Capítulo 2

- **Objetos en Java**
- **Manejo de Memoria al trabajar con Objetos**
- **Packages**
- **Compilando y Ejecutando Clases en Packages**
- **Herencia**
- **Polimorfismo**
- **Interfaces**
- **Casting e instanceof**
- **Arreglos en Java**

Objetos en Java

Creación de Instancias

- Una vez definida una clase en Java, es posible escribir otros programas que creen instancias (objetos) de dicha clase.

Ejemplo: Considérese nuevamente la clase Vehiculo del capítulo anterior. El siguiente programa crea varias instancias de dicha clase, utilizando los distintos constructores vistos en dicho capítulo.

```
/* PruebaVehiculo.java */

public class PruebaVehiculo
{
   public static void main (String [] args)
   {
      Vehiculo a = new Vehiculo ("Corsa", "AAA 4031", 15000);
      Vehiculo b = new Vehiculo ("Swift", "SCJ 3047", 20000);
      Vehiculo c = new Vehiculo ();
   }
}
```

Objetos en Java

Creación de Instancias (continuación)

Observación: Para que la clase **`PruebaVehiculo`** compile y ejecute Correctamente, debe estar contenida en el **mismo** directorio que la clase **`Vehiculo`**. Más adelante veremos cómo compilar colectivamente clases que están en distintos directorios.

Objetos en Java

Creación de Instancias (continuación)

- Para **instanciar** una clase (crear un objeto de ella) se debe **invocar** a alguno de sus constructores, mediante el uso del operador **new**
- Al hacer la invocación se debe tener en cuenta que:
  - Pasar tantos parámetros como haya en el cabezal del constructor.
  - Sus tipos deben ser compatibles con los del cabezal y en el mismo orden.
  - Pueden ser valores, variables, objetos, constantes o expresiones
- Si no se define **explicitamente** un constructor dentro del cuerpo de la clase, el compilador creará uno, llamado **constructor por defecto**. Dicho constructor no posee parámetros, y simplemente inicializa los atributos (que pudiera tener la clase) con valores nulos. El constructor por defecto de Java **NO** será creado si hay algún otro constructor definido en la clase.

Objetos en Java

Invocación a los métodos de una clase

- Muy Similar a C++, tanto para las funciones como para los procedimientos. Las consideraciones sobre los parámetros vistas para los constructores son también válidas para los procedimientos y las funciones.
- Un método de una clase puede invocar a otro de la misma clase en forma directa o mediante la palabra reservada **`this`**.

Ejemplo: Considere los siguientes métodos para la clase **`Vehiculo`**:

```
public double getPrecio()
{  return precio;  }

public double conImpuesto(double tax)
{  return tax * getPrecio();  }
```

o bien

```
public double conImpuesto(double tax)
{  return tax * this.getPrecio();  }
```

Objetos en Java

Invocación a los métodos de una clase (continuación)

- Para invocar métodos de otras clases, sólo podemos hacerlo en forma indirecta, mediante el operador **.** (punto) aplicado sobre un objeto.

Ejemplo:

```
/* PruebaVehiculo2.java */

public class PruebaVehiculo2
{
  public static void main (String [] args)
  {
    Vehiculo v = new Vehiculo("Corsa","AAA 4031",15000);
    v.setPrecio(15000);
    String datos = "Datos del Vehiculo:";
    System.out.println(datos + v.toString());
  }
}
```

Manejo de Memoria al trabajar con Objetos

Referencias

- A diferencia de C++, Java **no** provee al programador manejo de punteros. Pero, internamente, la asignación de memoria al crear instancias siempre se realiza en forma **dinámica**.
- Cuando **declaramos** una instancia de una clase, lo que se crea es en realidad una **referencia** (un puntero).

Ejemplo:
```
String s; // s es una referencia
```

- Cuando **creamos** la instancia usando **`new`**, se reserva **dinámicamente** un espacio en memoria para la misma y se asigna a la referencia la dirección de memoria del espacio reservado.

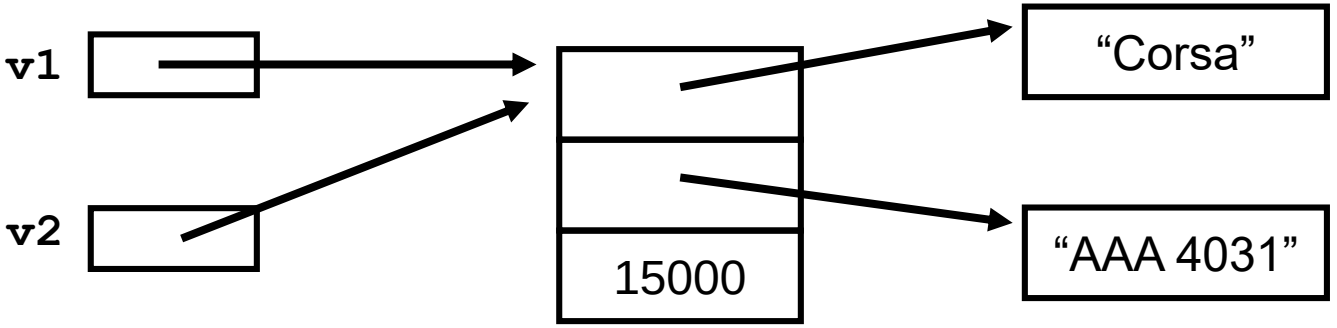
Ejemplo:
```
s = new String ("Hello");
        // s contiene ahora la dirección de un sector
        // de memoria donde está almacenado "Hello"
```

