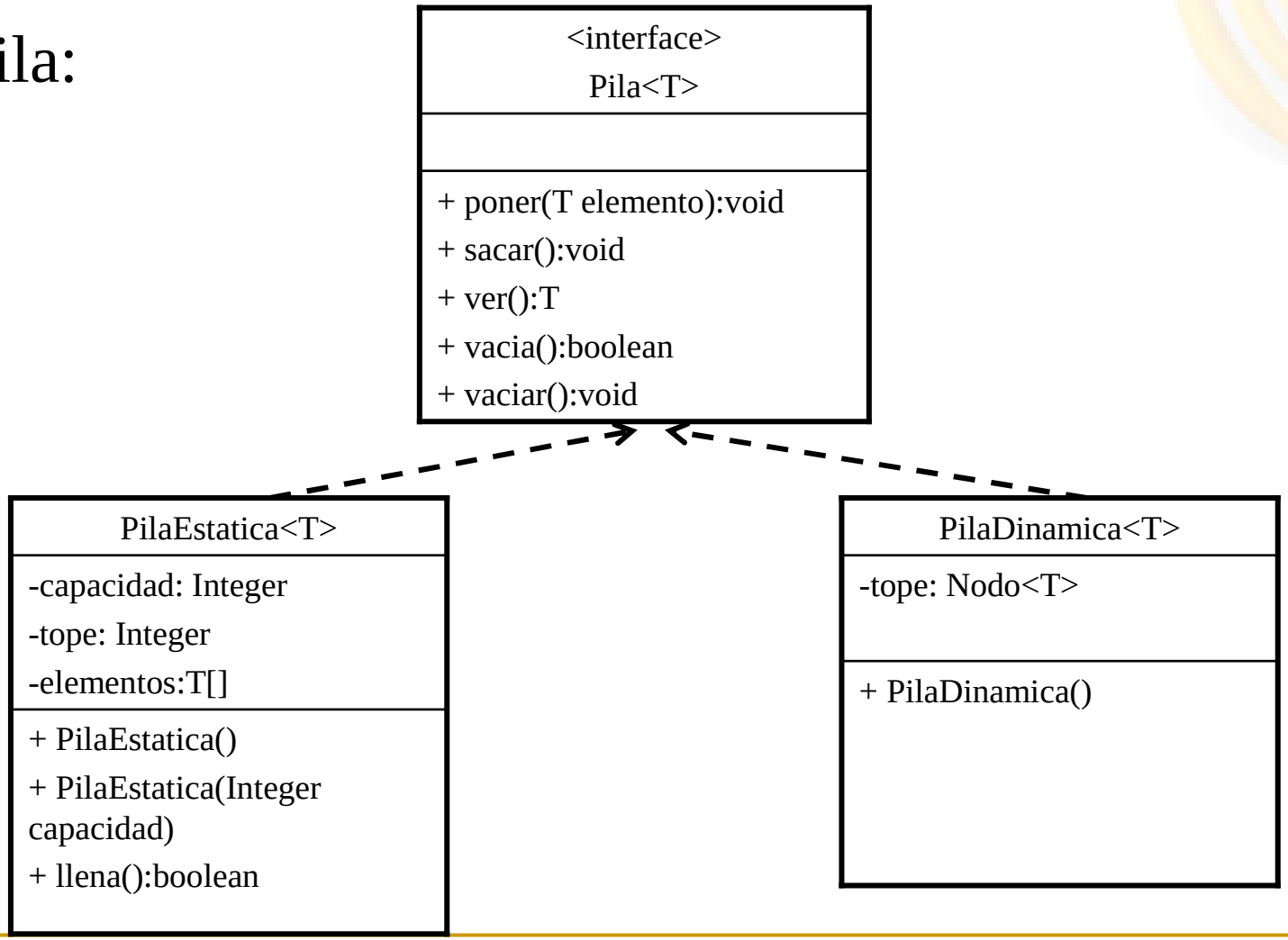
- Pila:





Colecciones - Pila

- Pila:



Colecciones - Pila

Pila estática:

```
public class PilaEstatica<T> implements Pila<T> {
	private int capacidad;
	private int tope;
	private T[] elementos;
	public PilaEstatica() {
		this(10);
	}
	public PilaEstatica(int capacidad) {
		if (capacidad <= 0)
			throw new IllegalArgumentException("Capacidad ilegal: " + capacidad);
		this.setCapacidad(capacidad);
		this.setTope(-1);
		this.setElementos((T[]) new Object[capacidad]);
	}
	private int getCapacidad() {
		return this.capacidad;
	}
	private void setCapacidad(int capacidad) {
		this.capacidad = capacidad;
	}
	private void setTope(int tope) {
		this.tope = tope;
	}
```

Colecciones - Pila

Pila estática:

```
public class PilaEstatica<T> implements Pila<T> {

	private int getTope() {
		return this.tope;
	}
	private T[] getElementos() {
		return this.elementos;
	}
	private void setElementos(T[] elementos) {
		this.elementos = elementos;
	}
	public boolean vacia() {
		return (this.getTope() == -1);
	}
	public boolean llena() {
		return this.getTope() == (this.getCapacidad() - 1);
	}
	public void poner(T elemento) throws PilaLlenaException {
		if (this.llena())
				throw new PilaLlenaException();
		this.setTope(this.getTope() + 1);
		this.getElementos()[this.getTope()] = elemento;
	}
```

Colecciones - Pila

Pila estática:

```
public class PilaEstatica<T> implements Pila<T> {

	public void sacar() throws PilaVaciaException {
		if (this.vacia())
			throw new PilaVaciaException();
		this.setTope(this.getTope() - 1);
	}
	public T ver() throws PilaVaciaException {
		if (this.vacia())
			throw new PilaVaciaException();
		return this.getElementos()[this.getTope()];
	}
	public void vaciar() {
		this.setTope(-1);
	}

}
```

Colecciones - Pila

Pila dinámica:

```
public class PilaDinamica<T> implements Pila<T> {
	class Nodo<T> {
		private T elemento;
		private Nodo<T> siguiente;
		public Nodo() {
			this(null, null);
		}
		public Nodo(T elemento, Nodo<T> siguiente) {
			this.setElemento(elemento);
			this.setSiguiente(siguiente);
		}
		public Nodo(T elemento) {
			this(elemento, null);
		}
		public T getElemento() {
			return this.elemento;
		}
		public void setElemento(T elemento) {
			this.elemento = elemento;
		}
		public Nodo<T> getSiguiente() {
			return this.siguiente;
		}
```

Colecciones - Pila

Pila dinámica:

```
public class PilaDinamica<T> implements Pila<T> {

                public void setSiguiente(Nodo<T> siguiente) {
                                this.siguiente = siguiente;
                }

        }

        private Nodo<T> tope;
        public PilaDinamica() {
                this.setTope(null);
        }
        private Nodo<T> getTope() {
                return this.tope;
        }
        private void setTope(Nodo<T> tope) {
                this.tope = tope;
        }
        public boolean vacia() {
                return (this.getTope() == null);
        }
        public void poner(T elemento) {
                this.setTope(new Nodo<T>(elemento, this.getTope()));
        }
```

Colecciones - Pila

Pila dinámica:

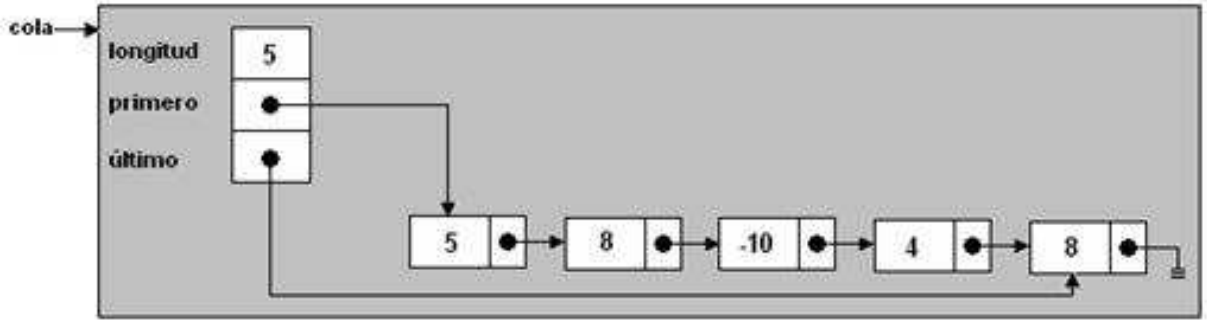
```
public class PilaDinamica<T> implements Pila<T> {

	public void sacar() throws PilaVaciaException {
		if (this.vacia())
			throw new PilaVaciaException();
		this.setTope(this.getTope().getSiguiente());
	}
	public T ver() throws PilaVaciaException {
		if (this.vacia())
			throw new PilaVaciaException();
		return this.getTope().getElemento();
	}
	public void vaciar() {
		while (!this.vacia())
		try {
			this.sacar();
		} catch (PilaVaciaException pilaVaciaException) {
			pilaVaciaException.printStackTrace();
	}

}
```

Colecciones - Cola

- Cola:
  - Una cola es una estructura de datos en la cual sus elementos sólo pueden ser sacados (y consultados) por uno de sus extremos e insertados por el otro. Este tipos de estructuras suele llamarse FIFO (first-in-first-out) por el mismo mecanismo utilizado para la eliminación y adición de elementos.
  - Las colas pueden asociarse con las colas de atención que existen en el mundo real. Los usuarios que llegan primero serán atendidos antes que los últimos en llegar. Cuando un usuario llega debe situarse al final de la cola, lo que implica que debe esperar a que todos los usuarios que se encuentran antes que él sean atendidos. No es posible situarse en un lugar distinto al último al llegar ni esperar ser atendido si se encuentra en un lugar diferente al primero.

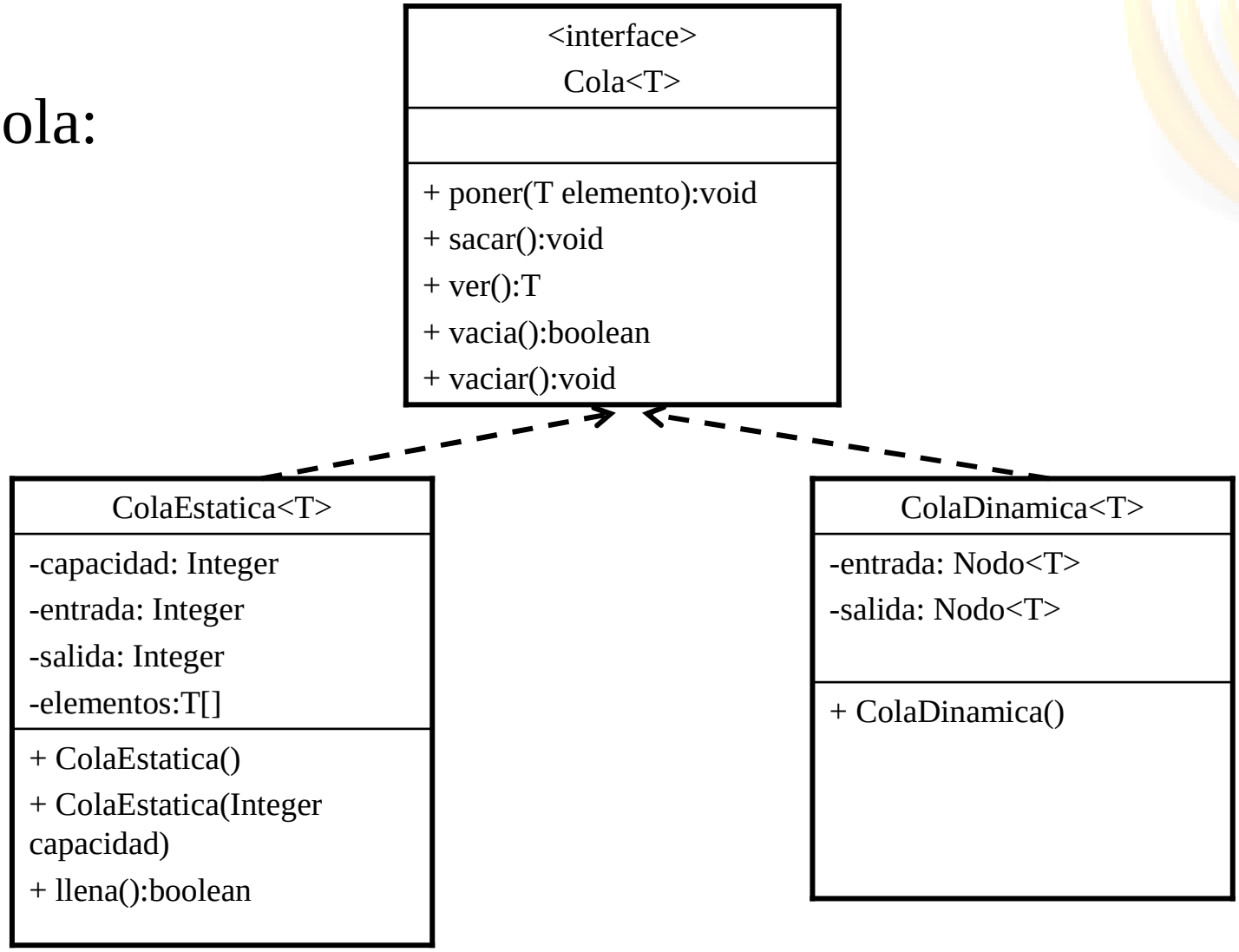
Colecciones - Cola

- Cola:



Colecciones - Cola

- Cola:



Colecciones - Cola

Cola estática:

```
public class ColaEstatica<T> implements Cola<T> {
	private int capacidad;
	private int entrada;
	private int salida;
	private T[] elementos;
	public ColaEstatica() {
		this(10);
	}
	public ColaEstatica(int capacidad) {
		if (capacidad <= 0)
			throw new IllegalArgumentException("Capacidad ilegal: " + capacidad);
		this.setCapacidad(capacidad);
		this.setEntrada(-1);
		this.setSalida(-1);
		this.setElementos((T[]) new Object[capacidad]);
	}
	private int getCapacidad() {
		return this.capacidad;
	}
	private void setCapacidad(int capacidad) {
		this.capacidad = capacidad;
	}
	private void setEntrada(int entrada) {
		this.entrada = entrada;
	}
```

Colecciones - Cola

- Cola estática:

```
public class ColaEstatica<T> implements Cola<T> {

	private int getEntrada() {
		return this.entrada;
	}
	private void setSalida(int salida) {
		this.salida = salida;
	}
	private int getSalida() {
		return this.salida;
	}
	private T[] getElementos() {
		return this.elementos;
	}
	private void setElementos(T[] elementos) {
		this.elementos = elementos;
	}
	public boolean vacia() {
		return (this.getEntrada() == -1);
	}
	public boolean llena() {
		return ((this.getEntrada() == this.getCapacidad() - 1 && this.getSalida() == 0) || (this.getEntrada() + 1 == this.getSalida()));
	}
```

Colecciones - Cola

Cola estática:

```
public class ColaEstatica<T> implements Cola<T> {

	public void poner(T elemento) throws ColaLlenaException {
		if (this.llena())
			throw new ColaLlenaException();
		if (this.getEntrada() == this.getCapacidad() - 1 && this.getSalida() != 0)
			this.setEntrada(0);
		else
			this.setEntrada(this.getEntrada() + 1);
		this.getElementos()[this.getEntrada()] = elemento;
		if (this.getSalida() == -1)
			this.setSalida(0);
	}
	public void sacar() throws ColaVaciaException {
		if (this.vacia())
			throw new ColaVaciaException();
		if (this.getSalida() == this.getEntrada()) {
			this.vaciar();
			return;
		}
		if (this.getSalida() == this.getCapacidad())
			this.setSalida(0);
		else
			this.setSalida(this.getSalida() + 1);
	}
```

Colecciones - Cola

Cola estática:

```
public class ColaEstatica<T> implements Cola<T> {

	public T ver() throws ColaVaciaException {
		if (this.vacia())
			throw new ColaVaciaException();
		return this.getElementos()[this.getSalida()];
	}
	public void vaciar() {
		this.setEntrada(-1);
		this.setSalida(-1);
	}

}
```

Colecciones - Cola

Cola:

```
public class ColaDinamica<T> implements Cola<T> {
	class Nodo<T> {
		private T elemento;
		private Nodo<T> siguiente;
		public Nodo() {
			this(null, null);
		}
		public Nodo(T elemento, Nodo<T> siguiente) {
			this.setElemento(elemento);
			this.setSiguiente(siguiente);
		}
		public Nodo(T elemento) {
			this(elemento, null);
		}
		public T getElemento() {
			return this.elemento;
		}
		public void setElemento(T elemento) {
			this.elemento = elemento;
		}
		public Nodo<T> getSiguiente() {
			return this.siguiente;
		}
```

Colecciones - Cola

Cola:

```
public class ColaDinamica<T> implements Cola<T> {
        public void setSiguiente(Nodo<T> siguiente) {
                this.siguiente = siguiente;
        }
}
private Nodo<T> entrada;
private Nodo<T> salida;
public ColaDinamica() {
        this.setEntrada(null);
        this.setSalida(null);
}
private Nodo<T> getEntrada() {
        return this.entrada;
}
private void setEntrada(Nodo<T> entrada) {
        this.entrada = entrada;
}
private Nodo<T> getSalida() {
        return this.salida;
}
private void setSalida(Nodo<T> salida) {
        this.salida = salida;
}
```

Colecciones - Cola

Cola:

```
public class ColaDinamica<T> implements Cola<T> {

	public boolean vacia() {
		return (this.getEntrada() == null);
	}
	public void poner(T elemento) {
		Nodo<T> nodoNuevo = new Nodo<T>(elemento, null);
		if (this.getEntrada() == null)
			this.setSalida(nodoNuevo);
		else
			this.getEntrada().setSiguiente(nodoNuevo);
		this.setEntrada(nodoNuevo);
	}
	public void sacar() throws ColaVaciaException {
		if (this.vacia())
			throw new ColaVaciaException();
		this.setSalida(this.getSalida().getSiguiente());
		if (this.getSalida() == null)
			this.setEntrada(null);
	}
```

Colecciones - Cola

Cola:

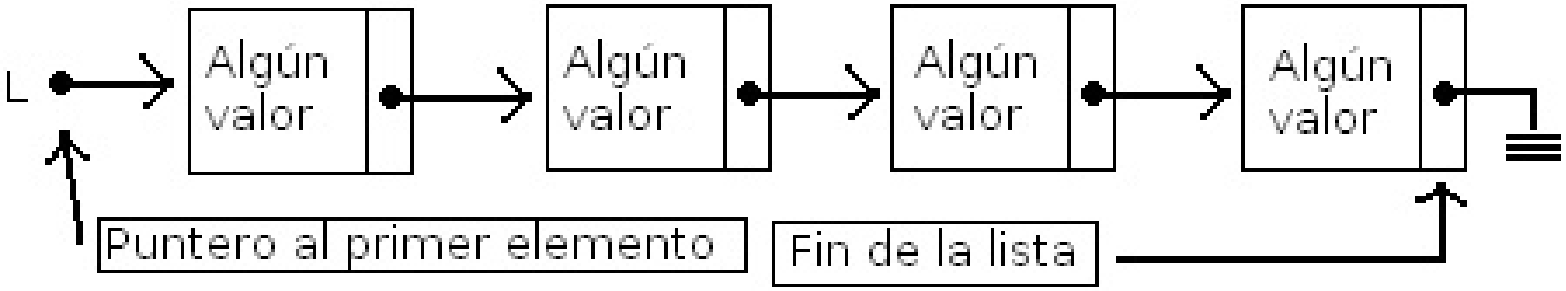
```
public class ColaDinamica<T> implements Cola<T> {

	public T ver() throws ColaVaciaException {
		if (this.vacia())
			throw new ColaVaciaException();
		return this.getSalida().getElemento();
	}
	public void vaciar() {
		while (!this.vacia())
		try {
			this.sacar();
		} catch (ColaVaciaException colaVaciaException) {
			colaVaciaException.printStackTrace();
		}
	}

}
```

Colecciones - Lista Simplemente Enlazada

- Lista Simplemente Enlazada:
  - Una lista es una estructura lineal compuesta por una secuencia de 0 o más elementos.
  - La forma más simple de estructura dinámica es la lista abierta. En esta forma, los nodos se organizan de modo que cada uno apunta al siguiente, y el último no apunta a nada, es decir, el puntero del nodo siguiente vale NULL.
  - En una lista cada elemento posee una posición que corresponde al lugar que ocupa dentro de la secuencia de elementos. El sucesor de un elemento se refiere al elemento que se encuentra en la siguiente posición. Del mismo modo, el antecesor de un elemento es el elemento que se encuentra en la posición anterior.
  - La complejidad de la búsqueda en el peor caso en este tipo de estructura es O(n), donde n es la longitud (el número de elementos) de la lista.

Colecciones - Lista Simplemente Enlazada

- Lista Simplemente Enlazada:



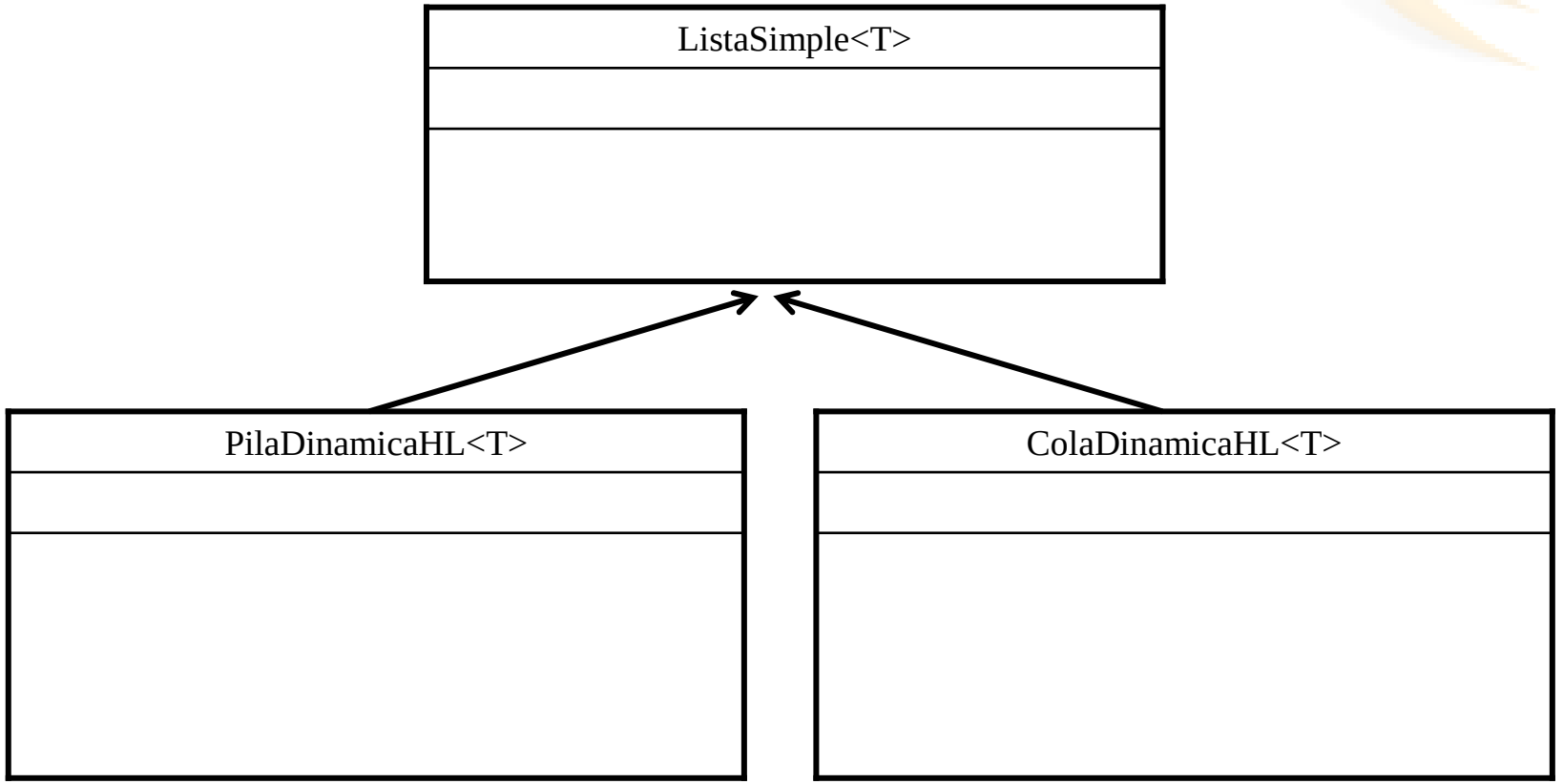
Colecciones - Lista Simplemente Enlazada

- Lista SE:

ListaSimple<T>
-entrada: Nodo<T> -cantidadElementos:Integer
+ ListaSimple() + insertarAtras(T elemento):void + insertarAdelante(T elemento):void + eliminarAtras():void + eliminarAdelante():void + eliminar(T elemento):void + reversa():void + insertar(Integer posicion, T elemento):void + eliminar(Integer):void + vacia():boolean + buscar(T elemento):boolean + buscar(Integer posicion):T + getPosicion(T elemento):Integer + vaciar():void

