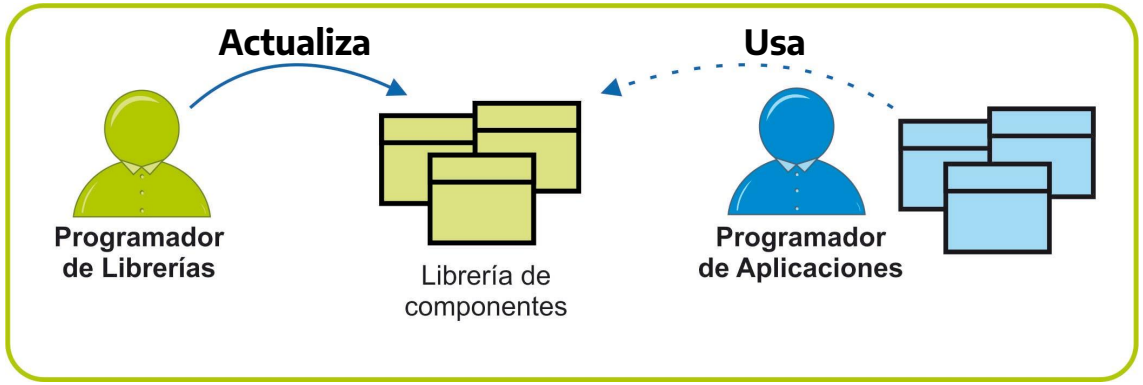
Taller de Lenguajes II

- **Paquetes**
  - Definición de paquetes
  - Archivos JAR
- **Clases abstractas y métodos abstractos**
  - Definición
  - La palabra clave abstract
- **Polimorfismo**
  - Definición
  - Binding Dinámico

¿Qué podría pasar si se modifica una librería de clases que está siendo usada por otros programadores?



El código podría romperse !!

El **programador de la librería** debe sentirse libre para **mejorar el código** y el **programador de aplicaciones** debería poder **aplicar las mejores** sin necesidad de reescribir su código.

¿Cómo se asegura esto?

(1) **Garantizando compatibilidad con versiones previas**: no quitar métodos existentes en la versión previa.

(2) **Usando especificadores de acceso** para indicarle al programador de aplicaciones qué está disponible y qué no.

Antes de entrar en especificadores de acceso, falta responder una pregunta útil en este contexto: **¿cómo se crea una librería de clases en java?**

Librería de componentes
Paquetes JAVA

- En Java **una librería de componentes** (clases e interfaces) es un agrupamiento de archivos `.class`, también llamado **paquete.**
- Para **agrupar componentes** en un paquete, debemos anteponer la palabra clave **`package`** junto con el nombre del paquete al comienzo del archivo fuente de cada una de las clases o interfaces.

```
package graficos;
public class Rectangulo {
  //código JAVA
}
```

```
package graficos;
public interface Centrable {
  //código JAVA
}
```

La clase **Rectangulo** y la interface **Centrable** pertenecen al paquete **graficos**

- Las clases e interfaces que se crean **sin usar la sentencia `package`** se ubican en un paquete sin nombre, llamado ***default package***.

```
public class HolaMundo {
      //código JAVA
}
```

La clase **HolaMundo** pertenece al **paquete por defecto**

Usar paquetes propios o el *default package* ¿Qué opinan? ¿por qué?

Librería de componentes

Paquetes JAVA

El **nombre completo de la clase o nombre canónico** contiene el nombre del paquete.

```
graficos.Rectangulo
java.util.Arrays
```

Nombre completo de la clase Rectangle

Nombre completo de la clase Arrays

Para usar la clase **Rectangulo** se debe usar la palabra clave **`import`** o **especificar el nombre completo** de la clase:

```
package ar.edu.unlp.taller2;
import graficos.Rectangulo;
//import graficos.*;
class Figuras {
 Rectangulo r = new Rectangulo();
}
```

```
package ar.edu.unlp.taller2;

class Figuras {
 graficos.Rectangle r;
 r = new graficos.Rectangle();
}
```

Importación por demanda: se tienen disponibles todos los nombres de clases e interfaces del paquete

La sentencia **`import`** permite usar el **nombre corto de la clase** en todo el código fuente. Si no se usa el **`import`** se debe especificar el **nombre completo** de la clase.

Librería de componentes

Paquetes JAVA

¿Qué sucede si se crean 2 clases con el mismo nombre?

Supongamos que 2 programadores escriben una clase de nombre **Vector** en el paquete *default,* **se plantea un conflicto de nombres**.

