ESCUELA DE SISTEMAS Y TECNOLOGÍAS

Transparencias de **ANALISTA DE SISTEMAS**

***Edición 2020 Materia:* Java Web**

TEMA: Lenguaje Java

Plantel y Contactos

- **Bedelía:**
  - Mail: bedeliasistemas@bios.edu.uy
- **Encargado de Sucursal:**
  - Pablo Castaño
  - Mail: pablocasta@bios.edu.uy

Recursos

- **Recursos Imprescindibles:**
  - Sitio Web de material (comunicarse con Bedelía por usuario/contraseña).
  - Transparencias del Curso.
  - Contar con el software necesario

Consideraciones

- Estas transparencias no tienen el objetivo de suplir las clases.
- Por tanto, serán complementadas con ejemplos, códigos, profundizaciones y comentarios por parte del docente.
- El orden de dictado de estos temas está sujeto a la consideración del docente.
- Lo que sigue es un resumen de los elementos más comúnmente utilizados del lenguaje Java, tomando como base los conocimientos ya adquiridos de la plataforma .NET y el lenguaje C#.

Acerca de la Materia (1)

- **Objetivos**: Desarrollar aplicaciones web de mediano porte, aplicando el patrón MVC y capas, utilizando las siguientes tecnologías:
  - Lenguaje Java
  - Orientación a Objetos en Java y UML
  - Reflection
  - Acceso a Datos con JDBC
  - Bases de Datos en MySQL
  - Servlets y páginas JSP
  - Servicios Web Java

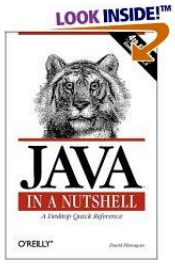
Acerca de la Materia (2)

- **Herramientas a Utilizar:**
  - OS: Windows 7 o posterior
  - IDE: NetBeans® (www.netbeans.org)
  - DBMS: MySQL Server® (www.mysql.org)

- **Lenguajes a Utilizar:**
  - Java
  - UML
  - SQL

Recursos (1)

Bibliografía:

	*"Java in a Nutshell"* de David Flanagan. Cuarta Edición. O'Reilly. Marzo 2002. ISBN 0596002831
	*"Piensa en Java"* de Bruce Eckel. Segunda Edición. Editorial Pearson. 2002. ISBN 8420531928. Versión **gratuita** en inglés: http://www.mindviewinc.com/downloads/TIJ-3rd-edition4.0.zip
	Sitio Web de Java http://www.oracle.com/technetwork/java/ http://docs.oracle.com/javase/tutorial/

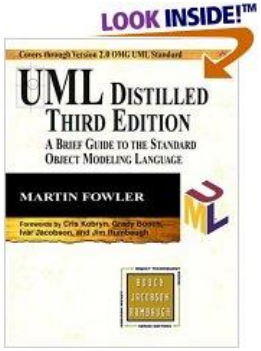
Recursos (2)

- **Sobre la Bibliografía:** Los libros referidos son principalmente para que el alumno pueda complementar y profundizar algunos temas, no siendo imprescindibles para el curso.

- **Recursos Imprescindibles:**
  - Sitio Web: http://www.portalbios.com/sistemas
  - Transparencias del Curso.
  - Contar con el software necesario.

Recursos (3)

- **Sobre UML:**
  - Libro recomendado: "***UML Distilled***" de Martin Fowler, 3rd Edition. Addison Wesley. 2003. Traducción: "UML Gota a Gota".
  - ***www.uml.org*** (Object Management Group)
  - "***Notas sobre UML***" disponible en el sitio web del curso.
  - Herramientas de diseño de diagramas (ejemplos):
    - Microsoft Visio (http://office.microsoft.com/visio)
    - Draw.io (https://www.draw.io)
    - Magic Draw (http://www.magicdraw.com)
    - UML Poseidon (http://www.gentleware.com)
    - ArgoUML (http://argouml.tigris.org)