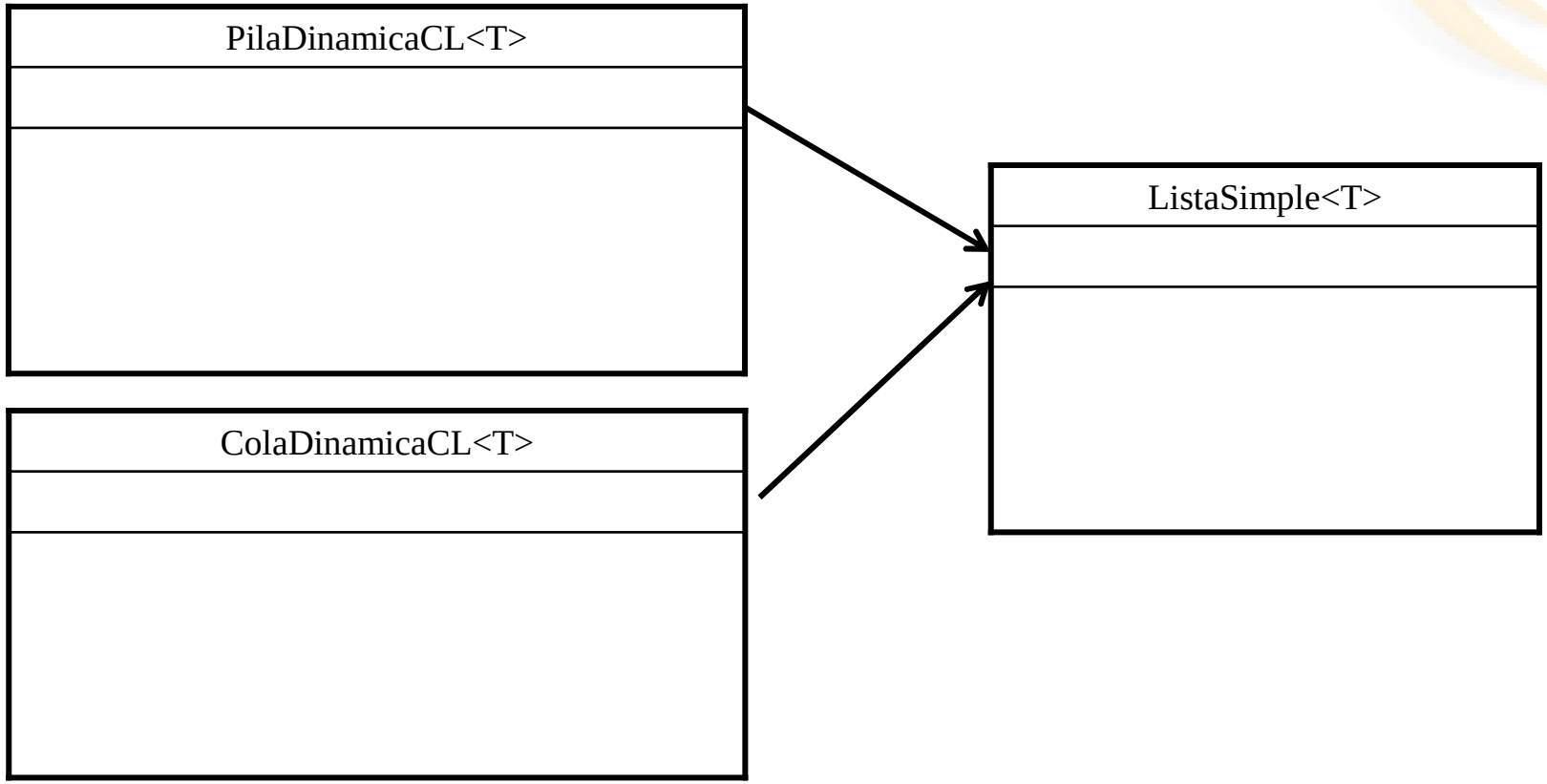
Colecciones - Lista Simplemente Enlazada

- Diseño 1: HL (herencia lista)



Colecciones - Lista Simplemente Enlazada

- Diseño 2: CL (contiene lista)



Colecciones - Lista Simplemente Enlazada

Lista SE:

```
public class ListaSimple<T> {
	class Nodo<T> {
		private T elemento;
		private Nodo<T> siguiente;
		public Nodo() {
			this(null, null);
		}
		public Nodo(T elemento, Nodo<T> siguiente) {
			this.setElemento(elemento);
			this.setSiguiente(siguiente);
		}
		public Nodo(T elemento) {
			this(elemento, null);
		}
		public T getElemento() {
			return this.elemento;
		}
		public void setElemento(T elemento) {
			this.elemento = elemento;
		}
		public Nodo<T> getSiguiente() {
			return this.siguiente;
		}
```

Colecciones - Lista Simplemente Enlazada

Lista SE:

```
public class ListaSimple<T> {
        public void setSiguiente(Nodo<T> siguiente) {
                this.siguiente = siguiente;
        }
    }
    private Nodo<T> entrada;
    private Integer cantidadElementos;
    public ListaSimple() {
        this.setEntrada(null);
        this.setCantidadElementos(0);
    }
    protected Nodo<T> getEntrada() {
        return this.entrada;
    }
    private void setEntrada(Nodo<T> entrada) {
        this.entrada = entrada;
    }
    public Integer getCantidadElementos() {
        return this.cantidadElementos;
    }
    protected void setCantidadElementos(Integer cantidadElementos) {
        this.cantidadElementos = cantidadElementos;
    }
```

Colecciones - Lista Simplemente Enlazada

Lista SE:

```
public class ListaSimple<T> {

	public T verEntrada() {
		return this.getEntrada().getElemento();
	}
	public void insertarAtras(T elemento) {
		if (elemento == null)
			throw new IllegalArgumentException();
		Nodo<T> nodoActual = this.getEntrada();
		Nodo<T> nodoAnterior = null;
		while (nodoActual != null) {
			nodoAnterior = nodoActual;
			nodoActual = nodoActual.getSiguiente();
		}
		if (nodoAnterior == null)
			this.insertarAdelante(elemento);
		else {
			nodoAnterior.setSiguiente(new Nodo<T>(elemento, null));
			this.setCantidadElementos(this.getCantidadElementos() + 1);
		}
	}
```

Colecciones - Lista Simplemente Enlazada

Lista SE:

```
public class ListaSimple<T> {
	public void insertarAdelante(T elemento) {
		if (elemento == null)
			throw new IllegalArgumentException();
		this.setEntrada(new Nodo<T>(elemento, this.getEntrada()));
		this.setCantidadElementos(this.getCantidadElementos() + 1);
	}
	public void eliminarAtras() {
		Nodo<T> nodoActual = this.getEntrada();
		Nodo<T> nodoAnterior = null;
		while (nodoActual != null) {
			nodoAnterior = nodoActual;
			nodoActual = nodoActual.getSiguiente();
		}
		if (nodoAnterior == null)
			this.setEntrada(null);
		else
			nodoAnterior.setSiguiente(null);
		this.setCantidadElementos(this.getCantidadElementos() - 1);
	}
```

Colecciones - Lista Simplemente Enlazada

Lista SE:

```
public class ListaSimple<T> {
	public void eliminarAdelante() {
		if (!this.vacia()) {
			this.setEntrada(this.getEntrada().getSiguiente());
			this.setCantidadElementos(this.getCantidadElementos() - 1);
		}
	}
	public void eliminar(T elemento) {
		if (elemento == null)
			throw new IllegalArgumentException();
		Nodo<T> nodoActual = this.getEntrada();
		Nodo<T> nodoAnterior = null;
		while ((nodoActual != null) && (!nodoActual.getElemento().equals(elemento))) {
			nodoAnterior = nodoActual;
			nodoActual = nodoActual.getSiguiente();
		}
		if (nodoActual != null)
			if (nodoAnterior == null)
				this.eliminarAdelante();
			else {
				nodoAnterior.setSiguiente(nodoActual.getSiguiente());
				this.setCantidadElementos(this.getCantidadElementos() - 1);
			}
	}
```

Colecciones - Lista Simplemente Enlazada

Lista SE:

```
public class ListaSimple<T> {

	public void reversa() {
		if (!this.vacia()) {
			Nodo<T> nodoActual = this.getEntrada();
			Nodo<T> nodoAux = null;
			while (nodoActual.getSiguiente() != null) {
				nodoAux = nodoActual.getSiguiente();
				nodoActual.setSiguiente(nodoAux.getSiguiente());
				nodoAux.setSiguiente(this.getEntrada());
				this.setEntrada(nodoAux);
			}
		}
	}
```

Colecciones - Lista Simplemente Enlazada

Lista SE:

```
public class ListaSimple<T> {

    public void insertar(Integer posicion, T elemento) {
        if (elemento == null)
            throw new IllegalArgumentException();
        if (posicion <= this.getCantidadElementos() + 1) {
            if (posicion == 1) {
                this.insertarAdelante(elemento);
            } else if (posicion == this.getCantidadElementos() + 1) {
                this.insertarAtras(elemento);
            } else {
                Nodo<T> nodoActual = this.getEntrada();
                for (Integer i = 1; i <= posicion - 2; i++)
                    nodoActual = nodoActual.getSiguiente();
                nodoActual.setSiguiente(new Nodo<T>(elemento, nodoActual.getSiguiente()));
                this.setCantidadElementos(this.getCantidadElementos() + 1);
            }
        }
    }
```

Colecciones - Lista Simplemente Enlazada

Lista SE:

```
public class ListaSimple<T> {
	public void eliminar(Integer posicion) {
		if (posicion <= this.getCantidadElementos())
			if (posicion == 1)
				this.eliminarAdelante();
		else {
			Nodo<T> nodoActual = this.getEntrada();
			for (Integer i = 1; i <= posicion - 2; i++)
				nodoActual = nodoActual.getSiguiente();
			nodoActual.setSiguiente(nodoActual.getSiguiente().getSiguiente());
			this.setCantidadElementos(this.getCantidadElementos() - 1);
		}
	}
	public boolean vacia() {
		return this.getEntrada() == null;
	}
	public boolean buscar(T elemento) {
		if (elemento == null)
			throw new IllegalArgumentException();
		Nodo<T> nodoActual = this.getEntrada();
		while ((nodoActual != null) && (!nodoActual.getElemento().equals(elemento)))
			nodoActual = nodoActual.getSiguiente();
		return (nodoActual != null);
	}
```

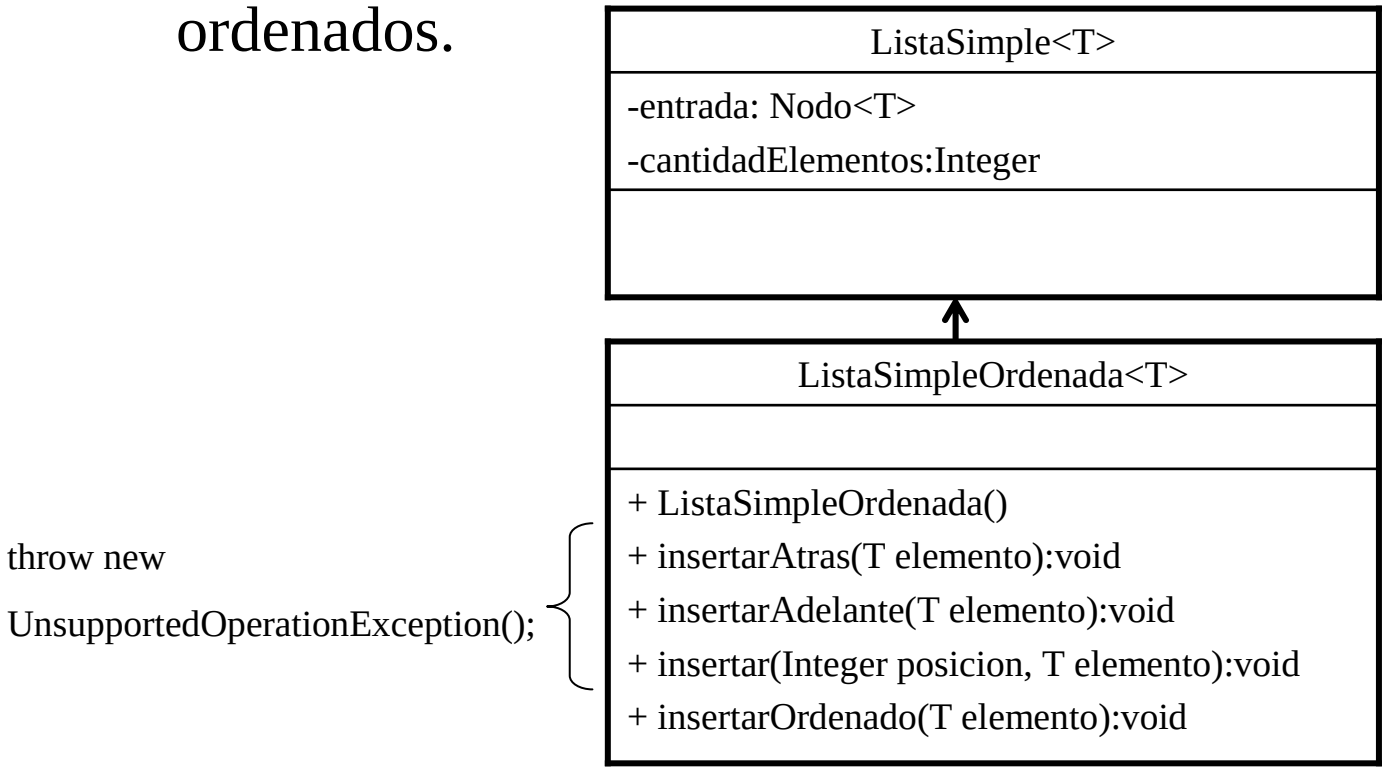
Colecciones - Lista Simplemente Enlazada

Lista SE:

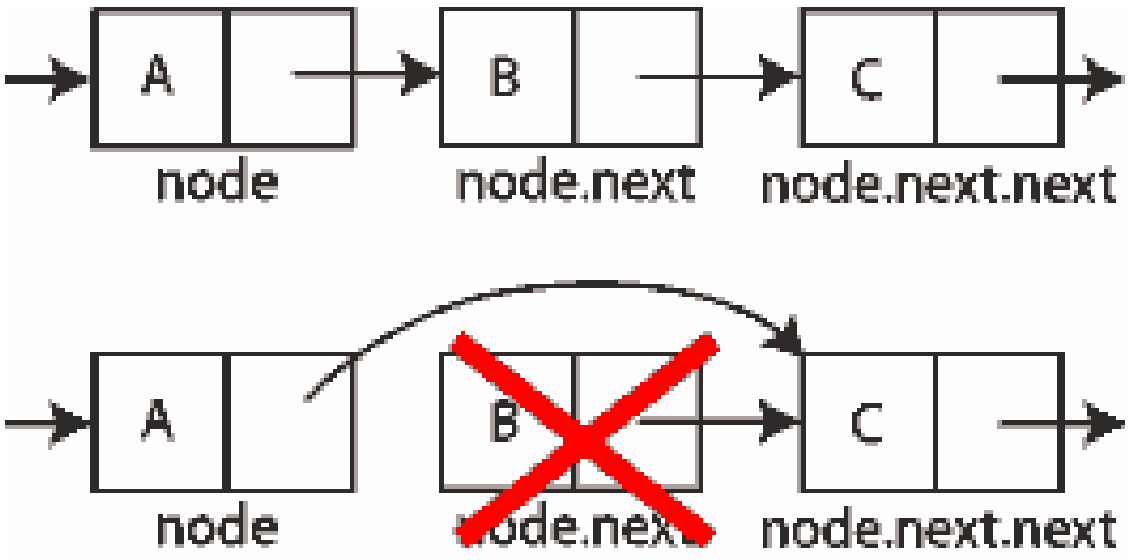
```
public class ListaSimple<T> {
	public T buscar(Integer posicion) {
		if (posicion <= 0 || posicion > this.getCantidadElementos())
			return null;
		Nodo<T> nodoActual = this.getEntrada();
		for (Integer i = 1; i < posicion; i++)
			nodoActual = nodoActual.getSiguiente();
		return (nodoActual != null) ? nodoActual.getElemento() : null;
	}
	public Integer getPosicion(T elemento) {
		if (elemento == null)
			throw new IllegalArgumentException();
		Nodo<T> nodoActual = this.getEntrada();
		Integer posicion = 0;
		while ((nodoActual != null) && (!nodoActual.getElemento().equals(elemento))) {
			posicion++;
			nodoActual = nodoActual.getSiguiente();
		}
		return (nodoActual != null) ? posicion + 1 : posicion;
	}
	public void vaciar() {
		while (!this.vacia())
			this.eliminarAdelante();
	}

}
```

Colecciones - Lista Simplemente Enlazada

- Lista Simplemente Enlazada Ordenada:
  - Una lista simplemente enlazada ordenada es una estructura lineal compuesta por una secuencia de 0 o más elementos ordenados.



Colecciones - Lista Simplemente Enlazada



Colecciones - Lista Simplemente Enlazada

Lista Simplemente Enlazada Ordenada:

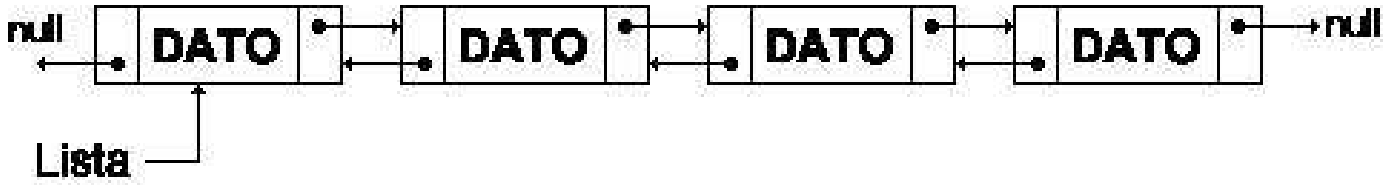
```
public class ListaSimpleOrdenada<T extends Comparable<T>> extends ListaSimple<Comparable<T>> {

	public ListaSimpleOrdenada() {
		super();
	}
	public void insertarOrdenado(T elemento) {
		if (elemento == null)
			throw new IllegalArgumentException();
		Nodo<Comparable<T>> nodoActual = this.getEntrada();
		Nodo<Comparable<T>> nodoAnterior = null;
		while ((nodoActual != null) && (nodoActual.getElemento().compareTo(elemento) < 0)) {
			nodoAnterior = nodoActual;
			nodoActual = nodoActual.getSiguiente();
		}
		if (nodoAnterior == null)
			super.insertarAdelante(elemento);
		else {
			nodoAnterior.setSiguiente(new Nodo<Comparable<T>>(elemento, nodoAnterior.getSiguiente()));
			this.setCantidadElementos(this.getCantidadElementos() + 1);
		}
	}
	public void insertar(Integer posicion, Comparable<T> elemento) { throw new UnsupportedOperationException(); }
	public void insertarAdelante(Comparable<T> elemento) { throw new UnsupportedOperationException(); }
	public void insertarAtras(Comparable<T> elemento) { throw new UnsupportedOperationException(); }

}
```

Colecciones - Lista Doblemente Enlazada

- Lista Doblemente Enlazada:
  - Una lista encadenada es una lista implementada mediante una estructura doblemente encadenada.
  - En este tipo de implementación una lista es representada con un campo en el que se guarda un apuntador al primer nodo de la lista y un campo con un apuntador al último nodo de la lista.
  - En cada nodo de la lista se guarda el elemento, un apuntador al nodo anterior y un apuntador al siguiente nodo. En el caso del último nodo de la lista el apuntador al siguiente nodo es nulo y para el caso del primer nodo de la lista el apuntador al nodo anterior es nulo.
  - La complejidad de la búsqueda en el peor caso en este tipo de estructuras es O(n), donde n es la longitud de la lista.

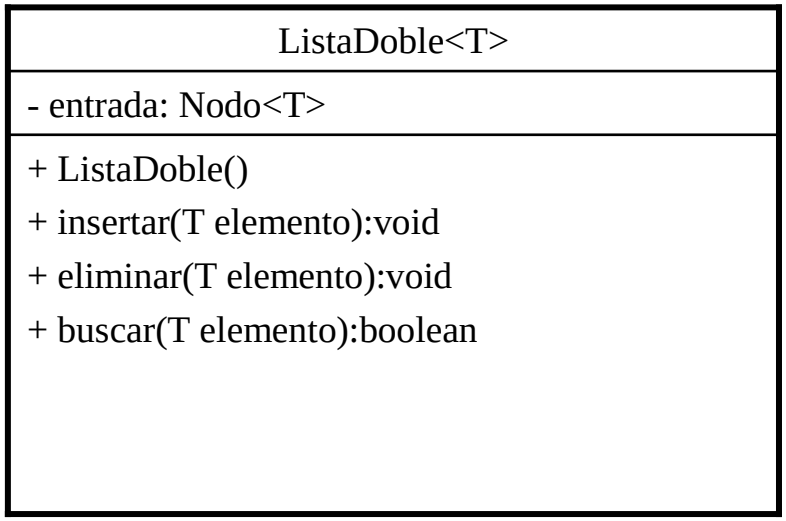
Colecciones - Lista Doblemente Enlazada

- Lista Doblemente Enlazada:



Colecciones - Lista Doblemente Enlazada

- Lista DE:



Colecciones - Lista Doblemente Enlazada

```
public class ListaDoble<T extends Comparable<T>> {
        class Nodo<T extends Comparable<T>> {
                private T elemento;
                private Nodo<T> anterior;
                private Nodo<T> siguiente;
                public Nodo() {
                                this(null, null, null);
                }
                public Nodo(T elemento) {
                                this(elemento, null, null);
                }
                public Nodo(T elemento, Nodo<T> anterior, Nodo<T> siguiente) {
                                this.setElemento(elemento);
                                this.setAnterior(anterior);
                                this.setSiguiente(siguiente);
                }
                public T getElemento() {
                                return this.elemento;
                }
                public void setElemento(T elemento) {
                                this.elemento = elemento;
                }
                public Nodo<T> getAnterior() {
                                return this.anterior;
                }
                public void setAnterior(Nodo<T> anterior) {
                                this.anterior = anterior;
                }
```

Colecciones - Lista Doblemente Enlazada

```
public class ListaDoble<T extends Comparable<T>> {

                public Nodo<T> getSiguiente() {
                        return this.siguiente;
                }
                public void setSiguiente(Nodo<T> siguiente) {
                        this.siguiente = siguiente;
                }
        }
        private Nodo<T> entrada;
        public ListaDoble() {
                this.setEntrada(null);
        }
        private Nodo<T> getEntrada() {
                return this.entrada;
        }
        private void setEntrada(Nodo<T> entrada) {
                this.entrada = entrada;
        }
```

Colecciones - Lista Doblemente Enlazada

```
public class ListaDoble<T extends Comparable<T>> {

    public void insertar(T elemento) {
        if (elemento == null)
            throw new IllegalArgumentException();
        Nodo<T> nodoNuevo = new Nodo<T>(elemento);
        Nodo<T> nodoActual = this.getEntrada();
        if ((nodoActual == null) || (nodoActual.getElemento().compareTo(elemento) > 0)) {
            nodoNuevo.setSiguiente(nodoActual);
            nodoNuevo.setAnterior(null);
            if (nodoActual != null)
                nodoActual.setAnterior(nodoNuevo);
            this.setEntrada(nodoNuevo);
        } else {
            while ((nodoActual.getSiguiente() != null) && (nodoActual.getSiguiente().getElemento().compareTo(elemento) < 0))
                nodoActual = nodoActual.getSiguiente();
            nodoNuevo.setSiguiente(nodoActual.getSiguiente());
            nodoActual.setSiguiente(nodoNuevo);
            nodoNuevo.setAnterior(nodoActual);
            if (nodoNuevo.getSiguiente() != null)
                nodoNuevo.getSiguiente().setAnterior(nodoNuevo);
        }
    }
```

Colecciones - Lista Doblemente Enlazada

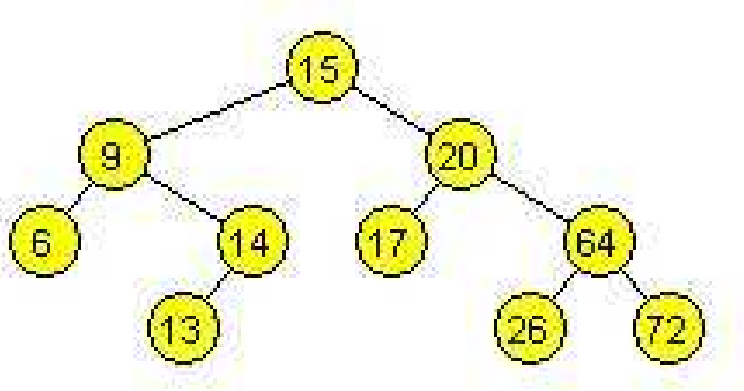
```
public class ListaDoble<T extends Comparable<T>> {

	public void eliminar(T elemento) {
		if (elemento == null)
			throw new IllegalArgumentException();
		Nodo<T> nodoActual = this.getEntrada();
		while ((nodoActual != null) && (!nodoActual.getElemento().equals(elemento)))
			nodoActual = nodoActual.getSiguiente();
		if (nodoActual != null) {
			if (nodoActual == this.getEntrada())
				this.setEntrada(nodoActual.getSiguiente());
			else
				nodoActual.getAnterior().setSiguiente(nodoActual.getSiguiente());
			if (nodoActual.getSiguiente() != null)
				nodoActual.getSiguiente().setAnterior(nodoActual.getAnterior());
		}
	}

	public boolean buscar(T elemento) {
		if (elemento == null)
			throw new IllegalArgumentException();
		Nodo<T> nodoActual = this.getEntrada();
		while ((nodoActual != null) && (!nodoActual.getElemento().equals(elemento)))
			nodoActual = nodoActual.getSiguiente();
		return (nodoActual != null);
	}

}
```