Manejo de Memoria al trabajar con Objetos

Referencias (continuación)

Otro Ejemplo:

```
Vehiculo v1, v2;
v1 = new Vehiculo ("Corsa","AAA 4031",15000);
v2 = v1;
```



Importante: Todos los objetos en Java son en realidad **referencias** a espacios de memoria pedidos en forma dinámica. Una **asignación** entre dos objetos copia la referencia, pero **no** la memoria referenciada.

Manejo de Memoria al trabajar con Objetos

Pasaje de Parámetros

- El pasaje de parámetros en Java es solamente **por valor**. Cuando invocamos a un método pasando un determinado parámetro efectivo, se crea una **copia** del mismo en el parámetro formal correspondiente.

- En el caso de los objetos, se crea una copia de la **referencia**, pero no de la memoria referenciada por ella.

- Esto significa que en Java **no** es posible escribir **procedimientos** que devuelvan varios resultados haciendo uso del pasaje por referencia como sí podíamos hacer en C++. En su lugar, debemos escribir varias funciones o bien escribir una función que devuelva un objeto conteniendo todos los resultados juntos. **No** podemos hacer procedimientos como éste en C++:

```
void sumProd (int a, int b, int &sum, int &prod)
```

Manejo de Memoria al trabajar con Objetos

Inicialización de variables

- A diferencia de C++ , Java **no** permite que ninguna variable sea utilizada antes de ser inicializada.
- Cuando se crea un objeto usando **`new`** ...

  1º) Se reserva dinámicamente un espacio en memoria para el mismo.

  2º) Se inicializan sus atributos con valores nulos:

`char` → **`'\u0000'`**		**`double`** → **`0.0`**	
`int` → **`0`**		**`boolean`** → **`false`**	
`long` → **`0L`**		**`referencias`** → **`null`**	

  3º) Se ejecuta el cuerpo del constructor invocado por **`new`**, que asigna a los atributos nuevos valores.

- A diferencia de los atributos, los parámetros formales y variables locales a métodos **no** son inicializados automáticamente. Deben ser inicializados **explícitamente** por el programador.

Packages

- Hasta el momento, para poder compilar y ejecutar todas las clases de una aplicación, teníamos que guardarlas juntas en un mismo directorio o carpeta.