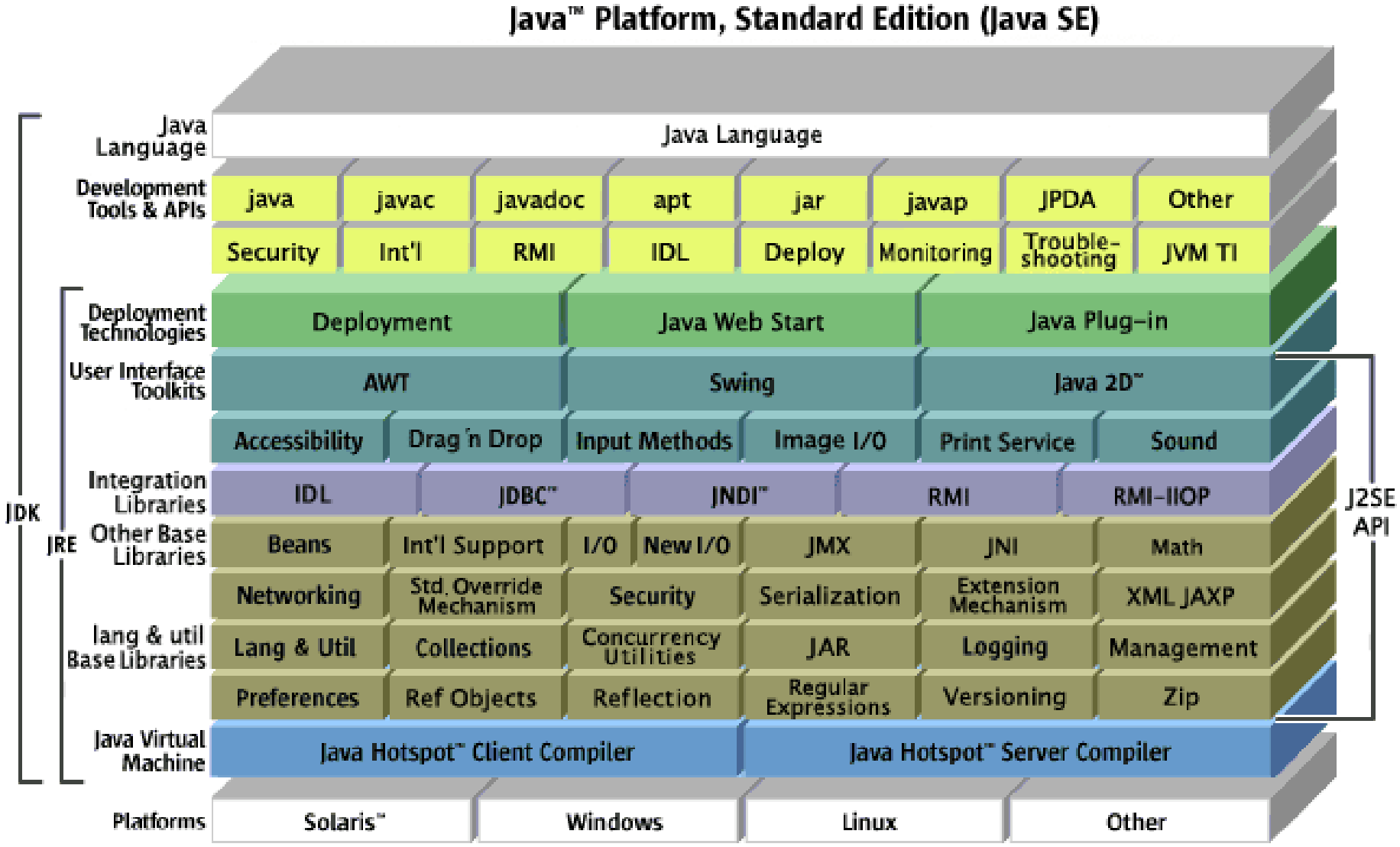
Agenda

- El Lenguaje Java
- Compilación
- Java Virtual Machine
- Diferencias Básicas Java / C#
- Salida y Entrada Estándar
- Conversión de tipos
- Colecciones de Largo Variable
- Manejo de Fechas y Horas
- Archivos de Texto
- Annotations
- Diferencias POO Java / C#
- Inner Classes e Inner Classes Anónimas
- Capas

El Lenguaje Java (1)

- Java es un lenguaje 100% orientado a objetos que tiene su origen a mediados de los años '90.
- Es un lenguaje simple, moderno, potente, flexible, distribuido ("*The network is the computer*®") y completo (varias plataformas: J2ME, J2SE, J2EE).
- Representa una evolución del ya consagrado lenguaje C++, y ha sido fuente de inspiración para otras plataformas y lenguajes (como .NET y C#).
- Permite el desarrollo completo de aplicaciones utilizando herramientas / tecnologías de acceso libre.
- Gracias a la JVM (*Java Virtual Machine*) su código intermedio (*bytecode*) puede ser ejecutado en cualquier equipo ("*Write it once, run it everywhere*®").