Colecciones - Árboles Binarios de Búsqueda

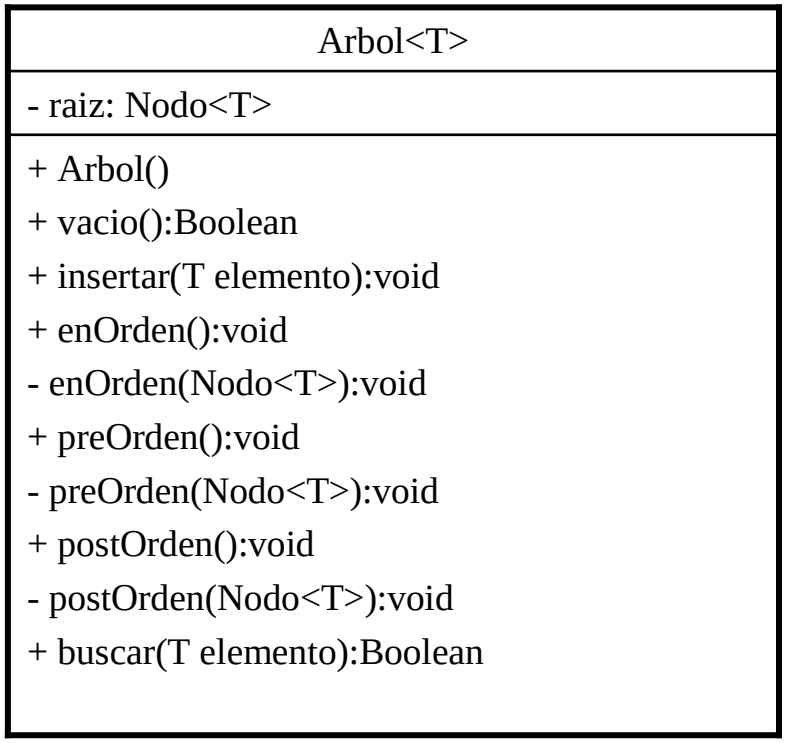
- Es una estructura no lineal en la que cada nodo puede apuntar a cero, uno o dos nodos.
- Es una estructura compuesta por un dato y varios árboles.
- Cada nodo sólo puede ser apuntado por otro nodo, es decir, cada nodo sólo tendrá un padre.
- En cuanto a la posición dentro del árbol:
  - Nodo raíz: nodo que no tiene padre. Este es el nodo que usaremos para referirnos al árbol.
  - Nodo hoja: nodo que no tiene hijos.
  - Nodo rama: son los nodos que no pertenecen a ninguna de las dos categorías anteriores.
- Todos los nodos contengan el mismo número de punteros, es decir, se usa la misma estructura para todos los nodos del árbol.
- Orden: es el número potencial de hijos que puede tener cada elemento de árbol. De este modo, diremos que un árbol en el que cada nodo puede apuntar a otros dos es de orden dos (árbol binario), si puede apuntar a tres será de orden tres, etc.
- Grado: el número de hijos que tiene el elemento con más hijos dentro del árbol.
- Nivel: se define para cada elemento del árbol como la distancia a la raíz, medida en nodos. El nivel de la raíz es cero y el de sus hijos uno. Así sucesivamente.
- Altura: la altura de un árbol se define como el nivel del nodo de mayor nivel. Como cada nodo de un árbol puede considerarse a su vez como la raíz de un árbol, también podemos hablar de altura de ramas.
- Árboles binarios de búsqueda: Árboles de orden 2 en los que se cumple que para cada nodo, el valor de la clave de la raíz del subárbol izquierdo es menor que el valor de la clave del nodo y que el valor de la clave raíz del subárbol derecho es mayor que el valor de la clave del nodo.

Colecciones - Árboles Binarios de Búsqueda

- Un árbol binario de búsqueda de tamaño 10 y profundidad 4, con raíz 15 y hojas 6, 13, 17, 26 y 72
- Recorridos:
  - **En orden** (el valor del nodo se procesa después de recorrer la primera rama y antes de recorrer la última) = [6, 9, 13, 14, 15, 17, 20, 26, 64, 72].
  - **Pre orden** (el valor del nodo se procesa antes de recorrer las ramas) = [15, 9, 6, 14, 13, 20, 17, 64, 26, 72].
  - **Post orden** (el valor del nodo se procesa después de recorrer todas las ramas.) = [6, 13, 14, 9, 17, 26, 72, 64, 20, 15].



Colecciones - Árboles Binarios de Búsqueda



Colecciones - Árboles Binarios de Búsqueda

```
public class Arbol<T extends Comparable<T>> {
        class Nodo<T extends Comparable<T>> {
                private T elemento;
                private Nodo<T> derecha;
                private Nodo<T> izquierda;
                public Nodo() {
                                this(null, null, null);
                }
                public Nodo(T elemento, Nodo<T> derecha, Nodo<T> izquierda) {
                                this.setElemento(elemento);
                                this.setDerecha(derecha);
                                this.setIzquierda(izquierda);
                }
                public Nodo(T elemento) {
                                this(elemento, null, null);
                }
                public T getElemento() {
                                return this.elemento;
                }
                public void setElemento(T elemento) {
                                this.elemento = elemento;
                }
                public Nodo<T> getDerecha() {
                                return this.derecha;
                }
                public void setDerecha(Nodo<T> derecha) {
                                this.derecha = derecha;
                }
```

Colecciones - Árboles Binarios de Búsqueda

```
public class Arbol<T extends Comparable<T>> {

                public Nodo<T> getIzquierda() {
                                return this.izquierda;
                }
                public void setIzquierda(Nodo<T> izquierda) {
                                this.izquierda = izquierda;
                }
        }
        private Nodo<T> raiz;
        public Arbol() {
                this.setRaiz(null);
        }
        private void setRaiz(Nodo<T> raiz) {
                this.raiz = raiz;
        }
        private Nodo<T> getRaiz() {
                return this.raiz;
        }
```

Colecciones - Árboles Binarios de Búsqueda

```
public class Arbol<T extends Comparable<T>> {

	public Boolean vacio() {
		return this.getRaiz() == null;
	}
	public void insertar(T elemento) {
		if (elemento == null)
			throw new IllegalArgumentException();
		Nodo<T> nodoNuevo = new Nodo<T>(elemento, null, null);
		Nodo<T> nodoPadre = null;
		Nodo<T> nodoActual = this.getRaiz();
		while ((nodoActual != null) && (!nodoActual.getElemento().equals(elemento))) {
			nodoPadre = nodoActual;
			if (nodoActual.getElemento().compareTo(elemento) > 0)
				nodoActual = nodoActual.getIzquierda();
			else
				nodoActual = nodoActual.getDerecha();
		}
		if (nodoActual != null)
			return;
		if (nodoPadre == null)
			this.setRaiz(nodoNuevo);
		else if (nodoPadre.getElemento().compareTo(elemento) > 0)
			nodoPadre.setIzquierda(nodoNuevo);
		else
			nodoPadre.setDerecha(nodoNuevo);
	}
```

Colecciones - Árboles Binarios de Búsqueda

```
public class Arbol<T extends Comparable<T>> {

        public void enOrden() {
                if (! this.vacio())
                        this.enOrden(this.getRaiz());
        }
        private void enOrden(Nodo<T> raiz) {
                if (raiz.getIzquierda() != null)
                        this.enOrden(raiz.getIzquierda());
                System.out.println(raiz.getElemento());
                if (raiz.getDerecha() != null)
                        this.enOrden(raiz.getDerecha());
        }
        public void preOrden() {
                 if (! this.vacio())
                        this.preOrden(this.getRaiz());
        }
        private void preOrden(Nodo<T> raiz) {
                System.out.println(raiz.getElemento());
                if (raiz.getIzquierda() != null)
                        this.preOrden(raiz.getIzquierda());
                if (raiz.getDerecha() != null)
                        this.preOrden(raiz.getDerecha());
        }
        public void postOrden() {
                 if (! this.vacio())
                        this.postOrden(this.getRaiz());
        }
```

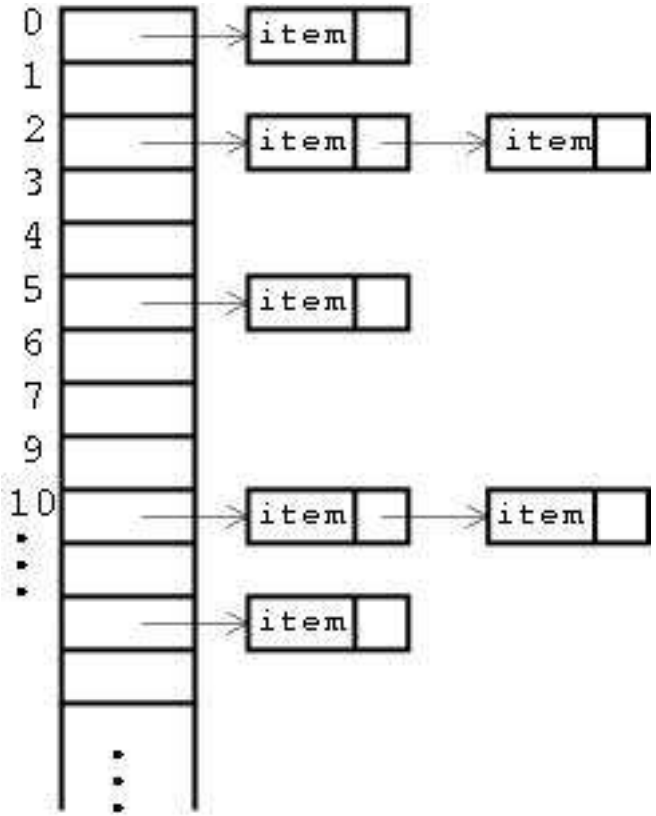
Colecciones - Árboles Binarios de Búsqueda

```
public class Arbol<T extends Comparable<T>> {

	private void postOrden(Nodo<T> raiz) {
		if (raiz.getIzquierda() != null)
			this.postOrden(raiz.getIzquierda());
		if (raiz.getDerecha() != null)
			this.postOrden(raiz.getDerecha());
		System.out.println(raiz.getElemento());
	}
	public Boolean buscar(T elemento) {
		Nodo<T> nodoActual = this.getRaiz();
		while (nodoActual != null) {
			if (nodoActual.getElemento().equals(elemento))
				return true;
			if (nodoActual.getElemento().compareTo(elemento) > 0)
				nodoActual = nodoActual.getIzquierda();
			else
				nodoActual = nodoActual.getDerecha();
		}
		return false;
	}

}
```

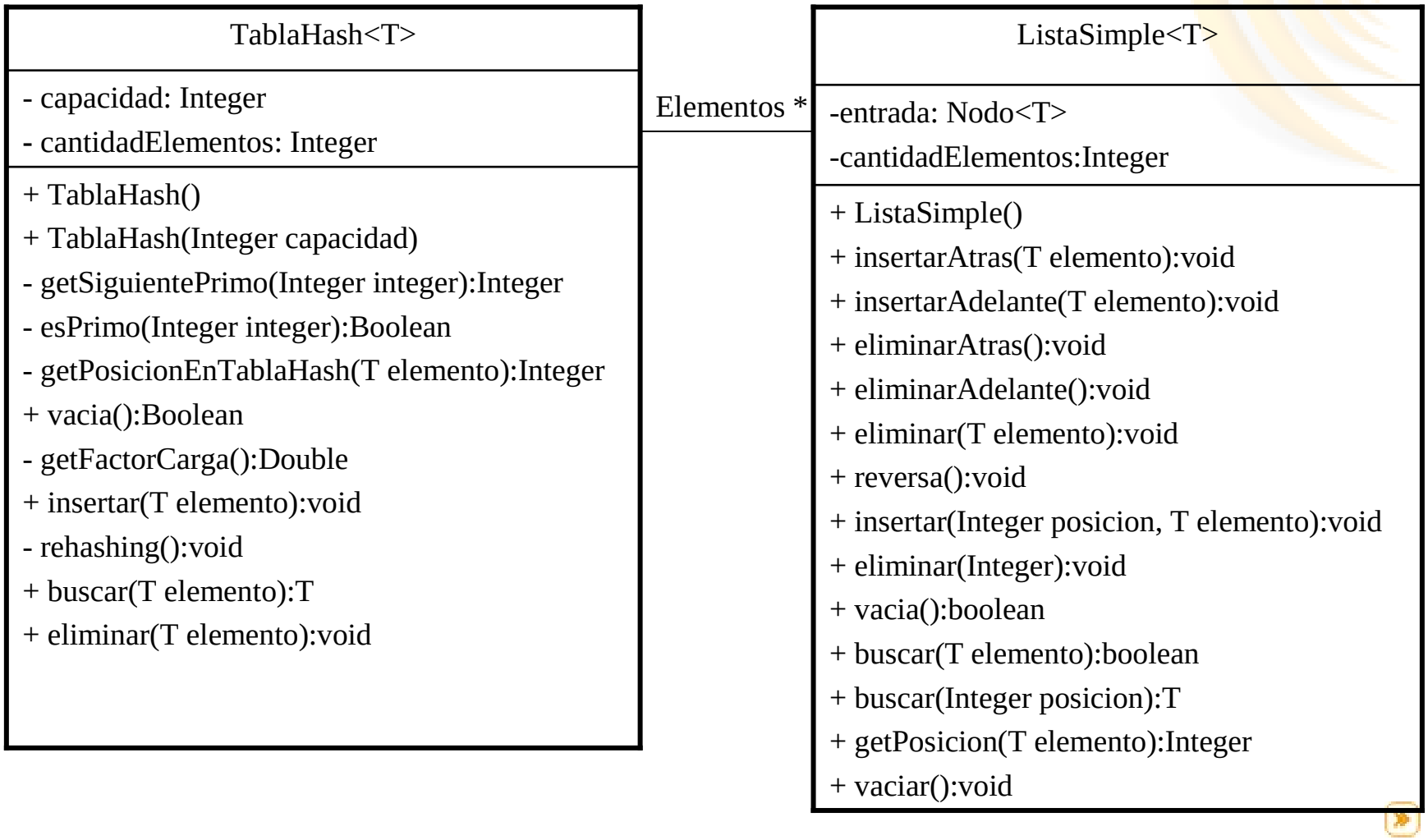
Colecciones - Tabla Hash

- Una tabla de Hashing es una estructura de acceso directo, en la que cada elemento tiene asociado una llave (contenedor asociativo). La llave de un elemento es un identificador que tiene significado en el contexto del problema en el que este siendo utilizada la tabla.
- En una tabla de Hashing no existe noción de orden tanto para los elementos como para sus respectivas llaves, no hay anterior ni sucesor, ni primero ni último.
- No es posible realizar recorrido alguno sobre las llaves ni elementos.
- La única forma de acceder a los elementos de la tabla es mediante sus llaves asociadas.
- La ventaja principal de la tabla de Hashing se encuentra en su efectividad para consultar los elementos a través de su llave de forma directa.
- Es apropiado utilizar tablas de Hashing cuando el espacio de llaves es significativamente mayor al área primaria.
- Ejemplos de Aplicación de Tablas de Dispersión:
  - Corrector Ortográfico,
  - Tablas de Símbolos de un Compilador, etc.
- Una tabla Hash ideal permite acceder a los valores clave en tiempo O(1).

Colecciones - Tabla Hash



Colecciones - Tabla Hash



Colecciones - Tabla Hash

```
public class TablaHash<T> {
        private ListaSimple<T> elementos[];
        private Integer capacidad;
        private Integer cantidadElementos;
        public TablaHash() { this(50); }
        public TablaHash(Integer capacidad) {
                this.setCapacidad(this.getSiguientePrimo((int) (capacidad / 0.75)));
                this.setElementos(new ListaSimple[this.getCapacidad()]);
                for (int i = 0; i < elementos.length; i++)
                        this.getElementos()[i] = new ListaSimple<T>();
                this.setCantidadElementos(0);
        }
        public Integer getSiguientePrimo(Integer integer) {
                Integer siguientePrimo = integer;
                while (true) {
                        siguientePrimo++;
                        if (esPrimo(siguientePrimo)) break;
                }
                return siguientePrimo;
        }
        private Boolean esPrimo(Integer integer) {
                Integer contador = 1;   Integer raices = 0;
                while (contador <= integer) {
                        if (integer % contador == 0)  raices++;
                        contador++;
                }
                return (raices == 2);
        }
```

Área primaria: es el espacio físico en el que se colocan los elementos de la estructura.

Capacidad: es el tamaño del área primaria.

Tamaño: es el número de elementos presentes en la tabla.

Para evitar la tendencia de que los hash de enteros grandes tengan divisores comunes con el tamaño de la tabla hash, lo que provocaría colisiones tras el cálculo del módulo. Mayor uniformidad en la distribución

Espacio de llaves: es el conjunto de todas las llaves que pueden existir. Conjunto de números enteros no negativos.

Número Primo: solo se divide por 1 y por sí mismo. Ejemplo: 2,3,5,7,11,13,17,19,23,29,31

Colecciones - Tabla Hash

```
public class TablaHash<T> {
        public ListaSimple<T>[] getElementos() {
                return this.elementos;
        }
        public void setElementos(ListaSimple<T>[] elementos) {
                this.elementos = elementos;
        }
        public Integer getCapacidad() {
                return this.capacidad;
        }
        public void setCapacidad(Integer capacidad) {
                this.capacidad = capacidad;
        }
        public Integer getCantidadElementos() {
                return this.cantidadElementos;
        }
        public void setCantidadElementos(Integer cantidadElementos) {
                this.cantidadElementos = cantidadElementos;
        }
        private Integer getPosicionEnTablaHash(T elemento) {
                Integer indiceHashX = elemento.hashCode() % this.getCapacidad();
                if (indiceHashX < 0)
                                indiceHashX += this.getCapacidad();
                return indiceHashX;
        }
```

Dirección de un elemento: es la posición que un elemento ocupa en el área primaria. Resulta necesario utilizar una función que, dado un Objeto cualquiera, le asocie un número que permita indexar la tabla. Se utiliza el operador módulo (resto de la división entera): % para obtener un índice adecuado para una Tabla Hash. ([0, capacidadDelArray-1])

Colisión: ocurre cuando dos llaves diferentes son proyectadas sobre la misma dirección en la llave primaria. la eficiencia de la tabla de Hashing radica en la forma en la que resuelve este tipo de conflictos (Hashing abierto)

Función de Hashing: es la función que transforma una llave en una dirección en el área primaria. Esta función es que permite que una tabla localice el elemento deseado.

```
public class TablaHash<T> {
        public Boolean vacia() {
                return (this.getCantidadElementos() == 0);
        }
        private Double getFactorCarga() {
                return (double) ((this.getCantidadElementos()) / this.getCapacidad());
        }
        public void insertar(T elemento) {
                if (elemento == null)  throw new IllegalArgumentException();
                ListaSimple<T> listaSimple = this.getElementos()[this.getPosicionEnTablaHash(elemento)];
                if (!listaSimple.buscar(elemento)) {
                                listaSimple.insertarAtras(elemento);
                                this.setCantidadElementos(this.getCantidadElementos() + 1);
                                if (this.getFactorCarga() > 1.5)
                                                rehashing();
                }
        }
        private void rehashing() {
                Integer capacidadAnterior = this.getCapacidad();
                ListaSimple<T> elementosAnterior[] = new ListaSimple[this.getCapacidad()];
                for (Integer i = 0; i < this.getCapacidad(); i++)
                                elementosAnterior[i] = this.getElementos()[i];
                this.setCapacidad(this.getSiguientePrimo((int) (this.getCapacidad() * 2 / 0.75)));
                this.setElementos(new ListaSimple[this.getCapacidad()]);
                for (Integer i = 0; i < this.getCapacidad(); i++)
                                this.getElementos()[i] = new ListaSimple<T>();
                for (Integer i = 0; i < capacidadAnterior; i++)
                                if (elementosAnterior[i] != null)
                                                for (Integer j = 1; j <= elementosAnterior[i].getCantidadElementos(); j++)
                                                                if (elementosAnterior[i].buscar(j) != null)
                                                                                this.insertar(elementosAnterior[i].buscar(j));
        }
```

Factor de carga: es el tamaño de la tabla sobre su capacidad. Indica que tan saturada la tabla se encuentra. Una buena función hash deberá tender a que el factor de carga esté cerca a 1

Rehashing: Se redimensiona el vector (área primaria) duplicando su capacidad.

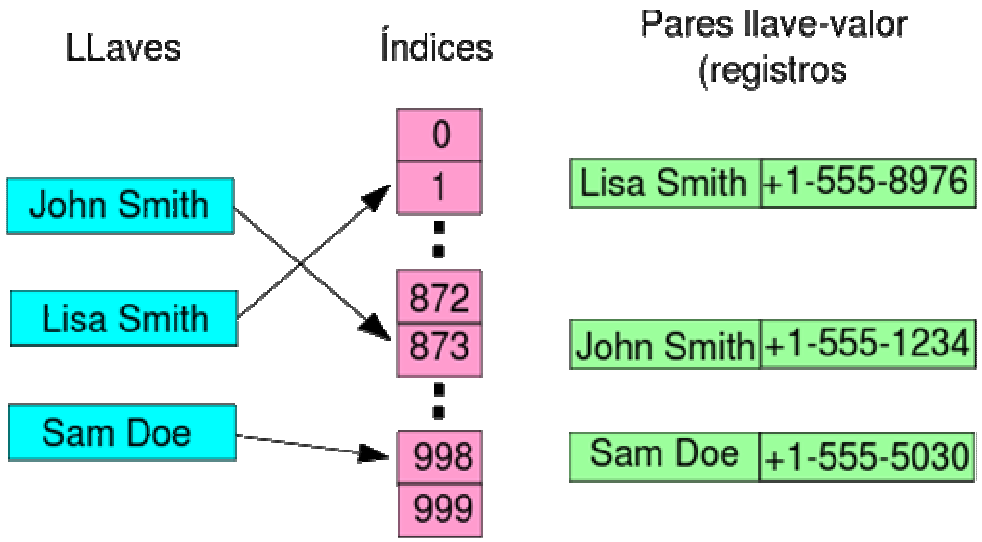
Colecciones - Tabla Hash

```
public class TablaHash<T> {
	public T buscar(T elemento) {
		if (elemento == null)
			throw new IllegalArgumentException();
		return this.getElementos()[this.getPosicionEnTablaHash(elemento)].buscar(this.getElementos()[this
			.getPosicionEnTablaHash(elemento)].getPosicion(elemento));
	}
	public void eliminar(T elemento) {
		if (elemento == null)
			throw new IllegalArgumentException();
		ListaSimple<T> listaSimple = this.getElementos()[this.getPosicionEnTablaHash(elemento)];
		if (!listaSimple.buscar(elemento))
			throw new NoSuchElementException();
		listaSimple.eliminar(elemento);
		this.setCantidadElementos(this.getCantidadElementos() - 1);
	}
	public String toString() {
		String res = "";
		for (Integer i = 0; i < this.getCapacidad(); i++)
			res = res.concat(String.valueOf(i)).concat(" :
	").concat(this.getElementos()[i].toString()).concat("\n");
		return res;
	}

}
```

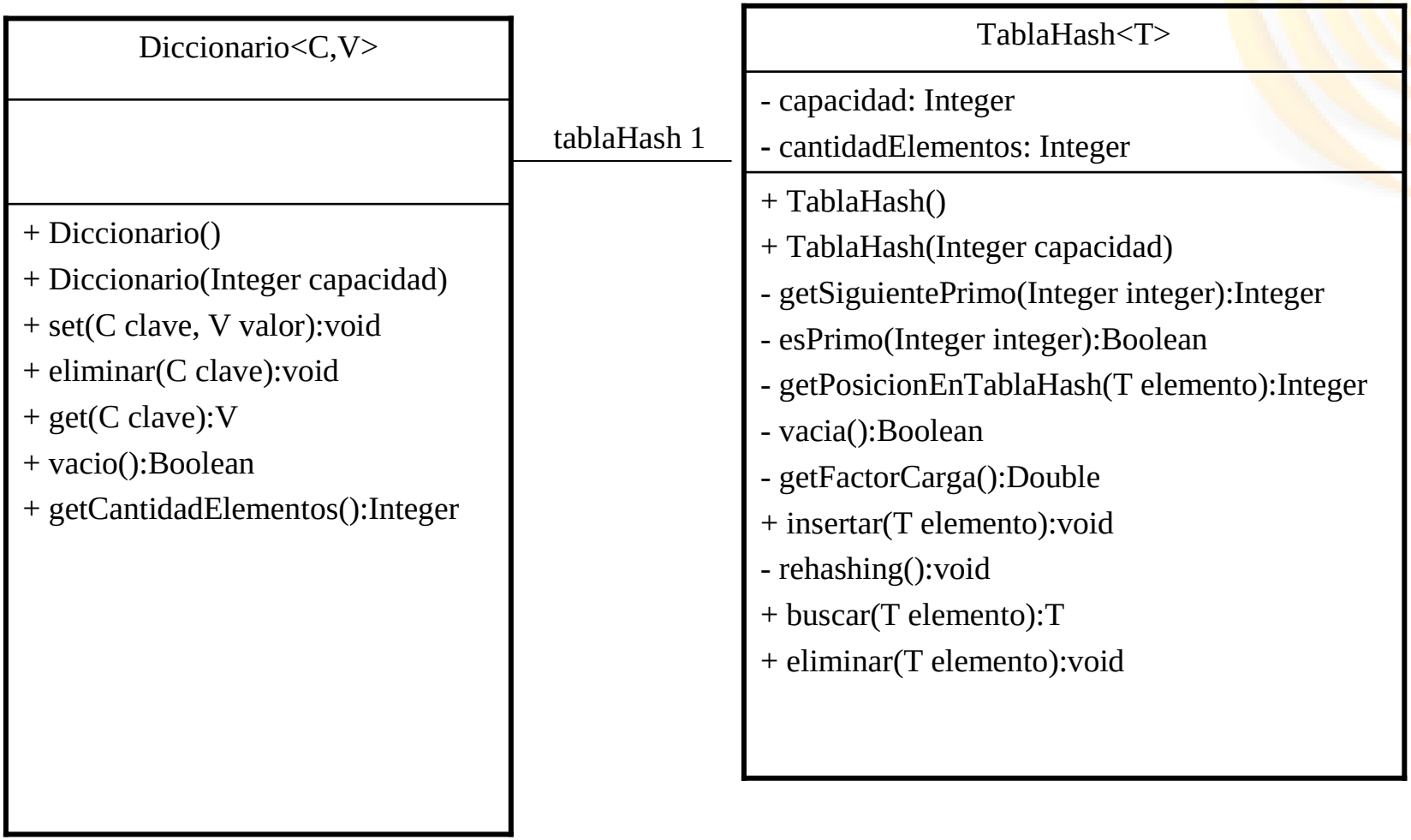
Colecciones - Diccionario

- El TAD diccionario es un conjunto S de elementos con una clave y una información asociadas. Las claves deben ser comparables entre ellas (es decir, el conjunto de claves están ordenadas bajo un orden lineal). Las operaciones definidas para el TAD diccionario son:
  - crear un diccionario vacío,
  - determinar (buscando su clave) si un elemento dado esta o no en el diccionario,
  - insertar un elemento dado en el diccionario y
  - borrar un elemento dado del diccionario.