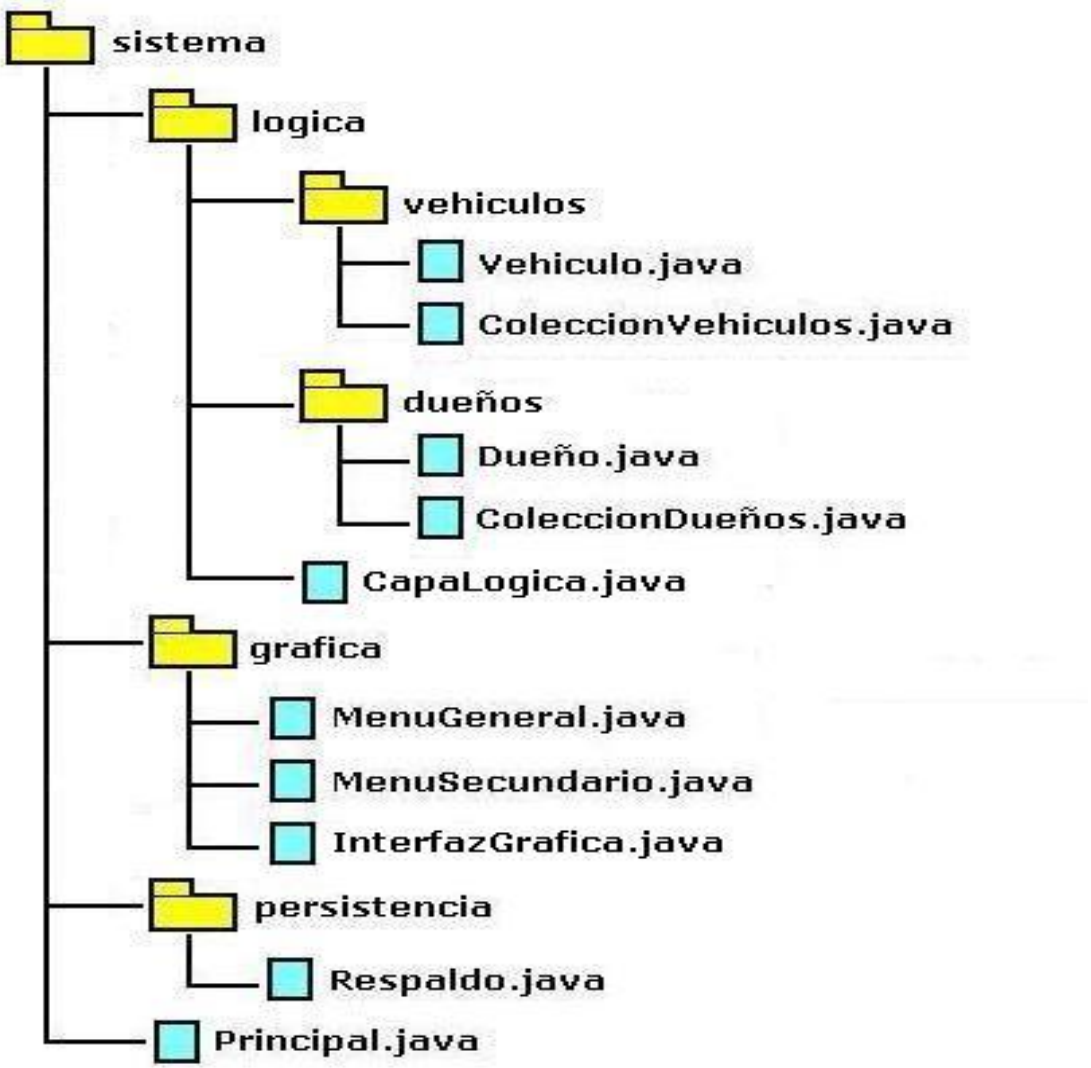
  Ejemplo: Considérense las siguientes clases para una aplicación hipotética. Todas ellas están almacenadas en la carpeta “sistema”.

Packages

- Si la aplicación tiene **muchas** clases, lo anterior puede resultar incómodo y desorganizado. Por esta razón, Java nos permite agrupar las clases en carpetas separadas.
- En Java, dichas carpetas se llaman ***packages*** (paquetes). Hacer una buena distribución en paquetes ayuda a organizar mejor el código y a agrupar las clases de un modo coherente.
- Una forma muy utilizada de agrupar clases en paquetes es considerando el ***principio de separación en capas***, el cual establece que:
  - Todas las clases de la ***Capa Lógica*** deben ir juntas
  - Todas las clases de la ***Capa Gráfica*** deben ir juntas
  - Todas las clases de la ***Capa de Persistencia*** deben ir juntas
  - Ninguna clase debe usar directamente las clases de las demás capas
- La idea entonces es hacer **un** paquete para **cada** capa. A su vez, dentro de cada paquete podemos agrupar las clases en sub-paquetes para una organización aún mejor.

Packages

Ejemplo:



Packages

- Para poder compilar y ejecutar las clases de una aplicación organizada en paquetes, debemos especificar dentro del código de cada clase lo siguiente:
  - El paquete al cual pertenece la clase
  - Las clases de otros paquetes que son utilizadas por la clase

Ejemplo: Veamos el código fuente de algunas clases del ejemplo anterior.

```
// Principal.java
package sistema;
import sistema.logica.CapaLogica;
import sistema.grafica.InterfazGrafica;
import sistema.persistencia.Respaldo;

public class Principal
{
        //ésta es la clase que contiene al main
        ...
}
```

Packages

Ejemplo (continuación):

```
// CapaLogica.java
package sistema.logica;
import sistema.logica.vehiculos.*;
import sistema.logica.dueños.*;

public class CapaLogica
{       ...
        ...
}

// Dueño.java
package sistema.logica.dueños;
import sistema.logica.vehiculos.Vehiculo;

public class Dueño
{       ...
        ...
}
```

Packages

Importando Clases predefinidas de Java

- Además de importar Clases de otros paquetes de nuestra aplicación, también podemos importar clases **predefinidas** de Java.