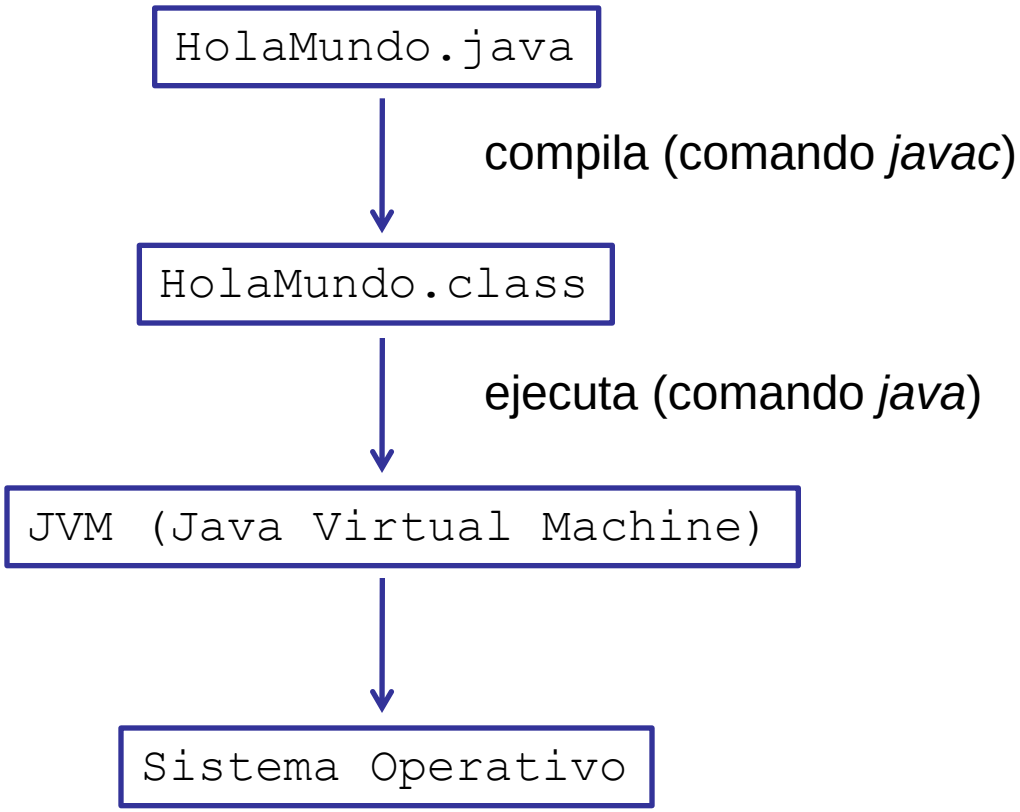
El Lenguaje Java (2)



Compilación (1)

- Java es un lenguaje que debe ser compilado a código que será interpretado por una máquina virtual (**JVM – Java Virtual Machine**).
- El compilador convierte el código fuente (archivo .java) en un conjunto de instrucciones llamadas **bytecode**, en un archivo cuya extensión es *.class*.
- El intérprete (máquina virtual) ejecuta cada una de estas instrucciones en un ordenador específico (Windows, Macintosh, etc.).
- El programa es compilado una vez, pero el bytecode generado se interpreta cada vez que se ejecuta.

Compilación (2)



Java Virtual Machine (1)

- La **JVM** es el entorno en el que se ejecutan los programas Java.
- Actúa como un puente entre el resultado de la compilación *(el **bytecode**)* y el sistema sobre el que se ejecuta la aplicación (Windows, Linux, etc.).
- Conoce el conjunto de instrucciones del sistema operativo sobre el que se ejecuta la aplicación, y traduce el bytecode a código nativo que es capaz de entender dicho sistema operativo.

Java Virtual Machine (2)

- Tiene instrucciones para los siguientes grupos de tareas:
  - Carga y almacenamiento
  - Liberación de memoria no usada (garbage collector)
  - Creación y manipulación de objetos
  - Conversión de tipos
  - Gestión del stack
  - Invocación y retorno de métodos
  - Excepciones

Diferencias Básicas Java / C# (1)

- Java es un lenguaje muy parecido al lenguaje C# (no olvidar que C# está fuertemente inspirado en Java y C++).
- Por lo tanto, los conocimientos ya adquiridos acerca del lenguaje C# son aplicables en un alto porcentaje al código que escribiremos en Java.
- Nos centraremos entonces en aquellos aspectos de Java que tengan una notoria diferencia con C#.
- Los aspectos del lenguaje Java que no tengan diferencias con el lenguaje C#, no serán tratados aquí