Colecciones - Diccionario



Colecciones - Diccionario



Colecciones - Diccionario

```
public class Diccionario<C, V> {
        class EntradaDiccionario<C, V> {
                private C clave;
                private V valor;
                public EntradaDiccionario(C clave) {
                                this(clave, null);
                }
                public EntradaDiccionario(C clave, V valor) {
                                this.setClave(clave);
                                this.setValor(valor);
                }
                public C getClave() {
                                return this.clave;
                }
                public void setClave(C clave) {
                                this.clave = clave;
                }
                public V getValor() {
                                return this.valor;
                }
                public void setValor(V valor) {
                                this.valor = valor;
                }
```

Colecciones - Diccionario

```
public class Diccionario<C, V> {
        class EntradaDiccionario<C, V> {
                public boolean equals(Object objeto) {
                        if (this == objeto) return true;
                        if (objeto == null) return false;
                        if (this.getClass() != objeto.getClass()) return false;
                        EntradaDiccionario<C, V> entradaDiccionario = (EntradaDiccionario<C, V>) objeto;
                        return (this.getClave().equals(entradaDiccionario.getClave()));
                }
                public int hashCode() {
                        return this.getClave().hashCode();
                }
                public String toString() {
                        return "(" + this.getClave() + "," + this.getValor() + ")";
                }
        }
        private TablaHash<EntradaDiccionario<C, V>> tablaHash;
        public Diccionario() {
                this.setTablaHash(new TablaHash<EntradaDiccionario<C, V>>());
        }
        public Diccionario(Integer capacidad) {
                this.setTablaHash(new TablaHash<EntradaDiccionario<C, V>>(capacidad));
        }
        public TablaHash<EntradaDiccionario<C, V>> getTablaHash() {
                return this.tablaHash;
        }
        public void setTablaHash(TablaHash<EntradaDiccionario<C, V>> tablaHash) {
                this.tablaHash = tablaHash;
        }
```

Colecciones - Diccionario

```
public class Diccionario<C, V> {
        public void set(C clave, V valor) {
                EntradaDiccionario<C, V> entradaDiccionario = this.getTablaHash().buscar(new EntradaDiccionario<C, V>(clave));
                if (entradaDiccionario != null)
                        entradaDiccionario.setValor(valor);
                else
                        this.getTablaHash().insertar(new EntradaDiccionario<C, V>(clave, valor));
        }
        public void eliminar(C clave) {
                this.getTablaHash().eliminar(new EntradaDiccionario<C, V>(clave));
        }
        public V get(C clave) {
                return this.getTablaHash().buscar(new EntradaDiccionario<C, V>(clave)).getValor();
        }
        public Boolean vacio() {
                return this.getTablaHash().vacia();
        }
        public Integer getCantidadElementos() {
                return this.getTablaHash().getCantidadElementos();
        }

}
```

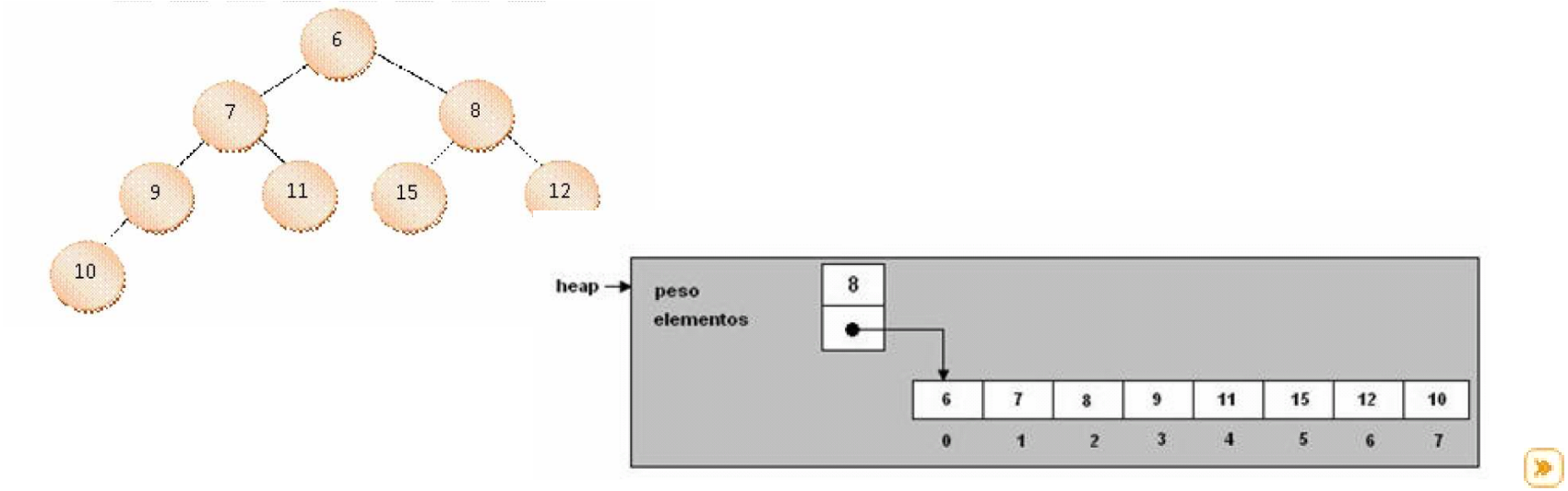
Si la clave existe, setear nuevo valor a la clave.

Colecciones - Montículo

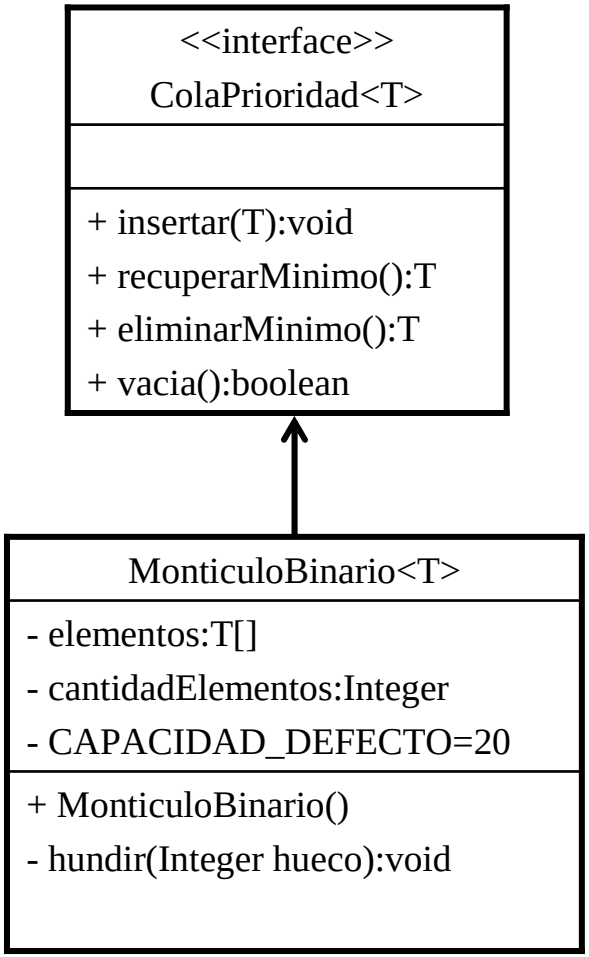
- Un heap es una estructura de datos similar a un árbol binario, pero donde se tienen que cumplir las siguientes características:
  - Es un árbol binario completo. Esto quiere decir que a cada nivel del árbol está lleno, excepto en el último nivel. En este nivel el llenado debe ser de izquierda a derecha.
  - Satisface la propiedad heap. Esto significa que los elementos que se encuentran en los nodos hijos son mayores o iguales al elemento guardado en el nodo padre. Esto implica que el elemento más pequeño siempre se encontrara en la raíz del árbol

Colecciones - Montículo

- Para poder representar un heap usando un arreglo, se deben numerar los nodos de arriba hacia abajo y de izquierda a derecha. Este número representara la posición en el arreglo que tendrá en nodo (Es necesario tener en cuenta que las posiciones de un arreglo comienzan desde 0).



Colecciones - Montículo



Colecciones - Montículo

```
public interface ColaPrioridad<T extends Comparable<T>> {

        void insertar(T elemento) throws MonticuloBinarioLlenoException;
        T recuperarMinimo();
        T eliminarMinimo();
        boolean vacia();

}
```

```
public class MonticuloBinario<T extends Comparable<T>> implements ColaPrioridad<T> {
        private T elementos[];
        private Integer cantidadElementos;
        private static final Integer CAPACIDAD_POR_DEFECTO = 20;
        public MonticuloBinario() {
                this.setElementos((T[]) new Comparable[CAPACIDAD_POR_DEFECTO]);
                this.setCantidadElementos(0);
        }
        public T[] getElementos() {
                return this.elementos;
        }
        public void setElementos(T[] elementos) {
                this.elementos = elementos;
        }
        public Integer getCantidadElementos() {
                return this.cantidadElementos;
        }
```

Colecciones - Montículo

```
public class MonticuloBinario<T extends Comparable<T>> implements ColaPrioridad<T> {
	public void setCantidadElementos(Integer cantidadElementos) {
		this.cantidadElementos = cantidadElementos;
	}
	public void insertar(T elemento) throws MonticuloBinarioLlenoException {
		if (elemento == null)
			throw new IllegalArgumentException();
		if (this.getCantidadElementos() == this.getElementos().length - 1)
			throw new MonticuloBinarioLlenoException();
		this.setCantidadElementos(this.getCantidadElementos() + 1);
		Integer hueco = this.getCantidadElementos();
		while (hueco > 1 && elemento.compareTo(this.getElementos()[hueco / 2]) < 0) {
			this.getElementos()[hueco] = this.getElementos()[hueco / 2];
			hueco = hueco / 2;
		}
		this.getElementos()[hueco] = elemento;
	}
	public T eliminarMinimo() {
		T elMinimo = this.recuperarMinimo();
		this.setCantidadElementos(this.getCantidadElementos() - 1);
		this.getElementos()[1] = this.getElementos()[this.getCantidadElementos()];
		this.hundir(1);
		return elMinimo;
	}
```

Colecciones - Montículo

```
public class MonticuloBinario<T extends Comparable<T>> implements ColaPrioridad<T> {
	private void hundir(Integer hueco) {
		T aux = this.getElementos()[hueco];
		Integer hijo = hueco * 2;
		Boolean esHeap = Boolean.FALSE;
		while (hijo <= this.getCantidadElementos() && !esHeap) {
			if (hijo != this.getCantidadElementos() && this.getElementos()[hijo + 1].compareTo(this.getElementos()[hijo]) < 0)
				hijo++;
			if (this.getElementos()[hijo].compareTo(aux) < 0) {
				this.getElementos()[hueco] = this.getElementos()[hijo];
				hueco = hijo;
				hijo = hueco * 2;
			} else
				esHeap = Boolean.TRUE;
		}
		this.getElementos()[hueco] = aux;
	}
	public T recuperarMinimo() {
		return this.getElementos()[1];
	}
	public boolean vacia() {
		return (this.getCantidadElementos() == 0);
	}

}
```

Colecciones - Ejercicios

- **Ejercicio 1:** En una terminal de teletipo existe un carácter de retroceso que permite cancelar el último carácter. Por ejemplo: si el carácter de retroceso es /, entonces la línea abc/d//e será interpretada como ae. Existe también un carácter anulador que elimina todos los caracteres ingresados hasta el momento, suponga que ese carácter es &. Realice un método que dada una tira de caracteres terminadas con * (leída del archivo “Fuente.txt”) ejecute las operaciones indicadas si se encuentra con el carácter / o el &. Debe por ultimo imprimir la tira resultante.
- **Ejercicio 2:** En un supermercado se mantiene una cola A con diversos clientes de los que se conoce número de ubicación en la cola y cantidad de productos que lleva. Se abre una nueva cola B para clientes que llevan menos de 5 productos. Usted debe dejar en la cola A los clientes que llevan más de 5 o hasta 5 productos en el orden en que estaban, y en la cola B los que compran menos de 5 artículos, respetando el orden que tenían en la cola A. En ambas colas reasignar un nuevo número de ubicación.
- **Ejercicio 3:** Se tiene ordenada por código de producto, una lista donde cada Producto, tiene como atributos código, descripción, importe y stock. Implementar métodos para:
  - A) Imprimir la lista completa.
  - B) Dado un entero k, imprimir el k-ésimo elemento.
  - C) Incrementar el importe de un producto dado en un 10%.
  - D) Devolver el stock de un producto dado o cero si no esta.
  - E) Devolver una lista de productos con stock inferior a 50 unidades.
  - F) Devolver una lista de productos con stock superior o igual a 50 unidades.

Excepciones

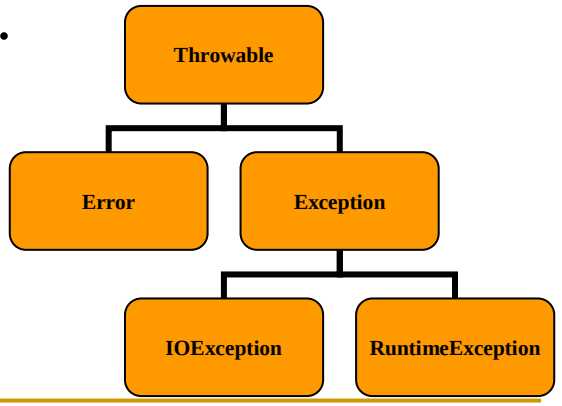
- Introducción:
  - Representan condiciones excepcionales que el programador quiere tratar.
  - La clase **Exception** extiende la clase **Throwable**.
  - La clase **Throwable** provee características útiles para tratar con excepciones.
  - Específicamente:
    - provee un slot para un mensaje.
    - contiene un stack trace.

Excepciones

- Manejo de errores en tiempo de ejecución:
  - Introducción de datos de formato incorrecto. (**NumberFormatException**)
  - Archivo con información incorrecta. (**IOException**)
  - Índice matricial incorrecto. (**ArrayIndexOutOfBoundsException**)
  - null.método() (**NullPointerException**)
- Usuarios esperan que el programa se comporte de forma sensata cuando se producen errores → volver a un estado seguro.
  - Refundiciones incorrectas. (**ClassCastException**)

Excepciones

- Clasificación de excepciones:
  - Error → describe errores internos y el agotamiento de recursos dentro del sistema de ejecución de Java. No se deben lanzar objetos de este tipo. No recuperable. No comprobada.
  - Exception → Se produce porque se ha cometido un error de programación. Recuperable. Comprobada.
  - RuntimeException → No Comprobada.



Excepciones

- Tipos de excepciones:
  - Comprobadas:
    - Compilador verifica que el programador proporcione un manejador.
    - Ejemplo: FileNoFoundException, SQLException, etc.
  - No Comprobadas:
    - Excepciones comunes, como acceder a una referencia nula que no se comprueban.
    - Ejemplo: Error y todas las subclases de RuntimeException son excepciones no comprobadas (NullPointerException, ArrayIndexOutOfBoundException, NumberFormatException, ClassCastException, ArithmeticException, etc.)

Excepciones

- Declaración:
  - Para crear excepciones propias generalmente se hace una subclase de **Exception**.

```
package nombrePaquete;
{importaciones}
[modificadores] class nombreException extends
  Exception {
}
```

Excepciones

- ¿Cómo se causan excepciones?
  - Implícitamente: el programa hace algo ilegal.
  - Explícitamente: ejecución de la sentencia **throw**.

```
class SinNaftaException extends Exception { }
class Auto {
   ...
   if (nafta < 0.1)
       throw new SinNaftaException();
   ...
}
```

Excepciones

- ¿Cómo manejar una excepción?

```
try {
   // Código que levanta una excepción.
}
catch (TipoExcepción nombre) {
   // Código que se ejecuta en caso de
   excepción.
}
finally {
   // Código que se ejecuta siempre.
   // Es decir, si se produce o no la exception.
}
```

Excepciones

- ¿Cómo manejar múltiples excepciones?

```
try {
   sentencias;
}
catch (TipoExcepcion1 nombre) {

}
catch (TipoExcepcion2 nombre) {

}
// En aumento a Exception (de lo particular a
   lo general); al revés no compila
```

Excepciones

- Ejemplo

```
public Object pop() throws SinElementosException {
   if (this.getElementos().isEmpty()) {
      throw new SinElementoException();
   } else {
       return this.getElementos().removeLast();
   }}
   ...
   try {
      elemento = pila.pop();
   } catch (SinElementosException e) {
       e.printStackTrace();
}
```

Excepciones

Comportamiento de tipo reanudación

```
boolean continuar = true;
while(continuar) {
  try {
    System.out.print("Introduce un número entero: ");
    InputStreamReader datosInsertados = new InputStreamReader(System.in);
    BufferedReader datos = new BufferedReader(datosInsertados);
    String cadenaDatos = datos.readLine();
    int numero = Integer.parseInt(cadenaDatos);
    int cuadrado = numero * numero;
    System.out.println("El cuadrado de " + numero + " = " + cuadrado);
    continuar = false;
  } catch(Exception e) {
      System.out.println(e.getMessage());
  }
}
```

Excepciones

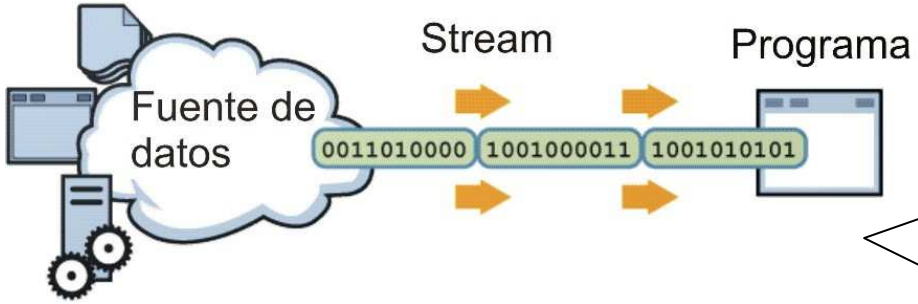
- Sobre el uso de excepciones:
  - Una condición de error es tratada sólo donde tiene sentido hacerlo y no en todo el nivel entre que ocurre y es tratada.
  - El código puede ser escrito como si todas las operaciones funcionaran correctamente.
  - Deben ser tratadas lo más específicamente posible.
  - No se deben dejar vacíos o solo imprimiendo el stack trace los bloques catch.
  - Stack Trace (Seguimiento de Pila):
    - Listado de todas las llamadas a métodos pendientes en un determinado momento de la ejecución de un programa.

Archivos

- Secuencia
  - Secuencia de entrada: Un objeto del que se puede leer una sucesión de bytes. (InputStream)
  - Secuencia de salida: Un objeto del que se puede escribir una sucesión de bytes. (OutputStream)
- La clase File
  - No sirve ni para leer ni para escribir en un archivo, sino que sirve para trabajar con el sistema de archivos de la máquina del usuario.
  - Puede representar tanto archivo como directorio.

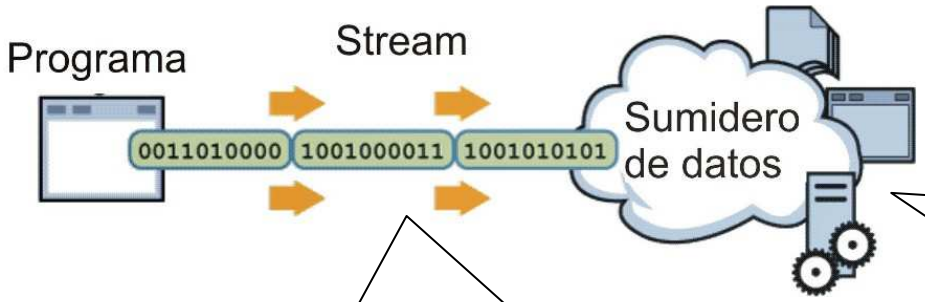
Archivos

Flujo de ENTRADA (BufferedReader): abstracción que consume información



Operaciones de Flujo:

1. Apertura del Flujo
2. Lectura/Escritura en el flujo
3. Cierre del Flujo



Flujo de SALIDA (BufferedWriter): abstracción que produce información

File: Representa un archivo o directorio (delete():boolean, exists():boolean, getPath():String, lastModified():long, length():long)

Archivos

```
File f = new File(“c:\\prueba.txt”);
       = new File(“c:\\”, “prueba.txt”);
       = new File(File directorio, String nombre);
canRead(): boolean
canWrite():boolean

delete():boolean
exits():boolean

getCanonicalFile(): File
getCannonicalPath(): String
getName(): String
getParent(): String
```

Archivos

```
getParentFile(): File
getPath(): String

isDirectory(): boolean
isFile(): boolean
isHidden(): boolean

lastModified(): long
length(): long

list(): String[]
listFiles(): File[]
list(FilenameFilter): String[]
listFiles(FilenameFilter): File[]
```

Archivos

```
createNewFile(): boolean

mkdir(): boolean

renameTo(File): boolean

setLastModified(long fecha): boolean

setReadOnly(): boolean

toURL(): URL
```

Archivos

- La interfaz FilenameFilter

```
public class FiltroExtensiones implements
   FilenameFilter
{
   private String extension;
   public FiltroExtensiones(String extension) {
       this.extension = extension;
   }
   public boolean accept(File dir, String nombre) {
       return nombre.endsWith(extension);
   }
}
```