  Ejemplo:

```
// ColeccionDueños.java
package sistema.logica.dueños;
import sistema.logica.dueños.Dueño;
import java.util.Hashtable;

public class ColeccionDueños
{       ...
        ...
}
```

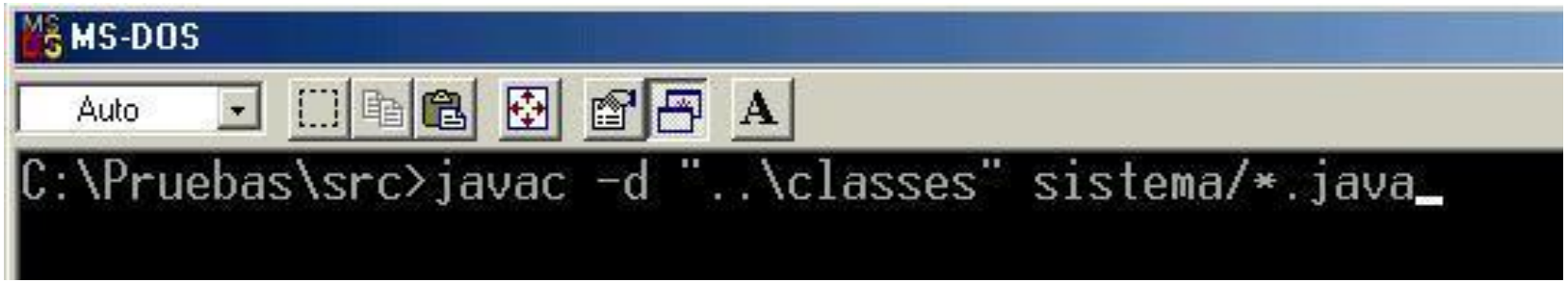
  Observación: Hay muchos paquetes predefinidos de java (**`java.lang`, `java.util`, `java.awt`, `javax.swing`**, etc.) Las únicas clases que **no** es necesario importar son las clases del paquete **`java.lang`**.

Compilando y Ejecutando Clases en Packages

Compilación (desde una Consola del Sistema Operativo)

- Nos posicionamos en la raíz de la estructura de paquetes y consideramos la ruta hacia la(s) clase(s) a compilar al momento de ejecutar el comando **javac**

  Ejemplo: Supongamos que la estructura de paquetes vista en los ejemplos anteriores se encuentra contenida dentro de la carpeta **src**.



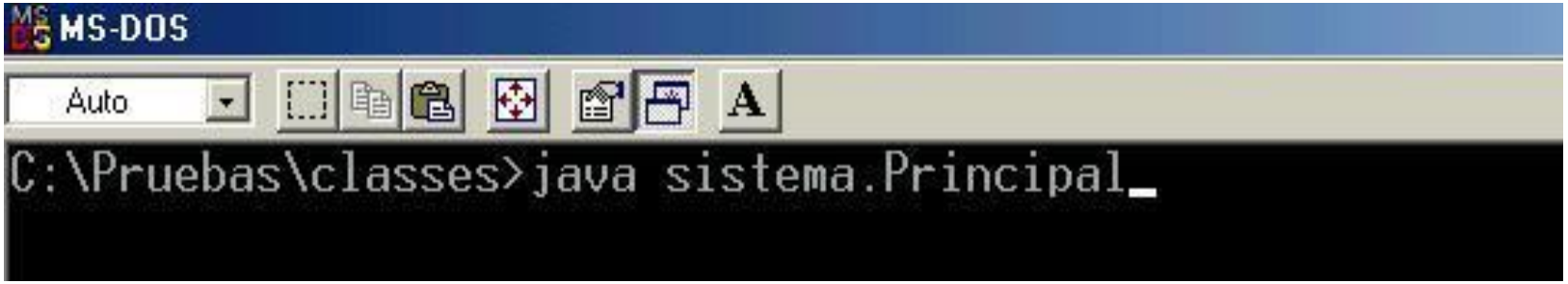
  El comando **javac** genera (dentro de la carpeta **classes**) una estructura de paquetes idéntica a la contenida dentro de la carpeta **src**. Allí guarda a todos los (**.**class) respetando la misma organización que tenían los (**.**java) dentro de la carpeta **src**.

Compilando y Ejecutando Clases en Packages

Ejecución (desde una Consola del Sistema Operativo)

- Ahora nos posicionamos en la raíz de la estructura de paquetes que contiene a todos los (**.**class) para ejecutar la clase que contiene al **main**.

  <u>Ejemplo</u>: Vamos a ejecutar la clase Principal vista antes. Al igual que en la compilación, debemos tener en cuenta la ruta hacia dicha clase.



Se acostumbra que los nombres de los packages sigan la misma convención de nombres que las variables y métodos. Es decir, empiezan con minúscula y utilizan mayúsculas para simular un cambio de palabra dentro del nombre.

Compilando y Ejecutando Clases en Packages

Desde un entorno de desarrollo

- Generalmente, en los entornos de desarrollo específicos para Java no suele haber diferencia entre la forma de compilar clases agrupadas en packages y clases que no lo están.

- Esto es porque al utilizar **proyectos**, son ellos quienes se encargan de (internamente) ejecutar los comandos **javac** y **java** con la sintaxis apropiada.