Diferencias Básicas Java / C# (2)

- **Estructura de un Programa**

```
package mipaquete;
import miotropaquete;
public class MiClase {
    double d = 1;
    public static void main(String[] args)
    {
        // punto de entrada del programa
        System.out.print("Hola Mundo!");
    }
    private static int f() {/*...*/}
    public static void g(int i) {/*...*/}
}
```

Diferencias Básicas Java / C# (3)

- **Estructura de un Programa**
  - **Método Main:**
    - En el ejemplo anterior la clase contiene al método main.
    - El método *main* es estático y es el punto de entrada del programa, es decir que el programa comienza a ejecutarse en ese método.
    - Los métodos del ejemplo son estáticos (aunque no tienen por qué serlo si no van a ser invocados por el método main o cualquier otro método estático).
  - **Paquetes:**
    - En el ejemplo anterior, la clase *MiClase* se encuentra definida dentro del paquete *mipaquete*.

Diferencias Básicas Java / C# (4)

❖ Paquetes (cont):

- Un paquete es una colección de clases, interfaces, etc.
- Permite una agrupación lógica de dichos elementos.
- Los directorios son una agrupación física de dichos elementos.
- La estructura de directorios debe respetar la jerarquía de paquetes.
- La cláusula *package* define a qué paquete pertenece la clase. Debe ser única.
- La cláusula *import* permite utilizar (importar) clases de otros paquetes. Pueden haber varias. Se puede utilizar paquete.* para importar todas las clases de un paquete.
- Al definir una clase, si se omite el modificador public, no será posible importarla desde otros paquetes.

Diferencias Básicas Java / C# (5)

- **Tipos de Datos Primitivos**:
  - **byte**: número entero de 8 bits (-128 a 127).
  - **short**: número entero de 16 bits (-32.768 a 32.767).
  - **int**: número entero de 32 bits (-2.147.483.648 a 2.147.483.647).
  - **long**: número entero de 64 bits (-9,223,372,036,854,775,808 a 9,223,372,036,854,775,807).
  - **float**: número real (punto flotante) de precisión simple de 32 bits.
  - **double**: número real (punto flotante) de precisión doble de 64 bits.
  - **boolean**: valor booleano (true/false).
  - **char**: carácter 16 bits Unicode (de '\u0000' a '\uffff').

Diferencias Básicas Java / C# (6)

- **Algunos Tipos de Datos No Primitivos**:
  - **String**: cadena de caracteres.
  - **BigDecimal**: número decimal de precisión arbitraria. Especialmente aconsejable para manejo de moneda.
  - Estos tipos de datos son clases, por ello no son primitivos y comienzan con mayúscula.
  - Por más información:

http://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html

Diferencias Básicas Java / C# (7)

- **Constantes**
  - Sintaxis:

    ```
    public static final float PI = 3.14;
    ```

  - Convención:
    - Utilizar nombres en mayúsculas

Diferencias Básicas Java / C# (8)

- **Comparación de Strings**:
  - En Java, los Strings se comparan con el método *equals*
  - Es preferible colocar primero la cadena literal (y no la variable) ya que ésta nunca será null. Si se hiciera al revés, y la variable apuntara a null, al invocar a equals(...) se dispararía una excepción de tipo NullPointerException

Diferencias Básicas Java / C# (9)

- **Bucle «*foreach*»**:
  - En general las estructuras de control del flujo de ejecución en Java son idénticas a las que existen en C#.
  - Pero en Java no existe el "foreach" como tal, sino que el "for" admite una segunda sintaxis (aparte de la clásica con inicialización, condición, actualización), que permite recorrer colecciones:

```
for (TipoBase variable : Coleccion)
{
        sentencias;
}
```

Diferencias Básicas Java / C# (10)

- **Procedimientos & Funciones**:
  - **Sintaxis**:

[<modif_acceso>] [**static**] <tipo_ret> <nombre> ({<tipo><param>[,]})
[throws <tipo_excepcion[,]>]
{
    <bloque>
    [**return** <expresion>;]
}

  - En Java todos los parámetros son pasados por valor.
  - **Para Procedimientos:** <tipo_ret> es **void**
  - En Java deben declararse en la firma de las operaciones, todas las excepciones (checked) que no sean controladas dentro del método.