Archivos

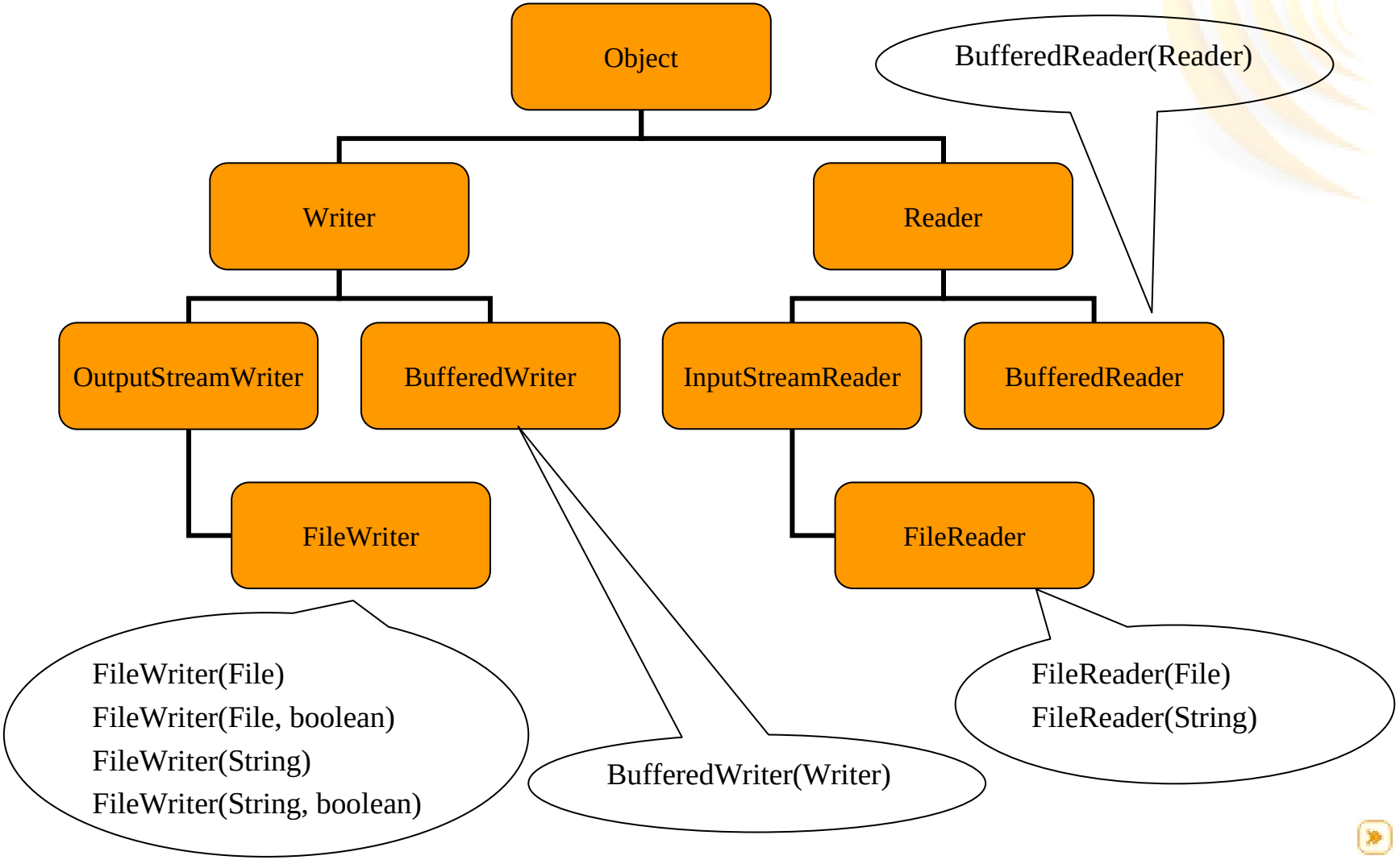
Lectura de Objetos

```
ObjectInputStream ois = new ObjectInputStream(new
   FileInputStream(new File("c:\\prueba.dat")));
Punto p = (Punto) ois.readObject();
System.out.println(p.getX());
```

Escritura de Objetos

```
ObjectOutputStream oos = new ObjectOutputStream(new
   FileOutputStream(new File("c:\\prueba.dat")));
oos.writeObject(new Punto(1,2));
```

Archivos


Object
Writer
Reader
OutputStreamWriter
BufferedWriter
InputStreamReader
BufferedReader
FileWriter
FileReader
BufferedReader(Reader)
FileWriter(File)
FileWriter(File, boolean)
FileWriter(String)
FileWriter(String, boolean)
BufferedWriter(Writer)
FileReader(File)
FileReader(String)

Archivos

Lectura de Texto

```
BufferedReader br = new BufferedReader(new
   FileReader("c:\\prueba.dat"));
while((linea=br.readLine()) != null)
   System.out.println(linea);
//otra forma de leer!!!
for (; in.ready(); linea += in.readLine() + "\n");
```

Escritura de Texto

```
BufferedWriter bw = new BufferedWriter(new
   FileWriter("c:\\prueba.dat"));
bw.write(“hola”);
bw.newLine();
```

Logs

- El sistema de generación de logs maneja un registrador predeterminado llamado Logger.global que se puede emplear en lugar de System.out
- java.util.logging.Logger
- Sintaxis inspirada en Log4j
- Logs dirigirlos a Consola (ConsoleHandler), Archivo (FileHandler) o tubería (SocketHandler)
- Se pueden definir registradores propios

  Logger registrador = Logger.getLogger("com.miApp");
- Existen 7 niveles de registros:
  - SEVERE → fatales
  - WARNING → advertencias
  - INFO → informaciones
  - CONFIG → configuraciones
  - FINE → depuraciones
  - FINER → detalles
  - FINEST → mas detalles
- De forma predeterminada se fijan los tres primeros niveles.

  registrador.setLevel(Level.FINE); //todos los niveles de FINE y superiores.

  registrador.setLevel(Level.ALL); //todos los niveles

  registrador.setLevel(Level.OFF); //desactiva todos los niveles

Logs

- Envío de mensajes:

  registrador.warning(“Esto es un warning”);

  registrador.fine(“Depuraciones”); //= registrador.log(Level.FINE, “Depuraciones”);
- Métodos de conveniencia para seguir el flujo de ejecución:

  registrador.entering(“UnaClase”, “unMetodo”);

  registrador.throwing(“UnaClase”, “unMetodo”, Throwable);

  registrador.exiting(“UnaClase”, “unMetodo”);

Logs

Ejemplo

```
import java.util.logging.*;
public class Main {
    private static Logger logger = Logger.getLogger(Main.class.getName());
    public static void main(String[] args) {
            logger.setLevel(Level.FINER);
            logger.setUseParentHandlers(false);
            Handler manejador = new ConsoleHandler();
            manejador.setLevel(Level.FINER);
            logger.addHandler(manejador);
            logger.entering(Main.class.getName(), "main");
            logger.exiting(Main.class.getName(),"main");
    }
}
```

Aserciones

- Expresiones de uso frecuente para hacer programación defensiva.
- Solo deben emplearse para hallar errores internos del programa durante las pruebas.
- El mecanismo de aserciones permite insertar pruebas durante la comprobación y hacer que se eliminen automaticamente en el código de produccion.
- Ejemplos:

  assert condición; assert x>=0;

  assert condición : expresión; assert x>=0 : x;
- Ambas sentencias evaluan la condición y lanzan un AssertionError si resulta ser false.
- En la segunda sentencia, la expresión se le pasa al constructor del objeto AssertionError y se transforma en una cadena de mensaje.
- Las aserciones por default estan desactivadas. Se activan ejecutando el programa con las opciones (argumentos de la VM):
  - -enableassertions o
  - -ea
- Y se desactivan:
  - -disableassertions o
  - -da

Aserciones

- Ejemplo

```
public class MainAsserciones {

    public static void main(String[] args) {
        int x = -1;
        assert x>0:x;
    }

}
```

Exception in thread "main" java.lang.AssertionError: -1
at MainAsserciones.main(MainAsserciones.java:5)

Depurador de Eclipse

- Se pueden fijar puntos de ruptura.
  - Sobre la línea deseada <CTRL+SHIFT+B>
- Empezar a depurar
  - Run → Debug As → Java Application
- Inspeccionar variables.
  - Cuando el depurador se detiene en un punto de ruptura, se puede ver la pila de llamadas y las variables locales.
- Avanzar paso a paso por un programa
  - Step into (F5)
  - Step over (F6)

Excepciones y Archivos - Ejercicios

- **Ejercicio 1:** Cree una clase con un método **main()** que genere un objeto de la clase **Exception** dentro de un bloque **try**. Proporcione al constructor de **Exception** un argumento **String**. Capture la excepción dentro de una cláusula **catch** e imprima el argumento **String**. Añada una clausula **finally** e imprima un mensaje para demostrar que pasó por allí.
- **Ejercicio 2**: Defina una referencia a un objeto e inicializela a **null**. Trate de invocar un método a través de esta referencia. Ahora rodee el código con una clausula **try-catch** para probar la nueva excepción.
- **Ejercicio 3**: Escriba código para generar y capturar una excepción **ArrayIndexOutOfBoundsException (**Índice de matriz fuera de límites).
- **Ejercicio 4**: Cree su propia clase de excepción utilizando la palabra clave **extends**. Escriba un constructor para dicha clase que tome un argumento **String** y lo almacene dentro del objeto como una referencia de tipo **String**. Escriba un método que muestre la cadena de caracteres almacenada. Cree una cláusula **try-catch** para probar la nueva excepción.
- **Ejercicio 5**: Defina un comportamiento de tipo reanudación utilizando un bucle **while** que se repita hasta que se deje de generar una excepción.

Excepciones y Archivos - Ejercicios

- **Ejercicio 6:** Implementar un simulador de Vehículos. Existen dos tipos de Vehículos: Coche y Camión. Sus características comunes son la matricula y la velocidad. En el momento de crearlos, la matricula se recibe por parámetro y la velocidad se inicializa a cero. El método toString() de los vehículos devuelve información acerca de la matricula y la velocidad. Además se pueden acelerar, pasando por parámetro la cantidad en km/h que se tiene que acelerar.
- Los coches tienen además un atributo para el número de puertas, que se recibe también por parámetro en el momento de crearlo. Tiene además un método que devuelve el número de puertas (este comportamiento esta descripto en la interfaz Puertable).
- Los camiones tienen un atributo de tipo Remolque que inicializa a null (para indicar que no tiene remolque). Además tiene un método porRemolque(), que recibe el Remolque por parámetro, y otro quita Remolque(). Cuando se muestre la información de un camión que lleve remolque, además de la matricula y la velocidad del camión, debe aparecer la información del remolque.
- En esta clase hay que sobrescribir el método acelerar de manera que si el camión tiene remolque y la velocidad mas las aceleración superan los 100 km/h se lance un excepción de tipo DemasiadoRapidoException.
- Hay que implementar la clase Remolque. Esta clase tiene un atributo de tipo entero que es el peso y cuyo valor se le da en el momento de crear el objeto. Debe tener un método toString() que devuelva la información del remolque.
- Utilizando esta implementación, desarrolle una aplicación que haga lo siguiente:
- Declare y cree un objeto de la clase ArrayList con 4 vehículos (2 camiones y 2 coches)
- Suponiendo que no se sabe en que posición del vector están los coches y los camiones: Ponga un remolque de 5000 Kg. a los camiones. Acelere todos los vehículos y escriba por pantalla la información de todos ellos.

Excepciones y Archivos - Ejercicios

- **Ejercicio 7:** Un banco contiene las Cuentas de sus clientes. Las CuentasDeAhorro no pueden tener números rojos. Las CuentasCorrientes pueden tener una CuentaDeAhorro asociada, de forma que si se intenta retirar más dinero del saldo actual, se debe retirar el dinero que falte de la CuentaDeAhorro asociada.
  - a) Define Cuenta de forma que no pueda instanciarse. De toda Cuenta se debe poder ingresar y retirar dinero, preguntar por el saldo, por el DNI del titular y debe tener un método toString de devuelva al menos el saldo y el DNI del titular.
  - b) Implementa las clases CuentaCorriente y CuentaDeAhorro.
  - c) Crea una especialización CuentaDeAhorroEsp de CuentaDeAhorro en la que se añade un entero penalización, de forma que se penaliza la retirada con una penalización % del dinero retirado. Sobrescribe sólo los métodos necesarios (incluyendo constructor y toString).
  - d) Si el saldo de CuentaDeAhorro fuese a quedar negativo, antes de debe lanzar una excepción SaldoNegativo (que hereda de Exception).
  - e) Implementa la clase Banco que contiene un array polimórfico de Cuentas, incluyendo el constructor que consideres más apropiado.
  - f) Incluye el método totalSaldoMaxPenalización en la clase Banco que devuelva la suma de los saldos de todas las cuentas corrientes y la máxima penalización entre las CuentaDeAhorroEsp .
- Crea una clase con método main en la que instanciamos un Banco con 100 Cuentas, nos creamos una CuentaCorriente con 500 euros y una cuenta de ahorro especial con una penalización del 5%. Finalmente debe mostrar información de todas las Cuentas del Banco. En un bloque try & catch retirar dinero de una cuenta de Ahorro.

Excepciones y Archivos - Ejercicios

- **Ejercicio 8:** Crear un archivo de texto con el nombre y contenido que quiera. Crear una aplicación que lea el archivo de texto y muestre su contenido por pantalla sin espacios. Por ejemplo, si el archivo tiene el siguiente texto “Esto es una prueba”, deberá mostrar “Estoesunaprueba”. Capturar las excepciones que sean necesarias.
- **Ejercicio 9:** Crear una aplicación donde se solicite la ruta de un archivo por teclado y un texto que se desea escribir en el archivo. Se deberá mostrar por pantalla el mismo texto pero variando entre mayúsculas y minúsculas, es decir, si se escribe “Bienvenido” deberá devolver “bIENVENIDO”. Si se escribe cualquier otro carácter, se quedara tal y como se escribió. Se deberá crear un método para escribir en el archivo el texto introducido y otro para mostrar el contenido en mayúsculas.
- **Ejercicio 10:** Crear una aplicación que solicite la ruta de dos archivos de texto y de una ruta de destino (solo la ruta, sin archivo al final). Se deberá copiar el contenido de los dos archivos en uno, este tendrá el nombre de los dos archivos separados por un guión bajo, y se guardará en la ruta donde se le haya indicado por teclado. Para unir los archivos en uno, crear un método donde se le pase como parámetro todas las rutas. En este método, además de copiar debe comprobar que si existe el archivo de destino, muestre un mensaje informando de si se quiere sobrescribir el archivo. Por ejemplo, si se tiene un archivo A.txt con “ABC” como contenido, un archivo B.txt con “DEF” y una ruta de destino D:\, el resultado será un archivo llamado A_B.txt en la ruta D:\ con el contenido “ABCDEF”.

JDBC

- **Introducción**
  - **JDBC = Java DataBase Connectivity**
  - JDBC es una API de Java para ejecutar sentencias SQL. Clases e Interfaces escritas en Java.
  - Representan conexiones a BD, sentencias SQL, cursores y otros elementos.
  - Permite escribir aplicaciones de Bases de Datos utilizando Java.
  - Es implementada vía un **Driver Manager** que puede soportar varios drivers conectándose a varias bases de datos.

JDBC

- Entornos soportados:
  - JDBC está implementada enteramente en Java.
  - Por lo tanto, puede correr en cualquier plataforma en la que se pueda instalar el JDK.
  - Usar JDBC requiere el uso de uno o más JDBC Drivers.
  - Cada driver tiene sus propias restricciones de base de datos y plataforma.

JDBC

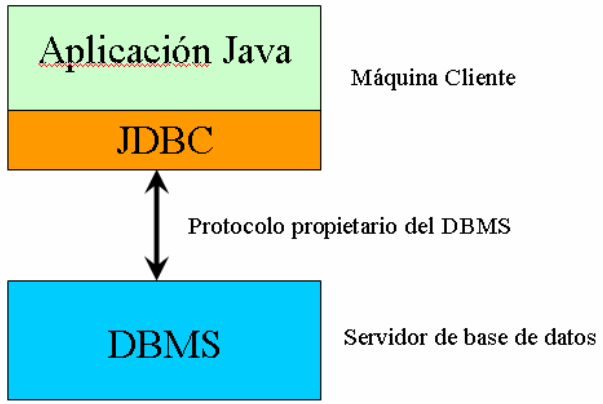
- Que se puede hacer con JDBC?
  - Tres etapas:
    - Establecer una conexión con la base de datos.
    - Enviarle sentencias SQL.
    - Procesar los resultados.

JDBC

- JDBC vs. ODBC
  - **ODBC** (Open DataBase Connectivity) es una interfaz muy utilizada para acceso a bases de datos relacionales.
  - Provee la posibilidad de conectarse con casi todas las bases de datos en casi todas las plataformas.
  - Entonces, ¿porque no usar ODBC desde java?
  - La respuesta es que se puede utilizar ODBC desde Java, pero esto se puede hacer mejor con la ayuda de JDBC en la forma del JDBC-ODBC Bridge.

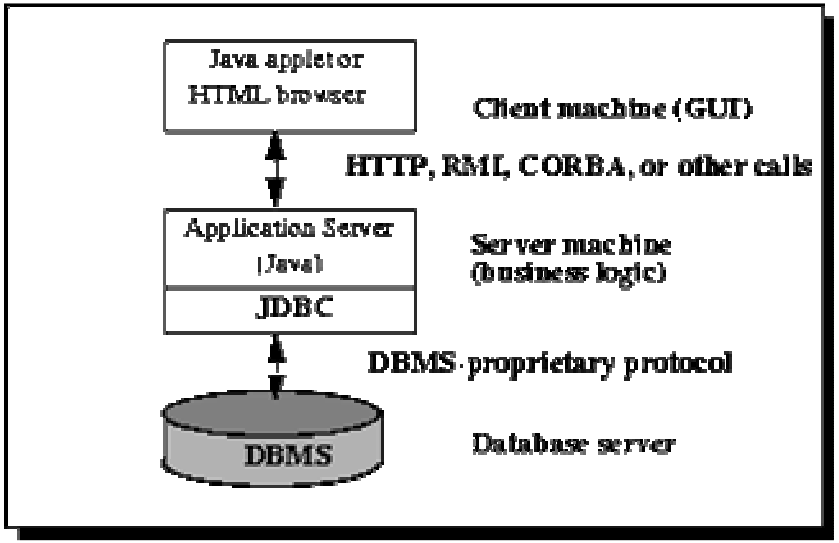
JDBC

- Modelo de acceso Two Tier
  - La aplicación se comunica directamente con la BD.
  - Requiere que un driver JDBC se comunique con el DBMS particular de la BD que se está accediendo.
  - Las sentencias SQL del usuario se envían a la BD y los resultados son enviados de vuelta al usuario.



JDBC

- Modelo de acceso Tree Tier
  - Separa:
    - Presentación visual (cliente)
    - Logica (negocio)
    - Datos puros (base de datos)



JDBC

- Implementación
  - JDBC esta implementada en el package *java.sql*.
  - Este paquete contiene un conjunto de clases e interfaces relacionadas con la manipulación de bases de datos.
  - Pasos:
    - Establecimiento de una conexión
      - Cargar el driver apropiado (se registra manualmente un controlador):
        - Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
      - Hacer la conexión propiamente dicha:
        - String url = "jdbc:odbc:BD"; //protocolo:subprocolo:DSN
        - Connection con = DriverManager.getConection(url,"myLogin","myPassword");
      - Obtener la conexión:
        - *DriverManager.getConnection().*
    - Crear sentencias JDBC:
      - Statement stmt = con.createStatement();
    - Ejecutar sentencia
      - Sentencias select → executeQuery(String):ResultSet [conjunto de filas]
      - Sentencias update → executeUpdate(String): int [cant. filas afectadas]
      - ResultSet rs = stmt. *executeQuery("SELECT * FROM tabla");*
    - Procesar resultado:
      - rs.next();
      - rs.getString(1);

JDBC

- Ejemplo:

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver")
Connection con =
   DriverManager.getConnection("jdbc:odbc:BD");
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("SELECT * FROM
   tabla");
while (rs.next())
   System.out.println(rs.getString(1));
rs.close();
con.close();
stmt.close();
```

JDBC

- Sentencias SQLs (Lenguaje de Consulta Estructurado)

```
SELECT campo
FROM tabla
[WHERE campo operador valor]
```

```
UPDATE tabla
SET campo = valor
[WHERE campo operador valor]
```

```
DELETE FROM tabla
[WHERE campo operador valor]
```

```
INSERT INTO tabla
VALUES (valor,
   ...)
```

JDBC

- Características avanzadas
  - La API JDBC de Java provee características más avanzadas:
    - Prepared statements
    - Stored procedures
    - Transacciones
  - Warnings y Excepciones

JDBC

- Prepared Statement
  - Los objetos *PreparedStatement* toman parámetros.
  - La ventaja reside en que se utiliza la misma sentencia pero se le proveen diferentes valores cada vez que se la ejecuta.

```
String sql =“SELECT * FROM tabla WHERE campo = ?”;
PreparedStatement pstmt =con.prepareStatement(sql);
pstmt.setString(1, “50”);
ResultSet rs = pstmt.executeQuery();

ParameterMetaData pmd =
   pstmt.getParameterMetaData();
int pcount = pmd.getParameterCount();
```

JDBC

- Metadatos:
  - Información sobre la estructura de una base de datos y de sus tablas.

```
DatabaseMetaData md = con.getMetaData();
md.getDatabaseProductName();
md.getMaxConnections();
```

```
ResultSet rs = …
ResultSetMetaData md = rs.getMetaData();
md.getColumnCount();
md.getColumnLabel(1);
md.getColumnDisplaySize(1);
```

JDBC

- Transacciones:
  - Es posible agrupar un conjunto de sentencias para formar una transacción.
  - La transacción se puede confirmar (OK) o deshacer (error)
  - Razón principal → integridad de la base de datos.
  - Por default, las conexiones a la base de datos se hacen en modo de confirmación automática, y cada orden de SQL se confirma en la base de datos en cuanto se ejecute.
  - Se puede desactivar el método de confirmación automática

JDBC

- Transacciones, ejemplo:

```
con.setAutoCommit(false);
Statement stmt = con.createStatement();
stmt.executeUpdate(orden1);
stmt.executeUpdate(orden2);
…
con.commit();
```

- Sin embargo, si se ha producido un error, se hace la llamada:

```
con.rollback();
```

JDBC - Ejercicios

- **Ejercicio 1:** Crear la siguiente base de datos de “Libros”

Autores	1	∞	ISBNAutor	∞	1	Titulos
idAutor nombre apellido			idAutor ISBN			ISBN titulo nroEdicion copyright editorial

- **Ejercicio 2:** Defina una aplicación de consulta completa para la base de datos “Libros”. Proporcione las siguientes consultas:
  - Seleccionar todos los autores de la tabla Autores.
  - Seleccionar un autor específico y mostrar todos los libros de ese autor. Incluir titulo, año e ISBN.
  - Seleccionar una editorial específica y mostrar todos los libros publicados por esa editorial. Incluir titulo, año e ISBN. Ordenar alfabéticamente por titulo.
- **Ejercicio 3:** Defina una aplicación de manipulación de base de datos para la base de datos “Libros”. El usuario puede editar los datos existentes y agregar nuevos datos a la base de datos.
  - Agregar un nuevo autor.
  - Editar la información existente para un autor.
  - Agregar un nuevo titulo para un autor.

Swing

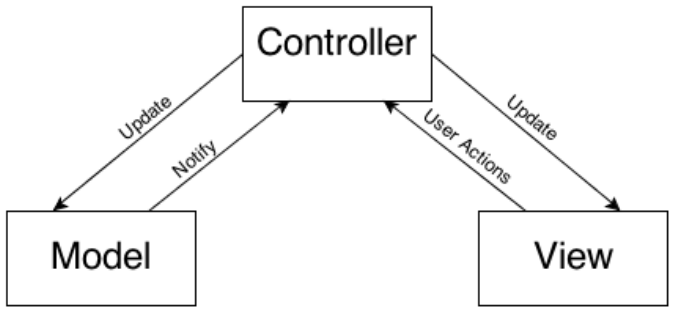
- Programas en Java que emplean una interfaz gráfica de usuario (GUI)
- Frameworks:
  - AWT (Abstract Window Toolkit)
    - Introducido por Sun en versión 1.0
    - Era una herramienta poderosa que impulsó la popularidad de Java.
    - Implementación limitada (IU poco seria y básica).
    - Muchos bugs y consume gran cantidad de recursos.
    - Cada componente AWT obtiene su propia ventana de la plataforma. Aborda los elementos de a GUI delegando su creación y comportamiento al juego de herramientas GUI nativo de cada plataforma.

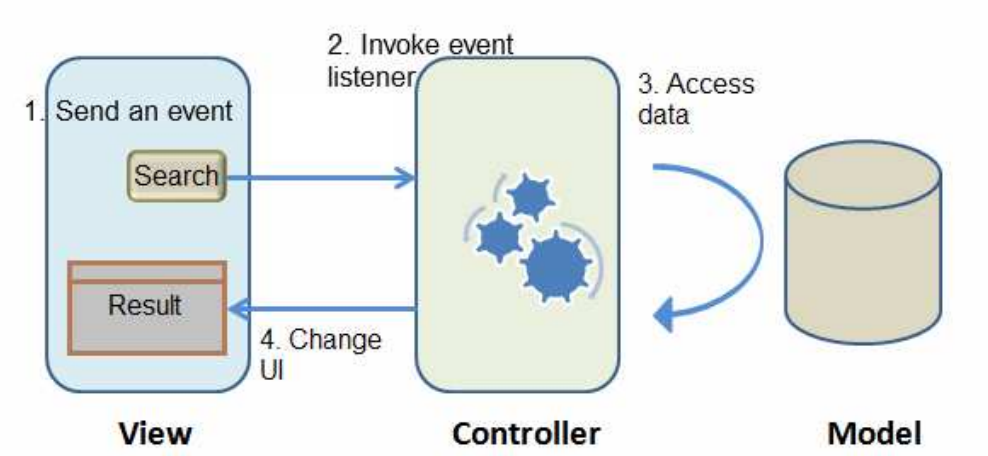
Swing

- Frameworks:
  - SWING
    - Introducido por Sun y Netscape en 1.2
    - Componentes swing como parte de las Java Foundation Classes (Swing, Cortar y Pegar, Colores del escritorio, Java 2D, Impresión)
    - Avances significativos:
      - Pocos recursos.
      - Componentes mas sofisticados (mas potentes).
      - Construir la apariencia de los programas.
    - Los componentes swing se dibujan en su contenedor, la apariencia será igual en todas las plataformas (poca dependencia de la plataforma)
    - Arquitectura Modelo – Vista – Controlador (Patrón: MVC): Es el corazón de la programación de componentes UI de swing.
      - Modelo → almacena los datos (contenido)
      - Vista → representación en pantalla del componente (color, tamaño)
      - Controlador → gestiona la entrada, como los clicks del mouse
    - No hacer que un objeto sea responsable de un número excesivo de cosas.

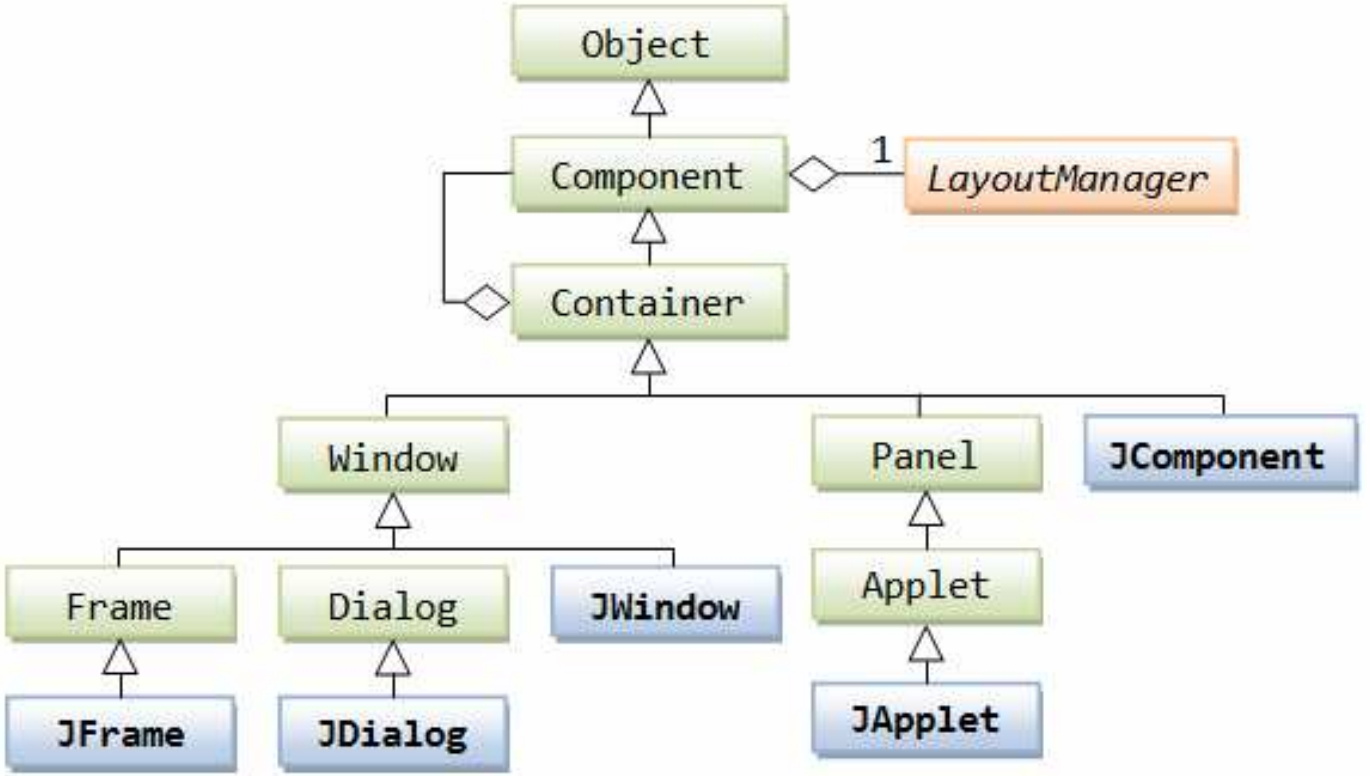
Swing - MVC

- Frameworks:
  - SWING
    - Patrón MVC





Swing - Jerarquía



Swing - JFrame

- Creación de una ventana
  - Marco: ventanas de nivel superior (las que no están dentro de otra ventana)
  - JFrame (extiende de java.awt.Frame)
  - Se encuentra en el paquete javax.swing
  - Constructores:
    - JFrame()
    - JFrame(String unTitulo)
    - JFrame(GraphicsConfiguration)
    - JFrame(String, GraphicsConfiguration)

Swing - JFrame

- Ejemplo:

```
import javax.swing.*;
public class PruebaMarco {
   public static void main(String [] args) {
       Marco marco = new Marco();
       marco.setDefaultCloseOperation
             (JFrame.EXIT_ON_CLOSE);
       marco.setVisible(true);
   }
}
```

Swing - JFrame

```
import javax.swing.*;
public class Marco extends JFrame {
   public static final int ANCHURA = 300;
   public static final int ALTURA  = 200;
   public Marco(){
       setSize(ANCHURA, ALTURA);
   }
}
```

- Tamaño por defecto: 0 x 0 pixeles.
- Los marcos se crean invisibles, posibilitando de agregar componentes al marco antes de mostrarlo por primera vez.
- Al mostrar el marco se activa un hilo de la GUI que mantiene vivo al programa.
- Se pueden desactivar todos los adornos del marco
  marco.setUndecorated(true);

Swing - JFrame

- Colocación de un marco
  - marco.dispose() // cierra la ventana
  - marco.setIconImage(Toolkit.getDefaultToolkit().getImage(“icon.gif”)); // agrega un icono
  - marco.setTitle(“UnTitulo”); // texto en la barra de títulos
  - marco.setResizable(false);   // determina si el usuario va a poder cambiar el tamaño de la ventana
  - marco.setLocation(50, 70) // traslada el componente a a una nueva ubicación desde la esquina superior izquierda del contenedor o de la pantalla en el caso que sea un marco (x=50 y=70)
  - marco.setLocation(new Point(50, 70));
  - marco.getLocation():Point //respecto del contenedor
  - marco.getLocationOnScreen():Point // respecto de la pantalla
  - marco.setExtendedState(Frame.MAXIMIZED_BOTH);

Swing - JFrame

```
import javax.swing.*;
public class MarcoCentrado extends JFrame {
   public MarcoCentrado(){
       Toolkit kit = Toolkit.getDefaultToolkit();
       Dimension tamPan = kit.getScreenSize();
       int altPan   = tamPan.height;
       int anchoPan = tamPan.width;
       setSize(anchoPan/2, altPan/2);
       setLocation(anchoPan/4, altPan/4);
       setTitle("Marco centrado");
   }
}
```

Swing - JPanel

- Visualización de información en una lámina (JPanel)
  - Los marcos están diseñados para ser contenedores de componentes.
  - Se dibuja en un componente llamado panel, que después se añade al marco.
  - Los paneles (JPanel) posee dos propiedades útiles:
    - Tienen una superficie en la que se puede dibujar
    - Son, a su vez, contenedores.

```
JFrame marco =  new JFrame();
Panel panel = new JPanel();
panel.add(new JButton("OK")); //contenedor
marco.add(panel);
```

Swing - JPanel

- Panel dibujable:

```
import javax.swing.*;
public class PanelDibujable extends JPanel {
   public void paintComponent(Graphics g){
       super.paintComponent();
       g.drawString("Hola", 75, 100);
   }
}
```

- Formas 2D

```
((Graphics2D) g).draw(new
   Rectangle2D.Double(10.0, 25.0, 22.5, 20.0));
```

Swing - JPanel

- Uso del color:

```
JPanel p = new JPanel();
p.setBackground(Color.BLUE);
p.setBackground(new Color(0,0,255)); //Modelo RGB

Color c = new Color(255,0,0);
c.brighter().brighter(); // brillante
c.darker().darker();     // apagado
```

```
public void paintComponent(Graphics g){
  g.setPaint(new Color(0,128,128));
  g.drawString("Bienvenido", 75, 125);
}
```

Swing - Font

- Uso de fuentes:
  - Las fuentes se construyen con:
    - Tipo de fuente
    - Estilo (Font.BOLD, Font.ITALIC, Font.PLAIN)
    - Tamaño en pixeles.

```
Font   f = new Font(“Arial”, Font.ITALIC, 16);
Button b = new JButton(“OK”);
b.setFont(f);
```

```
public void paintComponent(Graphics g){
  g.setFont(new Font(“Helvetica”,Font.BOLD,14));
  g.drawString(“Bienvenido”, 75, 125);
}
```

Swing - Border

- Uso de bordes:

```
Border grabado =
   BorderFactory.createEtchedBorder();
Border conTitulo =
   BorderFactory.createTitleBorder(grabado, “El
   título”);
panel.setBorder(conTitulo);

BorderFactory.createLineBorder(Color,int grosor);
BorderFactory.createEmptyBorder();
. . .
```

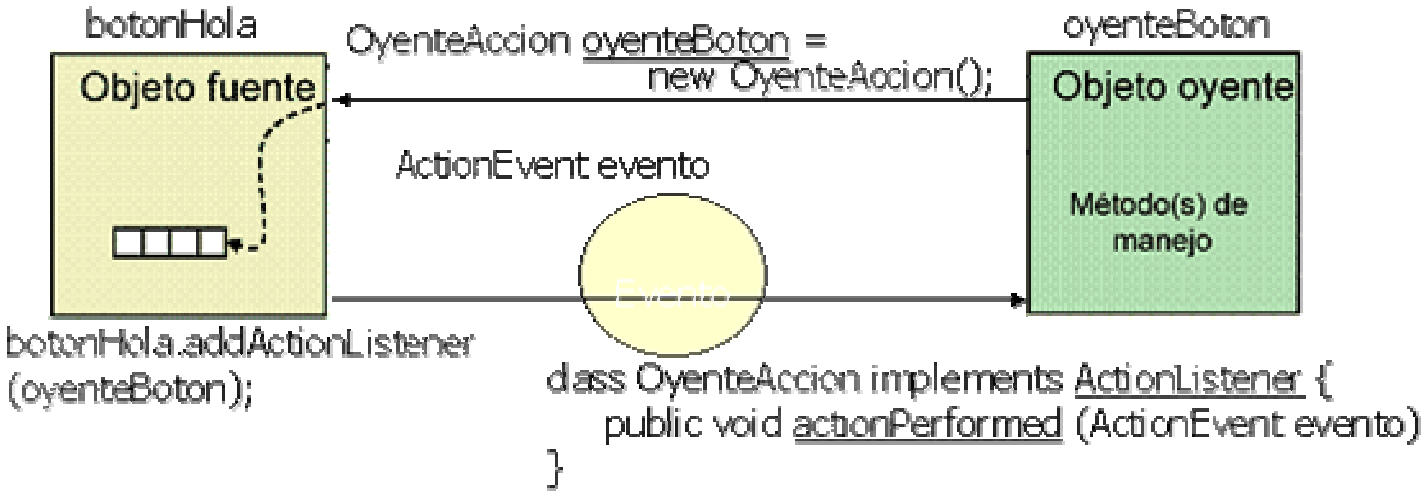
Swing - Evento

- Todo entorno operativo que admita GUI monitoriza constantemente los eventos como teclas pulsadas o clicks de mouse.
- El entorno operativo notifica estos eventos a los programas que están en ejecución.
- Funcionamiento:
  - Un objeto oyente es una instancia de una clase que implementa una interfaz de oyente.
  - Una fuente de eventos es un objeto en que se puede registrar objetos oyentes y enviar a esos objetos otros objetos de eventos.
  - La fuente de eventos envía objetos de evento a todos los oyentes registrados en ella cuando se produce un evento.
  - Los objetos oyentes utilizarán entonces la información contenida en el objeto de evento para determinar su reacción frente al evento.

```
objetoFuenteDeEventos.addEventListener(objetoOyenteDeEventos);
```

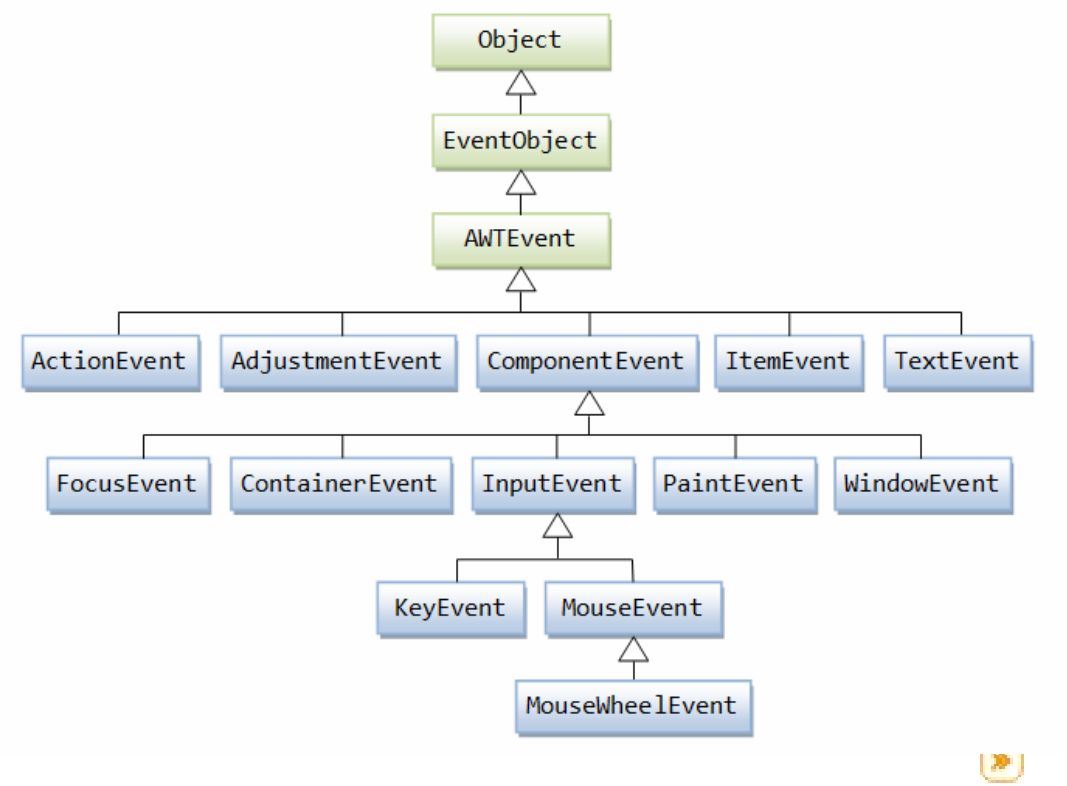
Swing - Evento

- Funcionamiento:



Swing - Evento

- Tipos de eventos:
  - Semánticos: Expresan lo que está haciendo el usuario. Ejemplo: evento de ítem.
  - Bajo nivel: Aquellos que hacen posible lo anterior. Ejemplo: evento de mouse.



Swing - Evento

INTERFAZ	MÉTODOS	PARAMETRO	GENERADO POR
ActionListener	actionPerformed	ActionEvent	AbstractButton JComboBox JTextField
AdjustmentListener	adjustmentValueChanged	AdjustmentEvent	JScrollBar
ItemListener	itemStateChanged	ItemEvent	AbstractButon JComboBox
FocusListener	focusGained focusLost	FocusEvent	Component
KeyListener	keyPressed keyReleased keyTyped	KeyEvent	Component
MouseListener	mousePressed mouseReleased mouseEntered mouseExited mouseClicked	MouseEvent	Component

Swing - Evento

INTERFAZ	MÉTODOS	PARAMETRO	GENERADO POR
MouseMotionListener	mouseDragged mouseMoved	MouseEvent	Component
MouseWheelListener	mouseWheelMoved	MouseWheelEvent	Component
WindowListener	windowClosing windowOpened windowIconified windowDeiconified windowClosed windowActivated widowDeactivated	WindowEvent	Window
WindowFocusListener	windowGainedFocus windowLostFocus	WindowEvent	Window
WindowStateListener	windowStateChanged	WindowEvent	Window
DocumentListener	insertUpdate removeUpdate changedUpdate	DocumentEvent	JTextField JTextArea

Swing - Evento

- Formas de manejar eventos:

```
new Button(“OK”).addActionListener(new
   MiOyente());
public class MiOyente implements ActionListener {
   public void actionPerformed(ActionEvent e) {}
}
```

```
new Button(“OK”).addActionListener(this);
```

```
new Button(“OK”).addActionListener(
   new ActionListener() {
    public void actionPerformed(ActionEvent e){
    }} );
```

Swing - Layout

- Layouts
  - Todos los componentes del contenedor son situados por un encargado de disposiciones.
  - Tipos:
    - FlowLayout
    - BorderLayout
    - GridLayout
    - BoxLayout
    - GridBagLayout

Swing - Layout

- FlowLayout
  - Encargado de disposición de flujo.
  - Layout predeterminado para un panel.
  - Alinea los componentes horizontalmente hasta que no quede espacio y después hace que empiece una nueva fina de componentes.
  - De forma predeterminada, los componentes se centran en el contenedor (FlowLayout.CENTER)

```
panel.setLayout(new FlowLayout());
panel.setLayout(new FlowLayout(FlowLayout.LEFT));
panel.setLayout(new FlowLayout(FlowLayout.RIGHT,
  10,   //espaciado horizontal en pixeles
  20)); //espaciado vertical en pixeles
```

Swing - Layout

- BorderLayout
  - Encargado de disposición de borde.
  - Layout predeterminado para un JFrame.
  - Permite seleccionar el lugar en que deseamos colocar cada componente (norte, centro, sur, este u oeste).

```
panel.setLayout(new BorderLayout());
panel.setLayout(new BorderLayout(
      10,   //espaciado horizontal en pixeles
      20)); //espaciado vertical en pixeles
panel.add(new JButton(“OK”), “South”);
panel.add(new JButton(“OK”), BorderLayout.SOUTH);
```

Swing - Layout

- GridLayout
  - Encargado de disposición de cuadrícula.
  - Organiza todos los componentes en filas y columnas.
  - Las celdas son siempre del mismo tamaño.

```
panel.setLayout(new GridLayout(5, 4));
panel.setLayout(new GridLayout(5, 4,
      10,   //espaciado horizontal en pixeles
      20)); //espaciado vertical en pixeles
panel.add(new JButton("OK"));
panel.add(new JButton("Cancel"));
```

Swing - Layout

- BoxLayout
  - Encargado de disposición de caja.
  - Permite ubicar una sola fila o columna de componentes con mas flexibilidad que GridLayout.
  - Box es un contenedor cuyo encargado de disposición predeterminada es una BoyLayout.

```
Box b = Box.createHorizontalBox(); //o
Box b = Box.createVerticalBox();
b.add(new JButton(“OK”));
b.add(new JButton(“Cancel”));
```

Swing - Layout

- BoxLayout
  - No tiene la noción de espacios entre componentes. Para separar las componentes se agregan unos rellenos invisibles.
  - Rellenos:
    - Puntales:

      b.add(new JLabel(“rotulo”));

      b.add(Box.createHorizontalStrut(10));

      b.add(new JTextField());
    - Zonas rígidas:

      b.add(Box.createRigidArea(new Dimension(5,20)));
    - Cola: las componentes se separan lo mas posible.

      b.add(new JButton(“OK”));

      b.add(Box.createGlue());

      b.add(new JButton(“Cancel”));

Swing - Layout

- GridBagLayout
  - Encargado de disposición de cuadricula flexible.
  - Filas y columnas pueden tener tamaños diferentes (variables)
  - Se pueden unir celdas adyacentes.

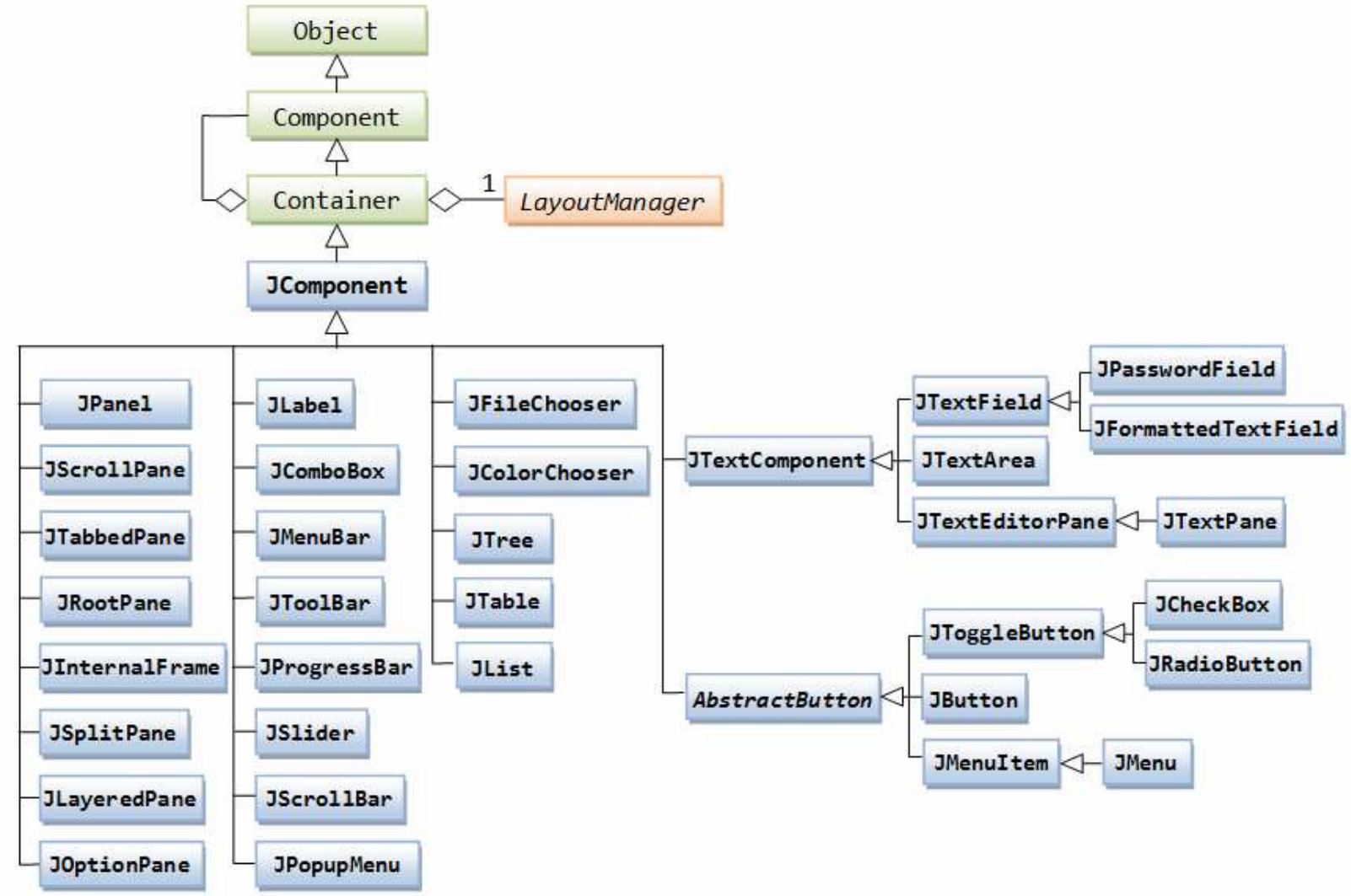
```
GridBagLayout disp =  new GridBagLayout();
panel.setLayout(disp);
GridBagConstraints restr = new
  GridBagConstraints();
restr.weightx = 100;
restr.weighty = 100;
restr.gridx = 0;
restr.gridy = 2;
```

Swing - Layout

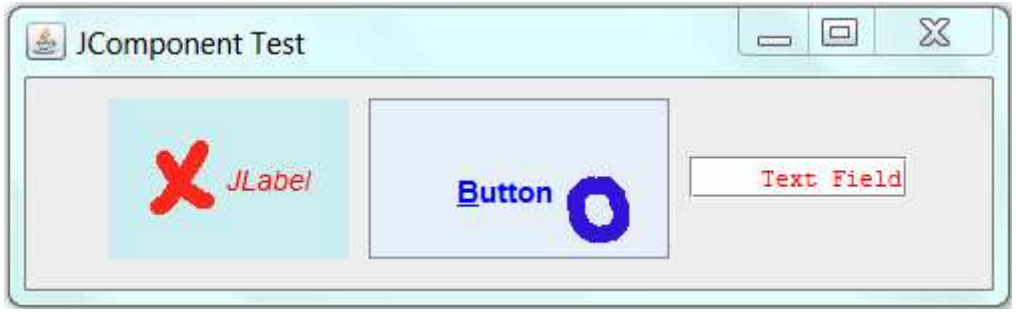
- GridBagLayout

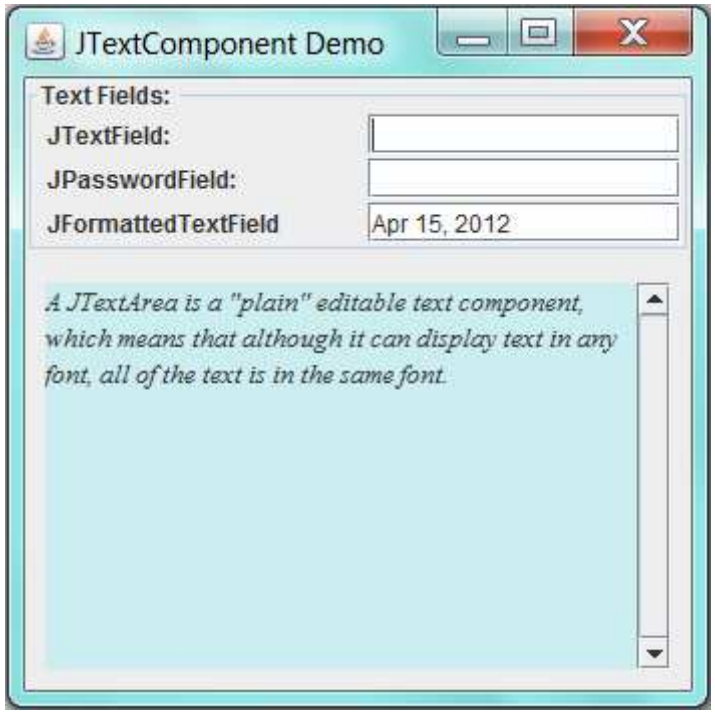
```
restr.gridwidth = 2; //abarca 2 columnas
restr.gridheight= 1; //abarca 1 fila
panel.add(estilo, restric);
```

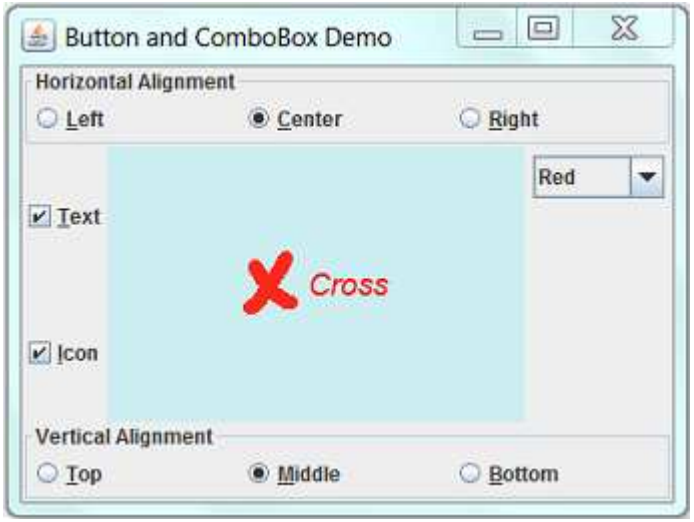
Swing - Componentes



Swing - Componentes básicas







Swing - Componentes básicas

- JButton

```
JButton b1 = new JButton();
JButton b2 = new JButton(“OK”);
JButton b3 = new JButton(new ImageIcon(...));
JButton b4 = new JButton(“OK”, new
  ImageIcon(...));
b1.setText(“OK”);
b1.setToolTipText(“Esto es un botón”);
b1.setEnabled(true);
b1.setVisible(false);
b1.setSize(new Dimension(20,50));
b1.setBackground(Color.RED);
b1.setBorder(BorderFactory.createLineBorder(new
  Color(255,0,0)));
```

Swing - Componentes básicas

- JTextField

```
JTextField t1 = new JTextField();
JTextField t2 = new JTextField(“Introd txt”);
JTextField t3 = new JTextField(20); //anchura
JTextField t4 = new JTextField(“Introd txt”,20);
t1.setText(“Pepe”);
t1.setEditable(true);
t1.setEnabled(false);
t1.setColumns(10);
t1.setFont(new Font(“Arial”, Font.BOLD, 20));
t1.getText();
T1.getDocument().addDocumentListener(DocumentList
  ener)
```

Swing - Componentes básicas

- JLabel

```
JLabel l1 = new JLabel();
JLabel l2 = new JLabel("Prom:");
JLabel l3 = new JLabel("Prom:", JLabel.RIGHT);
JLabel l4 = new JLabel(new ImageIcon("xxx.gif"));
JLabel l5 = new JLabel("Prom:", new
  ImageIcon("xxx.gif"), JLabel.LEFT);
JLabel l6 = new
  JLabel("<HTML>Prom<b>:</b></HTML>");
l1.setText("Prom:");
l1.setIcon(new ImageIcon("xxx.gif"));
l1.setOpaque(true);
l1.getText();
```

Swing - Componentes básicas

- JPasswordField
  - Los caracteres introducidos por el usuario no se llegan a visualizar. Emplea mismo modelo que los JTextField pero distinta vista.