- Concretamente, en **Eclipse** la compilación y ejecución de Clases agrupadas en packages se hace de forma análoga a como se explicó en el capítulo pasado para Clases simples.

Herencia

- Es un mecanismo que permite crear una nueva clase a partir de otra ya existente. Esta nueva clase (clase **derivada**) posee todos los atributos y métodos de la clase original (clase **base**) y puede definir otros nuevos, propios de ella.
- Java es un lenguaje que hace un fuerte uso de la Herencia. Muchas bibliotecas predefinidas de Java implementan enormes jerarquías de clases.
- Para implementar Herencia, se utiliza la palabra reservada **`extends`**

Ejemplo: Recordemos la clase **Vehiculo**:

```
// Vehiculo.java
public class Vehiculo
{
        private String modelo;
        private String matricula;
        private double precio;
        ...
}
```

Herencia

Hagamos algunas clases derivadas de **Vehiculo**:

```
// Auto.java
public class Auto extends Vehiculo
{       private String tipoMotor;
        private int cantPuertas;
        ...
}

// Camion.java
public class Camion extends Vehiculo
{       private long capCarga;
        private double potencia;
        ...
}

// ConRemolque.java
public class ConRemolque extends Camion
{       private long capRemolque;
        ...
}
```

Herencia

- Cada clase hereda todos los atributos y métodos de su clase base correspondiente, (así como todos los de aquellas clases anteriores en la jerarquía).

- Sin embargo, una clase derivada no puede acceder directamente a los atributos **`private`** de sus clases anteriores en la jerarquía. Debe hacerlo a través de los métodos, etiquetados como **`public`**.

- Otra alternativa posible (poco recomendable en términos de diseño, salvo en casos especiales) es etiquetar los atributos de la clase base como **`protected`**. Esto permite que puedan ser accedidos directamente tanto desde la clase base como desde sus clases derivadas.

- Existe en Java una clase predefinida llamada **`Object`** que es implícitamente extendida por toda clase en Java. Por tanto, aunque no se ponga **`extends`** en una clase, igualmente será derivada de **`Object`**

  Referencia: API - **`java.lang.Object`**

Herencia

Constructores de Clases derivadas

- Cuando se extiende una clase, se heredan los atributos y métodos de su clase base correspondiente, pero **no** se heredan los constructores.
- Por lo tanto, al definir un constructor para una clase derivada, se debe invocar explícitamente a algún constructor de su clase base. Dicha invocación se realiza mediante el uso de la palabra reservada **`super`**.

Ejemplo: Veamos un constructor para la clase **Auto**.

```
public Auto (String mod,String mat,double pre,String tm,int cp)
{    super (mod,mat,pre); // invoco al constructor de Vehiculo
     tipoMotor = tm;
     cantPuertas = cp;
}
```

Observación: La invocación a algún constructor de la clase base debe ser lo primero que se haga dentro del constructor de la clase derivada. De otro modo, el compilador produce un error.

Herencia

Sobre-escritura de métodos

- En ocasiones, se desea **redefinir** el comportamiento de un cierto método de una clase base en una clase derivada de ella.
- Para ello, la clase derivada puede **sobreescribir** dicho método. Esto se logra incluyendo en la clase derivada un nuevo método con el mismo nombre, parámetros y tipo de retorno, pero con diferente cuerpo.

Ejemplo:

```
public class Camion extends Vehiculo
{    ...
     // Método propio de la Clase Camión. NO fue
     // heredado de la Clase Vehiculo.
     public double cargaSemanal (int cantViajes)
     {  return capCarga * cantViajes;  }
     ...
}
```

Herencia

Sobre-escritura de métodos (continuación)

Ejemplo (continuación):

```
public class ConRemolque extends Camion
{    ...
     // Sobre-escribimos el método "cargaSemanal"
     // heredado de la Clase Camion.
     public double cargaSemanal (int cantViajes)
     {
        return (getCapCarga() + capRemolque) * cantViajes;
     }
     ...
}
```

- En ocasiones, no se desea sobre-escribir completamente un método de una clase base, sino **agregarle** comportamiento en la clase derivada. Esto puede lograrse haciendo que el método redefinido **invoque** al método original correspondiente en la clase base, mediante el uso de **`super`**.

Herencia

Sobre-escritura de métodos (continuación)

Ejemplo: Extendemos el comportamiento del método **toString()**

```
public class Vehiculo
{    ...
     public String toString ()
     {  return ("\n Modelo:" + modelo + "\n Matricula:" +
                matricula + "\n Precio:" + precio);
     }
}

public class Auto extends Vehiculo
{    ...
     public String toString ()
     {  return (super.toString() + "\n Tipo de Motor:" +
                tipoMotor + "\n Puertas:" + cantPuertas);
     }
}
```

Herencia

Clases Abstractas

- En ocasiones, se desea crear una clase que representa algún concepto fundamental al diseño del problema, pero no se desea **instanciar** dicha clase. Lo que se desea es implementar muchas de las funcionalidades de dicha clase en sus clases derivadas e instanciar éstas últimas.
- Este tipo de clase se conoce como clase **abstracta** y se implementa en Java mediante el uso de la palabra reservada `abstract` .