Diferencias Básicas Java / C# (11)

- **Matrices**:
  - En Java los vectores son iguales a los de c#, pero las matrices se trabajan como vectores de vectores (2 dimensiones).
  - **Sintaxis**:

```
<tipo>[][] <nombre> = new <tipo>[<filas>][<cols>];
```

  - Para saber los largos:

```
matriz.length          // cant. filas
matriz[0].length       // cant. cols
```

Diferencias Básicas Java / C# (12)

- **Manejo de Excepciones**:
  - En Java los bloque catch siempre deben indicar el tipo de excepción que atrapan
  - **Sintaxis**:

```
try
{
            <bloque_try_con_código_útil>
}
catch (<tipo_excepción> <nombre>) {
   <bloque_catch_manejar_este_tipo_de_excepción>
}
...
finally
{
   <bloque_finally_siempre_se_ejecuta>
}
```

Diferencias Básicas Java / C# (13)

- **Manejo de Excepciones (cont.)**:
  - Un método tiene 3 formas de lidiar con excepciones ocurridas durante la ejecución de un bloque try:
    1) **Capturar** la excepción en algún bloque catch.
    2) **No capturar** la excepción y enviarla hacia el método que invocó a éste. Para ello, debe declararlo en el encabezado del método.
    3) **Capturar** la excepción en algún bloque catch pero de todas formas **enviarla** al método que invocó a éste. Esto puede ser útil en aquellos casos en donde este método puede resolver parte de la excepción ocurrida y debe informar al método invocante sobe la excepción para que éste haga lo propio

Salida y Entrada Estándar (1)

- **Salida**:

```
// Mostrar una línea (con salto).
   System.out.println("hola mundo!");
// Mostrar un texto (sin salto de línea).
   System.out.print("chau");
// Mostrar el valor de una variable.
    System.out.print("Resultado: " + result)
```

- No se requiere importación alguna (pertenece al paquete java.lang que se importa automáticamente).

Salida y Entrada Estándar (2)

- **Entrada**:

```
// Conectarse a la entrada estándar.
   Scanner teclado = new Scanner(System.in);
// Declarar una variable tipo String.
   String entrada;
// Leer una línea (hasta el ENTER).
   entrada = teclado.nextLine();
```

- Se debe importar java.util.Scanner
- Existen otras formas de leer de la entrada estándar

Conversión de Tipos (1)

- Java admite conversiones de tipo implícitas y explícitas.
  - Conversión implícita:
    - Ocurre cuando los dos tipos son compatibles (por ejemplo asignar un int a un long) o el tipo destino es de mayor rango que el tipo de origen.
    - La JVM realiza la conversión automáticamente, sin necesidad de información adicional.
  - Conversión explícita:
    - Los métodos **parseInt**, **parseLong**, **parseFloat** y **parseDouble** pertenecientes a las diversas clases predefinidas de Java permiten convertir un objeto de tipo String en un dato de tipo primitivo.
    - Éstos son sólo algunos de los métodos de conversión.

Conversión de Tipos (2)

- Conversión de un **String a número**: Si se introducen caracteres no numéricos o el texto contiene espacios en blanco al comienzo o al final, se lanza una excepción de tipo **NumberFormatException**, por lo cual este tipo de conversiones deben hacerse dentro de un bloque try/catch.

- Conversión de un **número a String**: La clase *String* proporciona versiones de *valueOf* para convertir los datos primitivos **int, long, float, double** a String.

Colecciones de Largo Variable (1)

- Java provee varios tipos de colecciones. Se mostrará a modo de ejemplo el uso de las colecciones **ArrayList** y **LinkedList** (ambas implementan la interfaz **List**, y pueden utilizar un parámetro *generic* para hacerlas fuertemente tipadas).

- **ArrayList**: Es una buena opción cuando se requiere recorrer mucho una colección y no hacer demasiadas inserciones ni borrados sobre la misma, ya que es bastante rápido al iterar sobre sus elementos y al hacer accesos aleatorios.

Colecciones de Largo Variable (2)

- **ArrayList (cont.)**:

```
ArrayList lista = new ArrayList();
lista.add(<algo1>);    // agrega un objeto
lista.remove(0);       // elimina <algo1>
lista.size()           //cantidad de elementos
lista.get (<pos>)      //obtiene elemento de la posición
```

  - Debe importarse java.util.ArrayList
  - Puede agregarse cualquier tipo de objetos (si no se utiliza el parámetro generic).