```
JPasswordField p1 = new JPasswordField();
JPasswordField p2 = new JPasswordField(20);
JPasswordField p3 = new JPasswordField("Clave");
JPasswordField p4 = new JPasswordField("Clave",
  20);
p1.setEchoChar('*');
char [] textoClave = p1.getPassword();
```

Swing - Componentes básicas

- JTextArea

```
JTextArea a1 = new JTextArea();
JTextArea a2 = new JTextArea("Escriba txt");
JTextArea a3 = new JTextArea(10, 20);
JTextArea a4 = new JTextArea("Escriba txt", 10,
  20);
a1.setLineWrap(true); //lineas largas pasan a la
  siguiente.
a1.setWrapStyleWord(true); //salta por palabra
a1.append("Otro texto");
a1.insert("Otro texto", 2);
new JScrollPane(a1);
```

Swing - Componentes básicas

- Formateadores
  - NumberFormat.getIntegerInstance()
  - NumberFormat.getCurrencyInstance()
  - NumberFormat.getPercentInstance()
  - DateFormat.getDateInstance()
  - DateFormat.getTimeInstance()
  - DateFormat.getDateTimeInstance()

```
JFormattedTextField c = new
   JFormattedTextField(NumberFormat.getIntegerInst
   ance());
c.setColumns(6);
c.setValue(new Integer(100));
((Number) c.getValue()).intValue();
```

Swing - Componentes básicas

- Componentes de selección
  - Casillas de Verificación
  - Botones de radio
  - Listas de opciones
  - Controles deslizantes
  - Controles giratorios

Swing - Componentes básicas

JCheckBox

```
JCheckBox c1 = new JCheckBox();
JCheckBox c1 = new JCheckBox(new
   ImageIcon("xxx.gif"));
JCheckBox c2 = new JCheckBox(new
   ImageIcon("xxx.gif"), true);
JCheckBox c3 = new JCheckBox("Opc 1");
JCheckBox c4 = new JCheckBox("Opc 1", true);
JCheckBox c5 = new JCheckBox("Opc 1", new
   ImageIcon("xxx.gif"));
JCheckBox c5 = new JCheckBox("Opc 1", new
   ImageIcon("xxx.gif"), true);
C1.setSelected(true);
C1.isSelected();
```

Swing - Componentes básicas

- JRadioButton

```
ButtonGroup g = new ButtonGroup();
JRadioButton r1 = new JRadioButton(“r1”, false);
JRadioButton r2 = new JRadioButtton(“r2”, true);
g.add(b1);
g.add(b2);
b1.setSelected(true);
b1.isSelected();
b1.addActionListener(oyente);
B2.addActionListener(oyente);
```

Swing - Componentes básicas

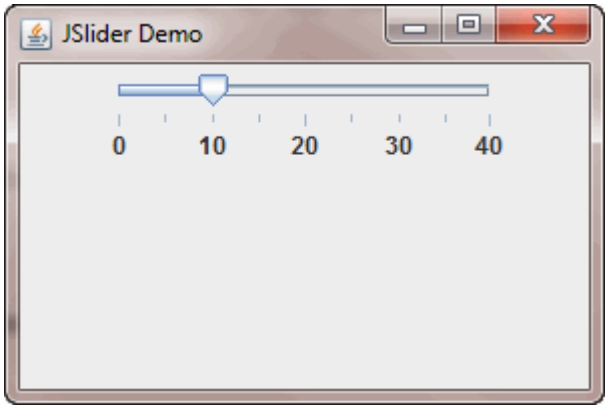
- JComboBox
  - Combina la flexibilidad de un campo de texto con un conjunto de opciones predeterminadas.

```
JComboBox c = new JComboBox();
c.setEditable(true);
c.addItem(“opc1”);
c.addItem(“opc2”);
c.insertItemAt(“opc0”, 0);
c.removeItem(“opc1”);
c.removeItemAt(0);
c.removeAllItems();
c.setSelectedItem(“opc2”);
c.getSelectedItem();
c.getSelectedIndex();
c.getItemCount();
c.addActionListener(oyente);
```

Swing - Componentes básicas

- JSlider (control deslizante)
  - Seleccionar una opción dentro de un espectro continuo de valores.

```
JSlider s1 = new JSlider(); //orientacion: HORIZONTAL, min:0,
   max:100, inicial:50
JSlider s2 = new JSlider(SwingConstants.VERTICAL);
JSlider s3 = new JSlider(10, 110);
JSlider s4 = new JSlider(10, 110, 60);
JSlider s5 = new JSlider(SwingConstants.VERTICAL, 10, 110,
   60);
s1.setValue(0); S1.getValue();
s1.setMajorTickSpacing(20);
s1.setMinorTickSpacing(5);
s1.setPaintTicks(true);
s1.setPaintLabels(true);
```



Swing - Componentes básicas

- JSpinner (control giratorio)

```
JSpinner s = new JSpinner();
int valor = (Integer) s.getValue();
JSpinner s = new JSpinner(new
  SpinnerNumberModel(5, 0, 10, 0.5)); //inicial:
  5, intervalo:0-10, incremento:0.5

String [] fuentes =
  GraphicsEnvironmet.getLocalGraphicsEnvironment(
  ).getAvailableFontFamilyNames();
JSpinner s =  new JSpinner(new
  SpinnerListModel(fuentes));
JSpinner s = new JSpinner(new
  SpinnerDateModel());
```

Swing - Menú

Barra de Menú

```
JMenuBar bm = new JMenuBar();
JMenu mEdicion = new JMenu("Edicion");
mEdicion.setMNemonic('E'); //ALT+E
JMenuItem mPegar = new JMenuItem("Pegar");
mPegar.setEnabled(false);
JMenuItem mCortar = new JMenuItem("Cortar");
mCortar.addActionListener(this);
mEdicion.add(mPegar);
mEdicion.addSeparator();
mEdicion.add(mCortar);
bm.add(mEdicion);
new JFrame().setJMenuBar(bm);
```

Swing - Menú

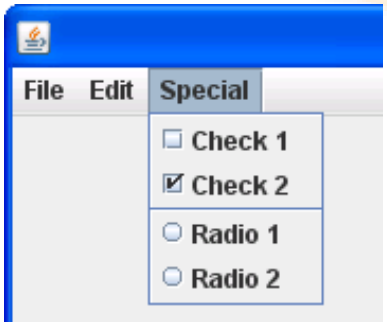
- Elementos de menú con íconos

```
JMenuItem mCortar = new JMenuItem(“Cortar”, new
   ImageIcon(“cut.gif”));
mCortar.setHorizontalTextPosition(
   SwingConstants.LEFT);
mCortar.setAccelerator(KeyStroke.getKeyStroke(
   ‘C’, InputEvent.CTRL_MASK)); // CTRL+C
mCortar.setEnabled(false);
```

Swing - Menú

- Elementos de menú con casillas de verificación y botones de radio.

```
JCheckBoxMenuItem opcSL = new
   JCheckBoxMenuItem("Solo lectura"));
mOpc.add(opcSL);
ButtonGroup g = new ButtonGroup();
JRadioButtonMenuItem opcI = new
   JRadioButtonMenuItem("Insertar");
opcI.setSelected(true);
JRadioButtonMenuItem opcS = new
   JRadioButtonMenuItem("Sobreescribir");
g.add(opcI);     g.add(opcS);
mOpc.add(opcI); mOpc.add(opcS);
```



Swing - Menú

- Menús emergentes
  - Es un menú que no está asociado a una barra de menús, sino que flota en algún lugar.

```
JPopupMenu emergente = new JPopupMenu();
JMenuItem opcion = new JMenuItem("Cortar");
opcion.addActionListener(this);
emergente.add(opcion);

emergente.show(panel, x, y);
```

Swing - Barra

- Barra de herramientas
  - Permite acceder rápidamente a las órdenes de uso mas frecuente en un programa.
  - Se puede trasladar a otro lugar (arrastrar a cualquiera de los 4 bordes del marco). Esto funciona si la barra de herramientas se encuentra dentro de un contenedor dotado de una disposición de borde).

```
JToolBar barra = new JToolBar();
barra.add(new JButton(“copiar”));
barra.addSeparator();
barra.add(new JButton(“pegar”));
new JFrame().add(barra, BorderLayout.NORTH);
```

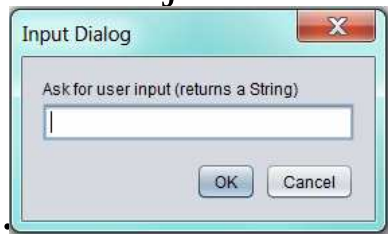
Swing - Menú

- Ayuda emergente
  - Se activa cuando el cursor se ubica sobre un botón durante algunos instantes.
  - El texto de la ayuda emergente se muestra dentro de un rectángulo coloreado.
  - Cuando el usuario aparte el mouse, la ayuda emergente desaparece.
  - Se puede agregar ayudas emergentes a cualquier JComponent.

```
new JButton("OK").setToolTipText("Esto es una
   ayuda emergente");
```

Swing - Diálogo Predefinido

- Para dar o pedir información al usuario.
- Diálogos con opciones:
  - showMessageDialog
    - (Mensaje + OK)
  - showConfirmDialog
    - (Mensaje + OK + Cancel)
  - showOptionDialog
    - Mensaje y obtiene una opción del usuario de un cierto conjunto de opciones.
  - showInputDialog
    - Mensaje y obtiene una línea de entrada del usuario.



Swing - Diálogo Predefinido

- showMessageDialog(
  - Componente_padre, //puede ser null
  - Object, //mensaje
  - String, //titulo
  - int, // Tipo de mensaje (ERROR_MESSAGE, INFORMATION_MESSAGE, WARNING_MESSAGE, QUESTION_MESSAGE, PLAIN_MESSAGE)
  - Icon) : void

```
JOptionPane.showMessageDialog(null, “El campo es
  obligatorio”, “Precaución”,
  JOptionPane.WARNING_MESSAGE, null);
```

Swing - Diálogo Predefinido

- showConfirmDialog(
  - Componente_padre, //puede ser null
  - Object, //mensaje
  - String, //titulo
  - int, // Tipo de mensaje (ERROR_MESSAGE, INFORMATION_MESSAGE, WARNING_MESSAGE, QUESTION_MESSAGE, PLAIN_MESSAGE)
  - int, //Tipo opción (DEFAULT_OPTION, YES_NO_OPTION, YES_NO_CANCEL_OPTION, OK_CANCEL_OPTION)
  - Icon) : int

```
JOptionPane.showConfirmDialog(null, “Esta seguro
  que desea salir?”, “Aviso”,
  JOptionPane.QUESTION_MESSAGE,
  JOptionPane.YES_NO_OPTION, null);
```

Swing - Diálogo Predefinido

- showOptionDialog(
  - Componente_padre, //puede ser null
  - Object, //mensaje
  - String, //titulo
  - int, //Tipo opción (DEFAULT_OPTION, YES_NO_OPTION, YES_NO_CANCEL_OPTION, OK_CANCEL_OPTION)
  - int, // Tipo de mensaje (ERROR_MESSAGE, INFORMATION_MESSAGE, WARNING_MESSAGE, QUESTION_MESSAGE, PLAIN_MESSAGE)
  - Icon,
  - Object[], //puede ser iconos, cadenas o componentes
  - Object // opción predeterminada que debe mostrarse al usuario

  ): int //indice de opción seleccionada por el usuario o CLOSED_OPTION si el usuario ha cancelado el diálogo.

Swing - Diálogo Predefinido

- showInputDialog(
  - Componente_padre, //puede ser null
  - Object, //mensaje
  - String, //titulo
  - int, // Tipo de mensaje (ERROR_MESSAGE, INFORMATION_MESSAGE, WARNING_MESSAGE, QUESTION_MESSAGE, PLAIN_MESSAGE)
  - Icon,
  - Object[], //valores que se muestran en un combo.
  - Object   // valor predeterminado que debe mostrarse al usuario.

  ): Object   // valor que el usuario ha escrito o seleccionado.

Swing - Diálogo

- JDialog

```
public class DialogoAcercaDe extends JDialog {
  public DialogoAcercaDe(JFrame propietario){
      super(propietario, “Acerca de”, true);
      add(new JLabel(“Java”),
      BorderLayout.CENTER);
      JButton b = new JButton(“OK”);
      b.addActionListener(new ActionListener() {
       public void actionPerformed(ActionEvent e)
       {    setVisible(false);
       }});
      add(b, BorderLayout.SOUTH);
      setSize(250, 150); }}
```

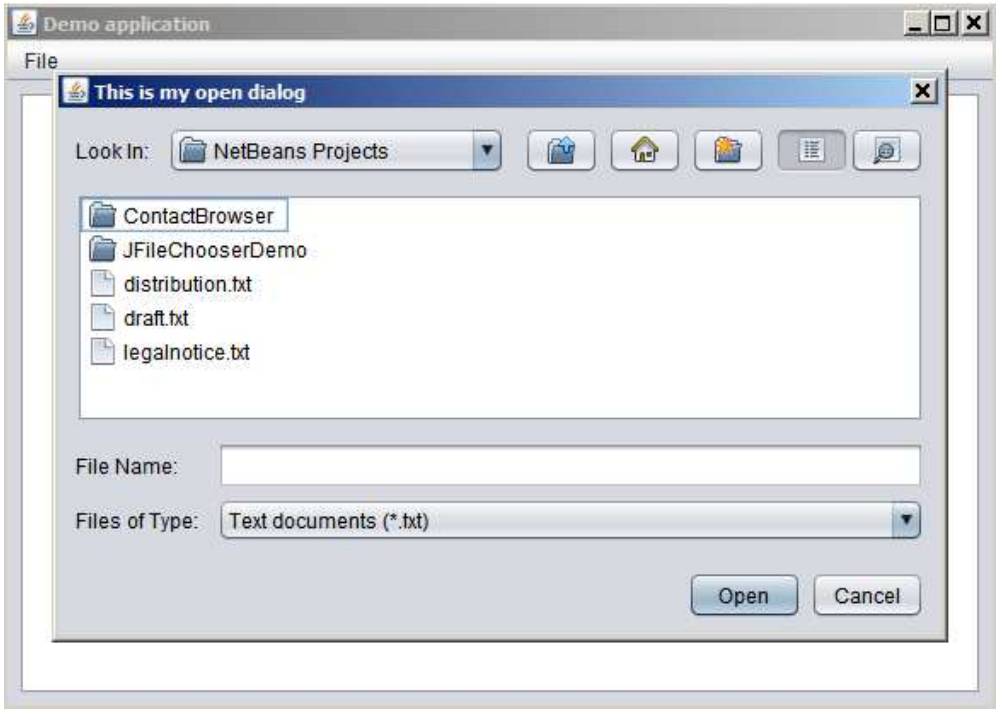
Swing - Diálogo Predefinido

- JFileChooser

```
JFileChooser sel = new JFileChooser();
sel.setCurrentDirectory(new File(“.”));
sel.setSelectedFile(new File(“Prueba.txt”));
int res = sel.showOpenDialog(padre);
   //o
int res = sel.showSaveDialog(padre);
res //puede ser JFileChooser.APPROVE_OPTION,
  JFileChooser.CANCEL_OPTION o
  JFileChooser.ERROR_OPTION
sel.getSelectedFile().getPath();
sel.setFileFilter(FileFilter);
```

Swing - Diálogo Predefinido

- JFileChooser



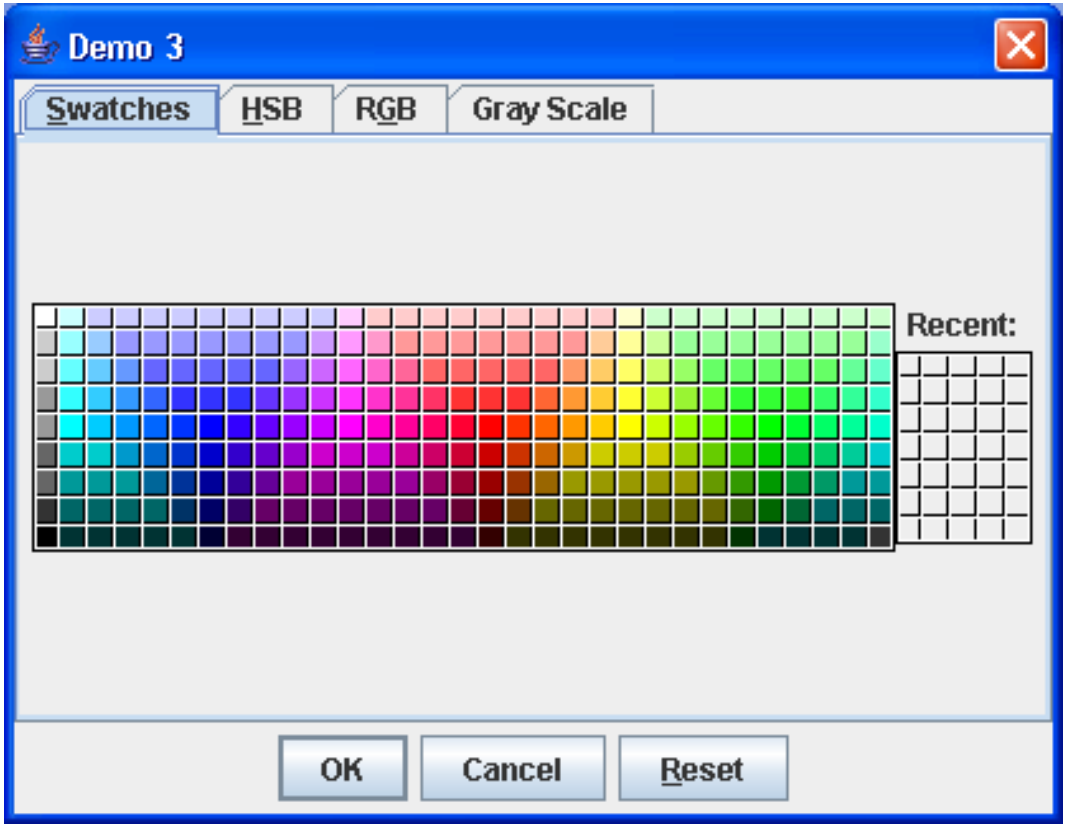
Swing - Diálogo Predefinido

- JColorChooser

```
JColorChooser sel = new JColorChooser();
JDialog dialogo = JColorChooser(
 padre,
 “Color fondo”, //titulo
 false, // no modal
 sel,
 new ActionListener(){
  public void actionPerformed(ActionEvent e){
   setBackground(sel.getColor());
 }}),
 null); //no hay oyente para botón cancelar
```

Swing - Diálogo Predefinido

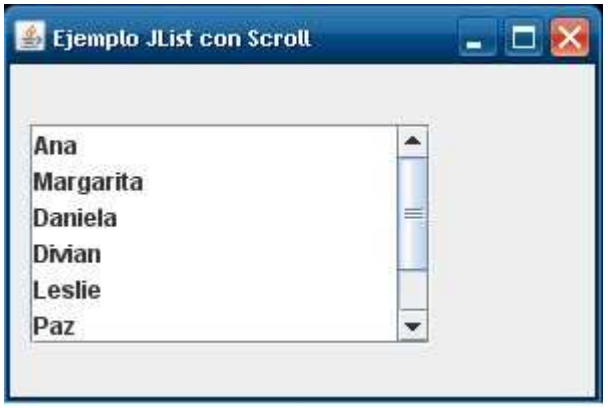
- JColorChooser



Swing - JList

- Se puede tener listas de cadenas y de objetos.
- Modelo → DefaultListModel (extiende a AbstractListModel, quien implementa a ListModel)
- Vista → JList
- Evento → ListSelectionListener

```
DefaultListModel modelo = new DefaultListModel();
JList lista = new JList(modelo);
modelo.addElement("a");
modelo.removeElement("a");
modelo.clear();
modelo.getSize();
marco.add(new JScrollPane(lista));
```



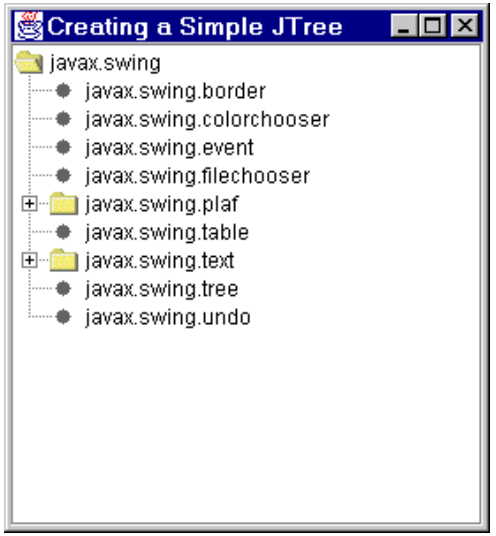
Swing - JList

```
lista.setVisibleRowCount(4);
lista.setLayoutOrientation(JList.VERTICAL);
lista.setSelectionMode(ListSelectionModel.MULTIPL
  E_INTERVAL_SELECTION);
lista.getSelectedValue(): Object
lista.getSelectedIndex(): int
lista.getSelectedValues(): Object[]
lista.getSelectedIndices(): int[]
lista.addListSelectionListener(new
  ListSelectionListener(){
  public void valueChanged(ListSelectionEvent e){
  }
  })
```

Swing - JTree

- Estructuras arborescentes, formadas por nodos (hojas, nodos descendientes)
- Modelo → DefaultTreeModel
- Vista → JTree
- Evento → TreeSelectionListener

```
DefaultMutableTreeNode raiz = new
   DefaultMutableTreeNode("raiz");
DefaultMutableTreeNode nodo = new
   DefaultMutableTreeNode("nodo");
raiz.add(nodo);
DefaultTreeModel modelo = new
   DefaultTreeModel(raiz);
JTree arbol = new JTree(modelo);
```



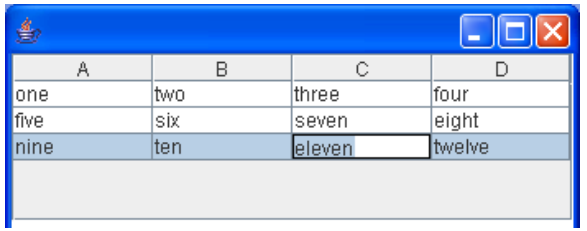
Swing - JTree