  Ejemplo: Supóngase un contexto en el cual ya no es de interés instanciar la clase `Vehiculo` . Podemos declararla abstracta simplemente agregando `abstract` en su encabezado, y manteniendo su implementación.

```
public abstract class Vehiculo ...
public class Auto extends Vehiculo ...
public class Camion extends Vehiculo ...
public class ConRemolque extends Camion ...
```

Herencia

Métodos Abstractos

- Una clase abstracta puede contener atributos y métodos de igual forma que una clase no abstracta. Pero con frecuencia se desean definir métodos en una clase abstracta cuya implementación sea realizada exclusivamente en sus clases derivadas.
- Dichos métodos se conocen como métodos **abstractos** . Se definen (**sin** implementación) dentro de clases abstractas etiquetados como **`abstract`**

Ejemplo:

```
public abstract class Vehiculo
{
        ...
        public abstract int cantidadRuedas();
        // será implementado en las Clases derivadas
        ...
}
```

Herencia

Métodos Abstractos (continuación)

Ejemplo (continuación):

```
public class Auto extends Vehiculo
{       ...
        public int cantidadRuedas()
        {  return 4;  }
}

public class Camion extends Vehiculo
{       ...
        public int cantidadRuedas()
        {  return 10;  }
}
```

Observación: Si se extiende una clase abstracta con algún método abstracto, se **debe** implementar dicho método en la clase derivada. Sino, el compilador da un error. No pueden haber métodos abstractos en Clases que **no** son abstractas

Polimorfismo

- Es una propiedad que permite tratar un objeto de una clase derivada como a cualquier objeto de su clase base. Cualquier operación aplicable a un objeto de la clase base es también aplicable a objetos de sus clases derivadas.
- El polimorfismo permite asignar a una referencia de clase base una instancia de una clase derivada.

Ejemplo:

```
Vehiculo v1 = new Auto ("Corsa", "AAA 4031", 15000,
                        "Nafta",5);
Vehiculo v2 = new Camion ("Scania", "SAP 3047", 25000,
                          550,320.5);
```

¿ Son correctas las siguientes invocaciones ?

```
double pre1 = v1.getPrecio();
double pre2 = v2.getPrecio();
```

Polimorfismo

- Al asignar a una referencia de clase base una instancia de clase derivada, sólo podemos acceder a los miembros de dicha instancia **que estén definidos en la clase base**.

  Ejemplo:

```
Vehiculo v1 = new Camion(...);
double carga1 = v1.cargaSemanal(17) ;
// Incorrecto !!

Camion v2 = new Camion(...);
double carga2 = v2.cargaSemanal(17) ;
// Correcto !!
```

  El método **`cargaSemanal`** fue definido por primera vez dentro de **`Camion`** , **no** fue heredado de **`Vehiculo`**. Desde el punto de vista del compilador, **`v1`** es un **`Vehiculo`**, **no** un **`Camion`**. En la Clase **`Vehiculo`** **no** hay ningún método llamado **`cargaSemanal`**.

Polimorfismo

Métodos Polimórficos

- En C++, para que un método sobre-escrito se comportase polimórficamente, teníamos que etiquetarlo como **virtual** en la Clase base. Sin embargo, en Java todo método es **implícitamente** virtual.

- Lo anterior ocurre porque a diferencia de C++, Java implementa **únicamente** la propiedad de **dynamic binding** (binding dinámico). Esto significa que la decisión de cuál método ejecutar al trabajar con métodos sobre-escritos se realiza **siempre** en tiempo de ejecución.

- Por esta razón es que **siempre** se ejecuta el método adecuado, o sea el que está implementado en la instancia referenciada, sea una instancia de la clase base (si ésta es instanciable) o de cualquier clase derivada.

Polimorfismo

Métodos Polimórficos (continuación)

Ejemplo:

```
Vehiculo v1 = new Auto ("Corsa", "AAA 4031", 15000,
                                    "Nafta", 5);
Vehiculo v2 = new Camion ("Scania", "SAP 3047", 25000,
                                550, 320.5);
Vehiculo v3 = new ConRemolque("Scania", "HAA 312", 35000,
                                    550, 320.5, 750);

int cr1 = v1.cantidadRuedas(); // el de Auto
int cr2 = v2.cantidadRuedas(); // el de Camion
int cr3 = v3.cantidadRuedas(); // el de ConRemolque
```

Observación: El método **`cantidadRuedas`** había sido definido **abstracto** en la clase **`Vehiculo`** e implementado en cada una de las clases derivadas.