Colecciones de Largo Variable (3)

- **LinkedList**: Se usa para situaciones en las que se requieren muchas inserciones y borrados sobre la lista y pocos accesos aleatorios.
- Debe importarse java.util.LinkedList
- Implementan un doble enlace en cada elemento, uno hacia el elemento anterior y otro hacia el elemento siguiente, haciendo que la inserción y el borrado no sean tan costosos.
- El acceso aleatorio a los elementos es más lento que en el *ArrayList* ya que para acceder a un elemento determinado debe pasar por todos los anteriores

Colecciones de Largo Variable (4)

- **LinkedList (cont.)**:

```
LinkedList lista = new LinkedList();
lista.add(<algo1>);          // agrega un objeto
lista.addFirst(<algo3>);     // agrega al principio de la lista
lista.addLast(<algo4>);      // agrega al final de la lista
lista.removeFirst();         // elimina el primer objeto de la lista
lista.removeLast();          // elimina el último objeto de la lista
```

Manejo de Fechas y Horas (1)

- La clase **java.util.Date** permite manipular días, meses, años, horas, minutos y segundos.
- La clase **DateFormat** perteneciente al paquete *java.text* proporciona formatos predefinidos.
- Los más utilizados son:
  - DEFAULT (15-ene-2010)
  - SHORT (15/01/10)
  - MEDIUM (15-ene-2010)
  - LONG (15 de enero de 2010)
  - FULL (viernes 15 de enero de 2010)

Manejo de Fechas y Horas (2)

- Para formatear fechas:
  - se crea un formateador con el método **getDateInstance**.
  - se llama al método **format**, el cual devuelve un String que contiene la fecha formateada.
- Para formatear horas:
  - Las horas se manejan de forma similar a las fechas, excepto en que el formateador se crea con el método **getTimeInstance**.
  - Otra alternativa al manejo de fechas es utilizar la clase *SimpleDateFormat* del paquete *java.text*, la cual permite mostrar las fechas en el formato deseado o a reconstruirlas a partir de una cadena de texto.

Archivos de Texto (1)

- Frecuentemente los programas necesitan traer información desde una fuente externa o enviarla hacia una fuente externa.
- El paquete `java.io` contiene una colección de clases stream que soportan los algoritmos para leer y escribir.
- Para traer o enviar la información, el programa abre un stream sobre una fuente (archivo, memoria, socket) y la lee o escribe serialmente.
- Para leer un archivo de texto básicamente se precisa:
  - un **FileReader**: clase que contiene métodos para leer caracteres.

Archivos de Texto (2)

- Para leer un archivo de texto básicamente se precisa(cont.):
  - un **BufferedReader**: clase que contiene métodos para leer líneas completas. Se construye a partir del *FileReader*.
  - un bucle que lea el archivo línea a línea utilizando el método **readLine**() de la clase *BufferedReader* mientras haya información.
  - Si a la hora de realizar la lectura de un archivo éste no existiera se lanzará una excepción de tipo *FileNotFoundException*
- Escritura de un archivo de texto:
  - La clase PrintStream posee una funcionalidad que permite enviar todos los tipos primitivos en formato de texto.

Annotations (1)

- Son utilizadas para añadir metadatos al código fuente Java que están disponibles para la aplicación en tiempo de ejecución.
- Se aplican a declaraciones de paquetes, tipos, constructores, métodos, atributos, parámetros y variables locales.
- Además de las que ya existen, el JDK ofrece mecanismos para crear anotaciones de cualquier tipo e integrarles lógica personalizada.
- Siempre comienzan con @.
- Están disponibles a partir de la versión 1.5 del JDK.

Annotations (2)

- Son utilizadas por frameworks, herramientas, IDEs, etc.
- Algunos ejemplos:
  - **@Override**: le indica al compilador que el método al cual hace referencia redefine al método de la superclase.
  - **@Deprecated**: indica que el uso de determinado elemento no es recomendable o ha dejado de ser actualizado.
  - **@Suppresswarnings**: permite suprimir mensajes del compilador relacionados con advertencias/avisos.

Diferencias POO Java / C# (1)

- El modificador de acceso predeterminado es **protected** (accesible por cualquier clase que esté en el mismo *package*).
- Para que una clase no se pueda utilizar desde otro *package,* debe **omitirse** la palabra **public**: en vez de "public class ..." sería "class …".
- En Java no existen las "propiedades"; en su lugar se utilizan métodos públicos **getter** y **setter** (getX() / setX()).
- Si se trata de un valor booleano, el método getter sería isX().