```
marco.add(new ScrollPane(arbol));
arbol.getLastSelectedPathComponent();
arbol.setRootVisible(false);
arbol.scrollPathToVisible(ruta);
arbol.setEditable(true);
Path ruta = arbol.getSelectionPath();
arbol.addTreeSelectionListener(new
  TreeSelectionListener() {
  public void valueChanged(TreeSelectionEvent e){
  }});
modelo.insertNodeInto(nodoNew, nodoSelected,
  nodoSelected.getChildCount());
modelo.removeNodeFromParent(nodoSelected);
```

Swing - JTable

- Cuadricula bidimensional de objetos.
- Modelo → DefaultTableModel
- Vista → JTable
- Evento → TableModelListener

```
DefaultTableModel modelo=new DefaultTableModel();
JTable tabla = new JTable(modelo);
modelo.addColumn("columna1");
modelo.addColumn("columna2");
modelo.addRow(new String[]{"a1", "b1"});
tabla.setPreferredScrollableViewPortSize(new
   Dimension(500, 70));
marco.add(new JScrollPane(tabla));
```



Swing - JTable

```
modelo.getRowCount();
modelo.getColumnCount();
modelo.removeRow(1);
modelo.getValueAt(0,0);
modelo.setValueAt(“b2”, 0, 0);
modelo.getColumnName(1);
modelo.insertRow(0, new String[]{“c1”, “c2”});
modelo.moveRow(int desde, int hasta, int
  aPartirDeAqui);
tabla.print();
tabla.getSelectedRow(): int
tabla.getSelectedColumn(): int
tabla.clearSelection();
```

Swing - JTable

```
modelo.addTableModelListener(new
   TableModelListener(){
    public void tableChanged(TableModelEvent e){
    }
   });

tabla.addMouseListener(new
  MouseListener(){
    public void mouseClicked(MouseEvent e){
    }
    ...
   });
```

Swing - Organizadores de componentes

- Componentes que ayudan a organizar otros componentes:
  - Láminas partidas (JSplitPane)
  - Láminas con solapas (JTabbedPane)
  - Láminas de escritorio (JDesktopPane)
  - Marcos internos (JInternalFrame)

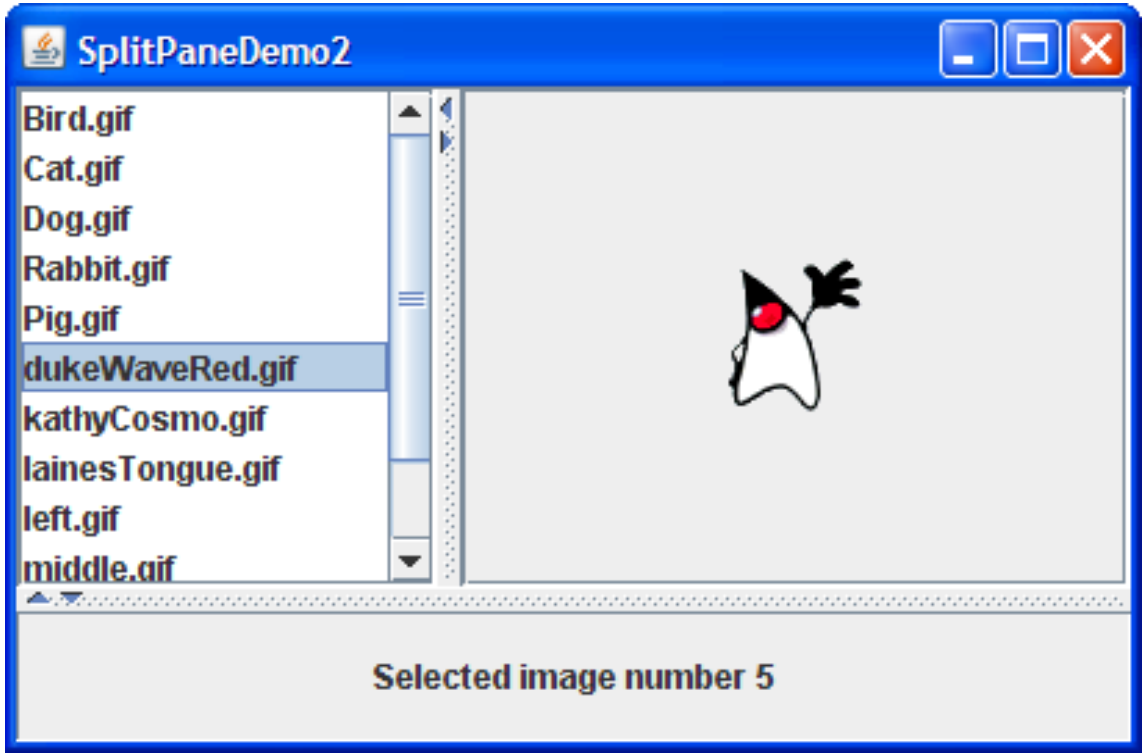
Swing - Organizadores de componentes

- Láminas partidas (JSplitPane)
  - Fragmentan un componente en dos partes.
  - Se construyen especificando la orientación:
    - JSplitPane.HORIZONTAL_SPLIT
    - JSplitPane.VERTICAL_SPLIT

```
JSplitPane p = new
  JSplitPane(JSplitPane.HORIZONTAL_SPLIT, new
  JPanel(), new JPanel());
p.setOneTouchExpandable(true); // iconos de
  expansión al primer clic en la barra de
  separación
p.setContinuousLayout(true);  // repinta
  continuamente a medida que se desplaza la barra
```

Swing - Organizadores de componentes

- Láminas partidas (JSplitPane)



Swing - Organizadores de componentes

- Láminas con solapas (JTabbedPane)
  - Fragmentan cuadros de diálogos complejos en subconjuntos de opciones relacionadas.

```
JTabbedPane p = new JTabbedPane();
JTabbedPane p1= new
   JTabbedPane(SwingConstants.TOP); //ubicación
p.addTab("unTítulo", icono, new JPanel());
p.addTab("unTítulo", new JPanel());
p.insertTab("Titulo", icono, new JPanel(),
   "ayudaEmergente", indice);
p.removeTabAt(indice);
p.setSelectedIndex(p.getTabCount()-1);
p.setTabLayoutPolicy(JTabbedPane.WRAP_TAB_LAYOUT)
```

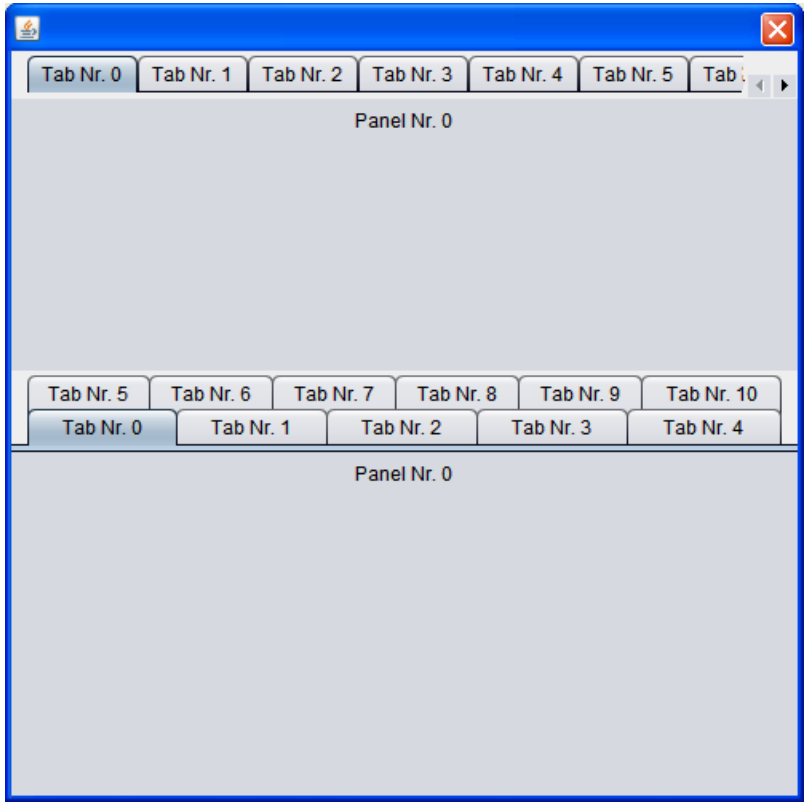
Swing - Organizadores de componentes

- Láminas con solapas (JTabbedPane)

```
p.indexOfTab(“unTitulo”);
p.indexOfTab(icono);
p.indexOfComponent(Component c);
p.setEnabledAt(1, false);
p.getSelectedIndex(): int
p.addChangeListener(new ChangeListener() {
   public void stateChanged(ChangeEvent e){
   }
   });
```

Swing - Organizadores de componentes

- Láminas con solapas (JTabbedPane)



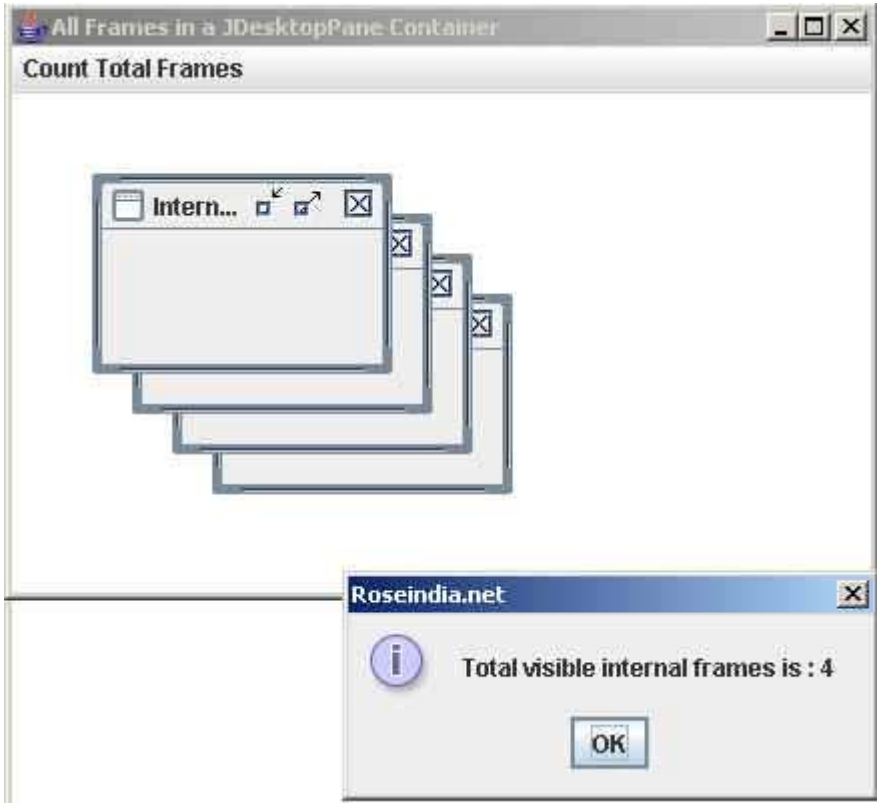
Swing - Organizadores de componentes

- Láminas de escritorio (JDesktopPane y JInternalFrame)

```
JFrame marco = new JFrame();
JDesktopFrame escr  = new JDesktopPane();
marco.add(escr);
JInternalFrame mint = new
  JInternalFrame(“unTitulo”,
                 true, //redimensionable
                 true, //se puede cerrar
                 true, //se puede maximizar
                 true);//puede reducir a icono
mint.add(new JButton(“OK”));
mint.setVisible(true);
escr.add(mint);
```

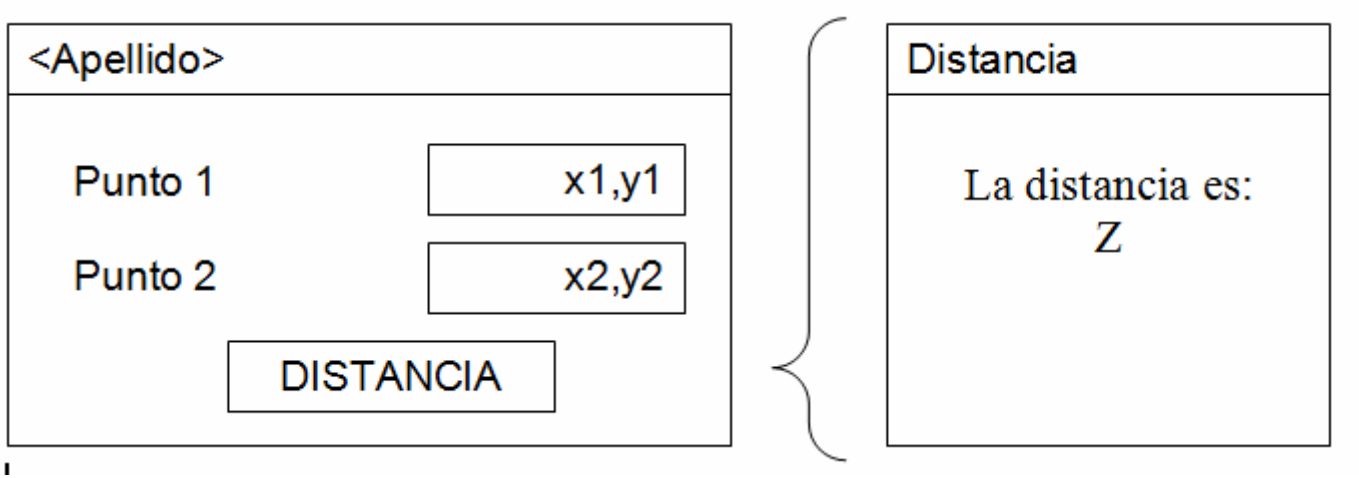
Swing - Organizadores de componentes

- Láminas de escritorio (JDesktopPane y JInternalFrame)



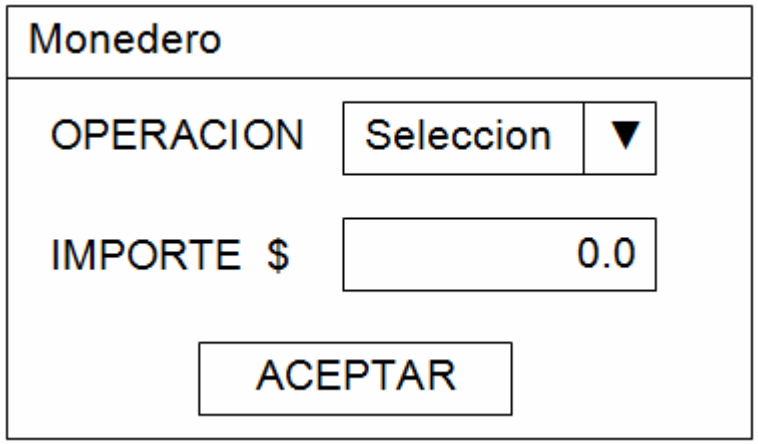
Swing - Ejercicios

- **Ejercicio 1:** Implementar una aplicación Java (siguiendo el patrón MVC) que permita al usuario ingresar dos puntos en el plano y muestre la distancia entre ambos en un diálogo predefinido.



Swing - Ejercicios

- **Ejercicio 2:** Implementar una aplicación Java (siguiendo el patrón MVC) que permita interactuar con la siguiente interfaz usuario:



Swing - Ejercicios

- **Ejercicio 3:** Implementar una aplicación Java que ofrezca como interfaz usuario una lista y dos botones: “Mutar” y “Mejor adaptado”:
  - a) Cargar en la lista la población de Cromosomas, cuyos genes son obtenidos de la base de datos. Ejemplo {1110, 0111, 0010, 1010}
  - b) Cada cromosoma esta formado por un conjunto finito de genes (símbolos binarios) y una adaptabilidad que se calcula decodificando los genes (decodificación genética). Ejemplo: 0111 adaptabilidad 7.
  - c) Al seleccionar un cromosoma en la lista se visualizará (tooltiptext) la adaptabilidad que el cromosoma tiene en ese momento.
  - d) Mutar: La mutación es una alteración genética, es decir, es una alteración aleatoria del valor que toma una posición en la cadena. Ejemplo: k=3 cromosoma seleccionado=0111 → cromosoma mutado=0110. Se recalcula adaptabilidad.
  - e) Mejor adaptado: Mostrar en un dialogo el cromosoma mejor adaptado (el que tiene mayor adaptabilidad).

Servlets

1) Crear workspace
2) Perspectiva JavaEE
3) Crear proyecto (File/New/Dinamic Web Project) y ponerle un Nombre (Prueba)
4) Definir el nombre del proyecto (Project name:Prueba) y el servidor en el cual se deploya (Target runtime: Apache Tomcat v7.0)
5) Definir Web module (Context root: Prueba) y (Generate web.xml deployment descriptor en TRUE) + <Finish>
6) Crear paquete servlets en src
7) Dentro del paquete servlets crear un Servlet (File/New/Servlet)
8) Definir el nombre del Servlet (Class name: MiServlet) <Finish>
9) En cuerpo de doGet poner:

```
ServletOutputStream out = response.getOutputStream();
out.println("Hola mundo, Servlet!");
out.flush();
out.close();
```

Servlets

- 10) Botón derecho Run As/Run on Server
- 11) Seleccionar Apache/Tomcat v7.0 Server <Finish>
- 12) En navegador URL: http://localhost:8080/Prueba/MiServlet
- 13) Aparecerá: Hola mundo, Servlet!

- doGet es cuando se llama la aplicación desde la URL, este llamado puede contener parámetros la sintaxis es la siguiente:
- http://localhost:8080/Prueba/MiServlet?usuario=pablo&password=12345
- el símbolo ? es para iniciar los parámetros y el & es para separarlos.

- 14) En cuerpo de doGet poner:

```
String usuario = (String) request.getParameter("usuario");
String password = (String) request.getParameter("password");
ServletOutputStream out = response.getOutputStream();
out.println("Su usuario es = " + usuario + " y password = " + password);
out.flush();
out.close();
```

Servlets

- 15) Crear una carpeta dentro de WebContent (File/New/Folder)

- 16) Definir el nombre de la carpeta (Folder name: templates)

- 17) Dentro de la carpeta templates crear un archivo index.html con el siguiente codigo:

```
<html>
	<form action="http://localhost:8080/Prueba/MiServlet" method="POST">
		<p>Usuario</p>
		<input type="text" id="usuario" name="usuario"/>
		<br>
		<p>Password</p>
		<input type="password" id="password" name="password"/>
		<input type="submit" value="Enviar"/>
	</form>
</html>
```

Servlets

18) Dentro de la carpeta templates crear un archivo resultado.jsp con el siguiente codigo:

```
<html>
<body>
        <%=(String)request.getAttribute("resultado")%>
</body>
</html>
```

19) En el cuerpo de doPost poner:

```
        String password = (String) request.getParameter("password");
        if (password.equals("12345678"))
                request.setAttribute("resultado", "OK");
        else
                request.setAttribute("resultado", "FAIL");
        String url = "/templates/resultado.jsp";
        ServletContext sc = getServletContext();
        RequestDispatcher rd = sc.getRequestDispatcher(url);
        rd.forward(request, response);
```

Servlets

- 20) Editar web.xml
- <?xml version="1.0" encoding="UTF-8"?>
- <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.com/
- xml/ns/javaee" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/
- xml/ns/javaee/web-app_3_0.xsd" id="WebApp_ID" version="3.0">
- <display-name>Prueba</display-name>
- <welcome-file-list>
- <welcome-file>/templates/index.html</welcome-file>
- </welcome-file-list>
- </web-app>

Rubricas

PAQ	**Paquetes**	Artefactos (clases, interfaces, annotations, enums, folders, files, packages) organizados en paquetes según especificación. Los nombres de paquetes se escriben en letras minúsculas.
SEG	**Seguridad**	Ocultar detalles de implementación: usar correctamente modificadores de acceso: private, protected y public. Todos los atributos del objeto son privates, solo las constantesson public y final. Métodos que necesiten un orden de ejecución declararlos como privates.
SEM	**Semántica**	Nombres de clases, métodos y variables deben declarar claramente lo que hacen. Uso de CamelCase("Clase", "ClaseNombreLargo", "metodo", "metodoNombreLargo", "variable", "variableNombreLargo", "CONSTANTE", "CONSTANTE_NOMBRE_LARGO").
ADV	**Advertencias**	Solucionar las advertencias informadas por el Compilador. Revisar cada mensaje para tomar las medidas pertinentes.
DOC	**Documentación**	Todo programa debe ser previamente documentado, explicando el propósito, funcionamiento completo y el resultado esperado.
IND	**Indentación**	Dentro de las funciones definidas, establecer un espaciado o indentación, que resalte la estructura funcional de la aplicación y facilite la lectura al programador al que le corresponda analizar el código.
VAR	**Variables**	Se recomienda declarar variables en líneas separadas, ya que se facilita la descripción de cada variable mediante comentarios. Poner un espacio después de cada coma(,) facilita la legibilidad del código.
LEB	**Líneas en blanco**	Separan declaración de clase, atributos de constructores, constructores de métodos, métodos de métodos.
SEN	**Sentencia**	Evitar la incorporación de más de una sentencia por línea.
OPE	**Operadores**	En caso de usar operadores binarios (por ejemplo +, -, &&, ||, entre otros) se recomienda poner espacio a los extremos de cada operador.
DEP	**Depuración**	Eliminar código comentado y de depuración
GYS	**Getters/Setters**	Definir métodos accesores, getters y setters. La mejor forma de hacer inmodificable el objeto es devolviéndolo clonado en lugar del objeto mismo
TYS	**this/super**	Hacer uso de ellos.
HAR	**Hardcode**	Evitar “hardcode”, usar constantes y/o archivo de configuración.
REU	**Reutilización**	Reutilización (no repetir conocimiento – clases, métodos, librerías, documentación, etc.) Usar librerías estándar.
COH	**Cohesión**	Maximizar la cohesión: métodos que realicen una sola tarea.
ACO	**Acoplamiento**	Minimizar el acoplamiento: Cada componente (bloque de código, clase, método, etc.) debe minimizar las dependencias con otros componentes.
TRA	**Trazabilidad**	Trazabilidad entre Diseño e Implementación.

Rubricas

LDP	**Lotes de Prueba**	En el paquete /resources/lotesprueba/in colocar los archivos de lotes de prueba con el siguiente formato de nombre 00_xxx.in y en el paquete /resources/lotesprueba/out se generaran los archivos de salida con el siguiente formato de nombre 00_xxx.out. Los lotes de prueba deben estar orientados a casos normales, casos de datos inválidos y casos de performance.
RES	**Resultado**	Resultado de la Ejecución de los lotes de prueba.
PRE	**Presentación**	CD + Carpeta (Objetivos, Enunciado del Problema, Análisis, Diseño de Clases -UML- y/o Datos -DER-, Implementación, Conclusiones)
GAN	**GANTT**	Planificación del Trabajo (Tareas vs. Tiempo Estimado vs. Tiempo Real)
MET	**Métricas**	Líneas de Código (LDC) discriminadas por Pizarrón/Ejercicios/TP
CCE	**Complejidad Espacial**	Ordenes de Complejidad Computacional Espacial. Justificar representación de los datos en memoria.
CCT	**Complejidad Temporal**	Ordenes de Complejidad Computacional Temporal. Justificar tiempos en función del tamaño de las entradas.
RME	**Representación memoria**	Representación gráfica de los datos en memoria.
RTI	**Representación tiempo**	Representación gráfica de tiempos de respuesta vs tamaño de las entradas.

Cronograma 2015

Clase 1	Presentación de la materia. Introducción a la Programación con Objetos (conceptos fundamentales)
Clase 2	Introducción a la Programación con Objetos (diseño de software) Casos de estudio: Envío de flores.
Clase 3	Introducción al Lenguaje de Programación Java (conceptos fundamentales). Aplicación Hola Mundo
Clase 4	Introducción al Lenguaje de Programación Java (elementos del lenguaje Java)
Clase 5	Encapsulamiento, Herencia y Polimorfismo. (Encapsulamiento)
Clase 6	Encapsulamiento, Herencia y Polimorfismo. (Herencia)
Clase 7	Encapsulamiento, Herencia y Polimorfismo. (Polimorfismo)
Clase 8	Colecciones (Pilas, Colas, Listas)
Clase 9	Colecciones (Mapas)
Clase 10	Excepciones y Archivos (Excepciones)
Clase 11	Excepciones y Archivos (Archivos)
Clase 12	JDBC (básico)
Clase 13	JDBC (avanzado)
Clase 14	Repaso
Clase 15	Parcial

Clase 16	Resolución del parcial
Clase 17	Swing (Ventanas y láminas)
Clase 18	Swing (Eventos)
Clase 19	Swing (Layouts)
Clase 20	Swing (Componentes básicas)
Clase 21	Swing (Menús y diálogos)
Clase 22	Swing (Componentes avanzadas)
Clase 23	Servlets (básico)
Clase 24	Servlets (gestión de sesiones)
Clase 25	Servlets (JSP)
Clase 26	Servlets (JSP)
Clase 27	Corrección de TP
Clase 28	Corrección de TP