Interfaces

- Una interface es una variante a la noción de clase abstracta. Se trata de un tipo especial de clase que **no** posee atributos y en la cual todos sus métodos son **implícitamente abstractos**.

Ejemplo:

```
public interface Dibujo
{
  // No hay atributos, sólo cabezales de métodos

  public void dibujarPunto (int x, int y);
  public void dibujarLinea (int x1, int y1, int x2, int y2);
  public void dibujarCirculo (int x, int y, double radio);
}
```

Interfaces

- Las interfaces son creadas con el fin de que otras Clases implementen los métodos definidos en ellas

Ejemplo:

```
public class Grafico2D implements Dibujo
{
  // ... Atributos propios de la Clase ...

  // ... Métodos propios de la Clase ...

  // Ahora implementamos los métodos de la interface

  public void dibujarPunto (int x, int y)
  { ... }

  public void dibujarLinea (int x1, int y1, int x2, int y2)
  { ... }

  public void dibujarCirculo (int x, int y, double radio)
  { ... }
}
```

Interfaces

- Si una clase implementa una interface, está obligada a implementar **todos** los métodos especificados en dicha interface. En caso contrario, el compilador produce un error.
- **Extender** una clase abstracta e **Implementar** una interface son cosas muy parecidas. En ambos casos estamos dando comportamiento a métodos originalmente **abstractos**.
- De hecho, se puede dar a las interfaces el mismo tratamiento que se les da a las Clases abstractas.

Ejemplo:

```
public static void main (String args[])
{
   Dibujo d = new Grafico ();
   d.dibujarLinea (1,1,5,5);
   d.dibujarCirculo (3,3,7.5);
}
```

Casting e instanceof

Casting en tipos primitivos

- Consideremos las siguientes declaraciones:

```
int x = 5;
double y = 3.45;
```

- Consideremos además las siguientes asignaciones:

```
y = x;
x = y;
```

  - La primer asignación es válida, puesto que **`x`** es un **`int`**. Como todo entero es también un real, se lo puede asignar a **`y`** que es un **`double`**.
  - La segunda asignación **no** es válida, porque **`y`** vale 3.45 y **`no`** es un entero.

- Hay ocasiones en las cuales se desea asignar a una variable un valor cuyo tipo **no** es compatible con el de la variable. Esto puede lograrse haciendo un **casting** al momento de realizar la asignación.

```
x = (int) y;
```

Casting e instanceof

Casting en Objetos

- Hay circunstancias en las cuales se tiene una referencia correspondiente a una cierta clase base, pero se sabe que en realidad está referenciando a un objeto de una clase derivada.

```
Vehiculo ve = new Auto ("Corsa","AAA 403",15000,"Nafta",5);
```

- En el contexto anterior, podríamos querer invocar a un método específico de la clase derivada. Para el ejemplo anterior haríamos así:

```
String tipoMotor = ve.getTipoMotor();
```

- La asignación anterior es **incorrecta** pues desde el punto de vista del compilador `ve` es un `Vehiculo`, **no** un `Auto`. Por lo tanto **no** posee un método `getTipoMotor()`, el cual es específico de la Clase `Auto`. Este problema puede resolverse mediante un **casting**:

```
String tipoMotor = ((Auto) ve).getTipoMotor();
// Le indico al compilador que en tiempo de ejecución,
// ve es en realidad un Auto.
```

Casting e instanceof

Casting en Objetos (continuación)

- En el ejemplo anterior era claro que el objeto referenciado pertenecía a una Clase derivada. Sin embargo, esto **no** siempre es evidente en el código, y puede provocar **errores en tiempo de ejecución**.
- Para evitar confundir el tipo de los Objetos en tiempo de ejecución, Java posee un operador especial llamado **`instanceof`** que permite verificar el tipo de un Objeto en forma previa a realizarle un **casting**.

Ejemplo:

```
Vehiculo ve = new Auto ("Corsa","AAA 403",15000,"Nafta",5);

if (ve instanceof Vehiculo)
{    String tipoMotor = ((Auto) ve).getTipoMotor();
     ...
}
```

Observación: Usamos casting e **`instanceof`** **solamente** cuando queremos acceder específicamente a miembros que **no** están definidos en la Clase base.

Arreglos en Java

- Al igual que en C++, son estructuras de datos que permiten almacenar (en forma secuencial) una secuencia acotada de elementos. Sus celdas pueden contener tanto valores de tipos primitivos como Objetos.

- Siempre son dinámicos, en el sentido de que su tamaño es definido en tiempo de ejecución y, una vez determinado, **no** puede modificarse.

- A su vez, los arreglos en Java **son** objetos en sí mismos, sin importar que sus celdas contengan otros objetos o valores de tipos primitivos. Al ser objetos, cuando los declaramos no hacemos más que declarar una **referencia** , para la cual luego se asignará memoria en forma dinámica.