Diferencias POO Java / C# (2)

- Para que una clase extienda de otra se utiliza la palabra reservada **extends**.
- Para que una clase implemente una interfaz se utiliza la palabra reservada **implements**.
- Para invocar a un método de la clase base se utiliza la palabra reservada **super** → super.operacion()
- Para invocar a un constructor de la clase base se utiliza la palabra reservada **super** → super(…)
- La invocación desde un constructor a otro (de la misma clase con **this**(…), o de la clase base con **super**(…)), debe ir dentro de las llaves del método y ser la primera línea de código.

Diferencias POO Java / C# (3)

- En Java, todas las operaciones son redefinibles por defecto (no hay que indicar virtual en la clase base).
- Por el contrario, si se desea que una operación no sea redefinible, hay que indicarlo con la palabra reservada **final**.
- Al redefinir una operación de una clase base, no hay que indicarlo explícitamente (no existe override). Aunque sí existe la annotation **@Override** para que el compilador no emita un warning al compilar.
- Para comprobar si una referencia apunta a un objeto de determinado tipo, se utiliza el operador **instanceof**.

Inner Class (1)

- Una Inner Class (o clase anidada) es una clase definida dentro del cuerpo de otra clase.
- Puede ser privada, protegida o pública.
- Tiene acceso a todos los miembros definidos en su clase huésped, incluso aquellos marcados como privados.
- La definición de una Inner Class dentro de una clase se justifica cuando su existencia depende de la existencia de la clase huésped.
- Generalmente se define para utilizarla exclusivamente dentro de la clase huésped.

Inner Class (2)

- Si se piensa instanciar la clase anidada fuera de la clase huésped, además de definirle visibilidad **public** o **protected**, debe marcarse como estática.

- En tal caso los miembros de la clase anidada no podrán acceder a miembros de instancia de la clase huésped aunque sí obviamente a los marcados como **static**.

Inner Class Anónima (1)

- Una inner class anónima es una clase que se define justo en el momento en que se necesita una instancia de dicha clase, y se sabe que no se necesitará en otro lugar.
- No tiene nombre (anónima) y se define a partir de una clase base existente o más comúnmente de una interfaz.
- En el mismo lugar donde se define, se utiliza el operador new para crear la instancia.
- En el cuerpo de la inner class anónima, generalmente se redefinen operaciones de su clase baso o interfaz. Es decir, son útiles para proporcionar nuevas implementaciones a las operaciones de la clase base o interfaz que serán utilizadas en un único lugar

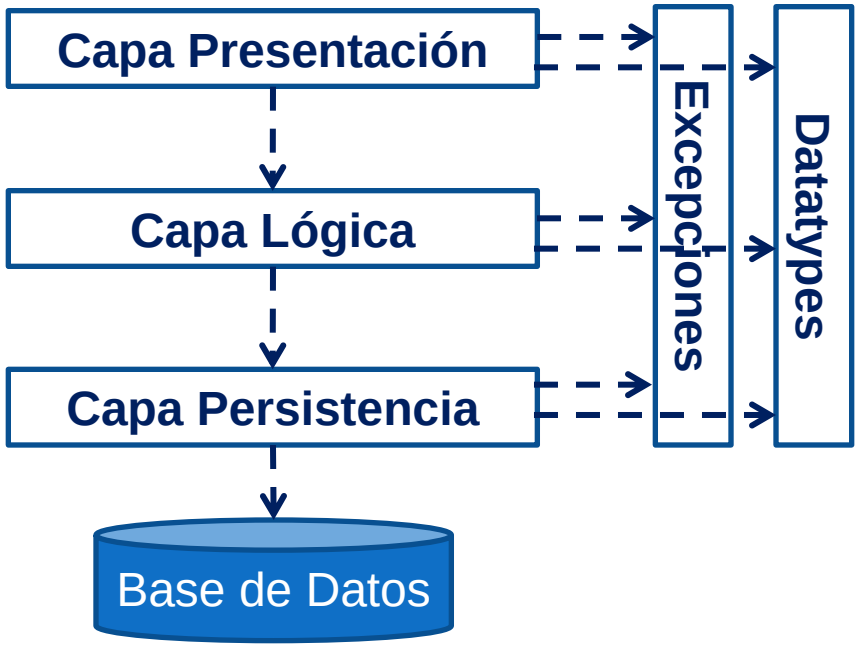
Inner Class Anónima (2)

- Por ejemplo, si un método necesita que le pasen por parámetro una instancia de A, se le puede pasar una nueva instancia de una inner class anónima que extienda o implemente A (según A sea una clase o interfaz), definida dentro de la propia invocación al método.