- Tanto la sintaxis de los arreglos como la manera de manipularlos son muy similares a C++.

Arreglos en Java

Creación y manipulación

- Los arreglos en Java se declaran de la siguiente manera:

```
tipoCelda nombreArreglo [];
```

- **`nombreArreglo`** es simplemente una **referencia**. El tamaño del arreglo aún no ha sido determinado. Ello puede hacerse de dos formas:
  - mediante el uso de la palabra reservada **`new`**.
  - mediante el uso de llaves en la propia declaración

Ejemplo:

```
int tam = (int) (Math.random() * 10);
double miArreglo [] = new double [tam];
int otroArreglo [] = {9, 17, 5, 10, 0};
```

- Los arreglos en Java se indizan desde 0 hasta (tamaño - 1) y **una vez creados** se accede a sus celdas usando la misma sintaxis que en C++. Si no son inicializados explícitamente, sus celdas se cargan con valores **nulos**.

Arreglos en Java

Creación y manipulación (continuación)

- Al igual que cualquier objeto, un arreglo puede ser un **atributo** de una Clase:

Ejemplo:

```
public class SecuenciaReales
{
   private double arre [];

   public SecuenciaReales (int tam)
   {  arre = new double [tam] ; }
   ...
   public void sumarUnoATodos ()
   {
       for (int i = 0 ; i < arre.length ; i++)
          arre[i] = arre[i] + 1;
   }
}
```

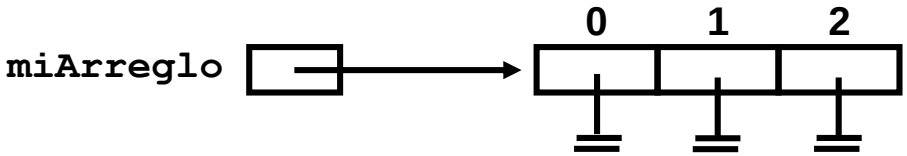
Observación: Todo arreglo en Java posee un atributo **`length`** que indica su tamaño Si se accede a una celda más allá del mismo, ocurre un error en tiempo de ejecución.

Arreglos en Java

Arreglos de Objetos

- Se declaran y se crean de igual forma que los arreglos de tipos primitivos. Sin embargo, la creación de un arreglo de Objetos **no** construye los objetos a almacenar en el mismo. Sólo asigna en sus celdas referencias a **`null`**.

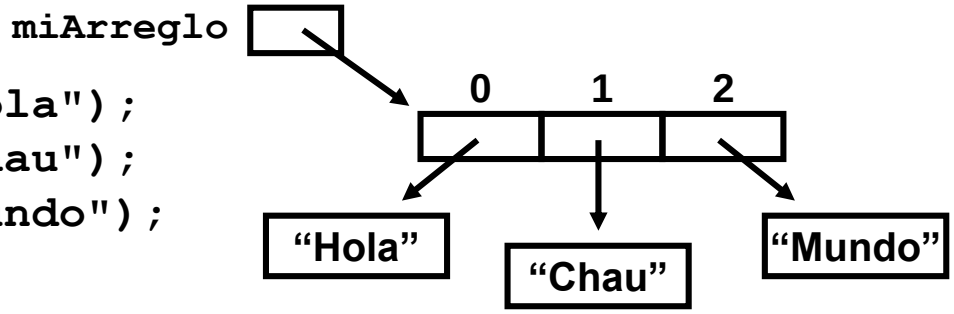
Ejemplo: `String miArreglo [] = new String[3];`



- Luego cada celda deberá ser instanciada por separado.

Ejemplo:

```
miArreglo[0] = new String("Hola");
miArreglo[1] = new String("Chau");
miArreglo[2] = new String("Mundo");
```



Arreglos en Java

Arreglos polimórficos

- Son arreglos cuyas celdas contienen referencias correspondientes a una determinada clase base. Al instanciar cada celda, se la puede crear como un objeto de la clase base (si la misma es instanciable) o como un objeto de cualquier clase derivada. Luego se puede recorrer el arreglo e invocar algún método **polimórfico** sobre los Objetos almacenados en el mismo.

Ejemplo:

```
Vehiculo arre [] = new Vehiculo[3] ;
arre[0] = new Auto ("Corsa","AAA 4031",15000,"Nafta",5);
arre[1] = new Camion ("Scania","SAP 3047",25000,550,320.5);
arre[2] = new ConRemolque ("Scania","HAB 3121",35000,550,
                           320.5,750) ;

int max = arre[0].cantidadRuedas();
for (int i = 1; i < arre.length; i++)
     if (arre[i].cantidadRuedas() > max)
            max = arre[i].cantidadRuedas();
```