- Dentro de la inner class anónima se podrán redefinir operaciones de A, por lo que la instancia pasada al método, será un A (por reemplazabilidad), pero con una implementación particular.

Capas (1)

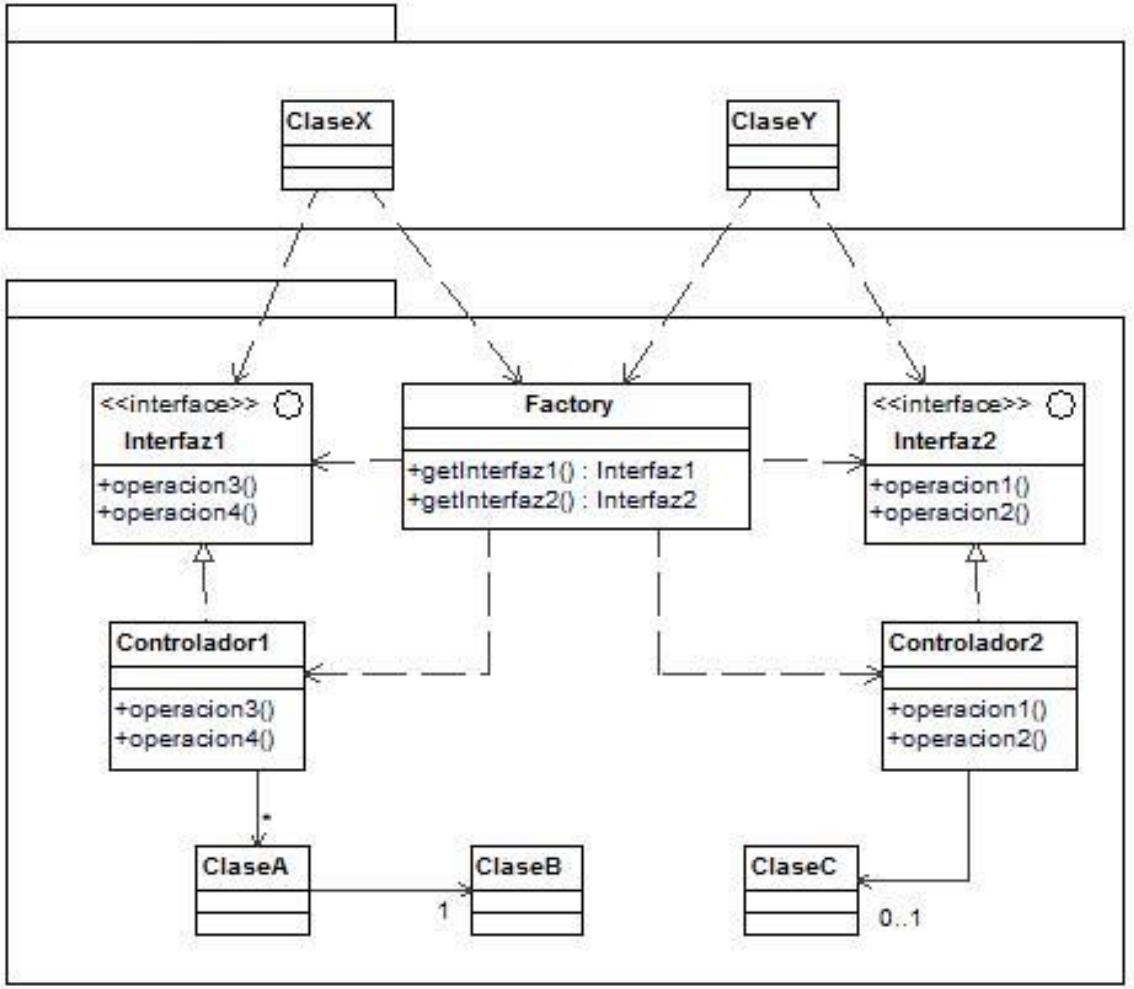
- Al trabajar en capas definiremos para cada una de éstas un package dentro del proyecto.
- Más adelante, al estudiar el patrón arquitectónico MVC, se verán más detalles acerca de la Capa Presentación.



Capas (2)

Capa i

Capa i+1



Capas (3)

- Tecnologías Java según Capa*:
  - **Presentación**:
    - Páginas JSP, Servlets, JavaBeans, etc.
  - **Lógica**:
    - Clases POJO (Plain Old Java Object: independientes de cualquier tecnología), etc.
  - **Persistencia**:
    - JDBC
  - **Datos**:
    - MySQL, archivos de texto, etc.

* *Lista no exhaustiva.*