Es necesario **crear nombres únicos: usamos paquetes.**

```
package util;
public class Vector {
   //código JAVA
}
```

```
package taller2.estructuras;
public class Vector {
    //código JAVA
}
```

¿Qué sucede si se importan dos librerías que incluyen el mismo nombre de clase?

```
import taller2.estructuras.*;
import util.*;

Vector vec1 = new Vector();
```

Ambos paquetes contienen la clase Vector

Colisión! ¿A qué clase hace referencia?: el compilador no puede determinarlo

Librería de componentes

Paquetes JAVA

```
package util;
public class Vector {
   //código JAVA
}
```

```
package taller2.estructuras;
public class Vector {
    //código JAVA
}
```

¿Qué sucede si se importan dos librerías que incluyen el mismo nombre de clase?

```
import taller2.estructuras.*;
import util.*;

Vector vec1 = new Vector();
```

Ambos paquetes contienen la clase Vector

Colisión! ¿A qué clase hace referencia?: el compilador no puede determinarlo

Una posible solución:

```
import util.*;
Vector vec1 = new Vector();

taller2.estructuras.Vector vec2 = new taller2.estructuras.Vector();
```

Usamos los dos nombres canónicos o combinamos

Librería de componentes

Paquetes JAVA

- Las clases e interfaces que son parte de la distribución estándar de JAVA están agrupadas en paquetes de acuerdo a su funcionalidad. Algunos paquetes son:

`java.lang`	clases básicas para crear aplicaciones.
`java.util`	librería de utilitarios, colecciones.
`java.io`	manejo de entrada/salida.
`java.awt/javax.swing`	manejo de GUI (Graphic User Interface).

- Los únicos paquetes que se importan automáticamente es decir no requieren usar la sentencia `import` son el paquete `java.lang` y el **paquete actual** (paquete en el que estamos trabajando).

```
String s" "hola";
System.out.print("hola");
```

String y **System** son clase de java.lang, se **pueden usar directamente**

```
package taller2.estructuras;
public class Vector {
  //código JAVA
}
```

Recomendación: usar como primera parte del **nombre del paquete el nombre invertido del dominio de Internet** y así evitar conflicto de nombres. Usar **minúscula para nombres de paquetes** e **inicial mayúscula para nombres de clases**.

Ejemplo: `ar.edu.unlp.graficos`

Librería de componentes

Paquetes JAVA

Un paquete normalmente está formado por varios archivos **.class**.
Java se beneficia de la **estructura jerárquica de directorios del sistema operativo** y ubica todos los **.class** de un mismo paquete en un mismo directorio. De esta manera, se resuelve:

- el nombre único del paquete
- la búsqueda de los **.class** (que de otra forma estarían diseminados en el disco)

```
package ar.edu.unlp.utiles;
public class Vector {
    //código JAVA
}
```

\ar\edu\unlp\utiles\Vector.class

Cuando el "intérprete" JAVA ejecuta un programa y necesita localizar dinámicamente un archivo **.class**, por ej. cuando se crea un objeto o se accede a un miembro **static**, procede de la siguiente manera:

- Busca en los **directorios estándares del JRE**
- Busca en el directorio actual (paquete de la clase que se está ejecutando)
- Recupera la variable de entorno **CLASSPATH**, que contiene la lista de directorios usados como raíces para buscar los archivos **.class**. Comenzando en la raíz, el intérprete toma el nombre del paquete (de las sentencias **import**) y reemplaza cada **"."** por una barra **"\"** o **"/"** (según el SO) para generar un camino donde encontrar las clases a partir de las entradas del **CLASSPATH**.

Librería de componentes

Paquetes JAVA

Consideremos el dominio **unlp.edu.ar** invertido y obtenemos un nombre de dominio único y global: **ar.edu.unlp**. Si creamos una librería **utiles** con las clases **Vector** y **List**, tendríamos:

```
package ar.edu.unlp.utiles;
public class Vector {
   //código JAVA
}
```

```
package ar.edu.unlp.utiles;
public class List {
   //código JAVA
}
```

Supongamos que a ambos archivos los guardamos en el directorio **c:\tallerjava**

```
C:\tallerjava\ar\edu\unlp\utiles\Vector.class
C:\tallerjava\ar\edu\unlp\utiles\List.class
```

El **"intérprete" JAVA** comienza a **buscar el paquete ar.edu.unlp a** partir de alguna de las **entradas** indicadas en la variable de entorno **CLASSPATH**:

```
CLASSPATH=.;c:\tallerjava;c:\java\librerias
```

Esta variable puede contener muchas entradas separadas por ";"

Paquetes en JAVA
Formato JAR

Es posible agrupar archivos **.class** pertenecientes a uno o más paquetes en un único archivo con extensión **jar (Java ARchive)**. El formato **JAR** usa el formato **zip**. Los archivos JAR son multi-plataforma, es estándar. Es posible incluir además de archivos .class, archivos de imágenes y audio, recursos en general, etc.

El JSE tiene una herramienta para crear archivos JAR, desde la línea de comando, es el utilitario **jar**.

Por ejemplo: si se ejecuta el comando **jar** desde el directorio donde están los archivos **.class** podríamos ponerlo así:

```
c:\tallerjava\ar\edu\unlp\utiles\jar cf utiles.jar *.class
```

- En este caso, en el **CLASSPATH** se especifica el nombre del archivo jar:

```
CLASSPATH=.;c:\utiles.jar;c:\java\librerias
```

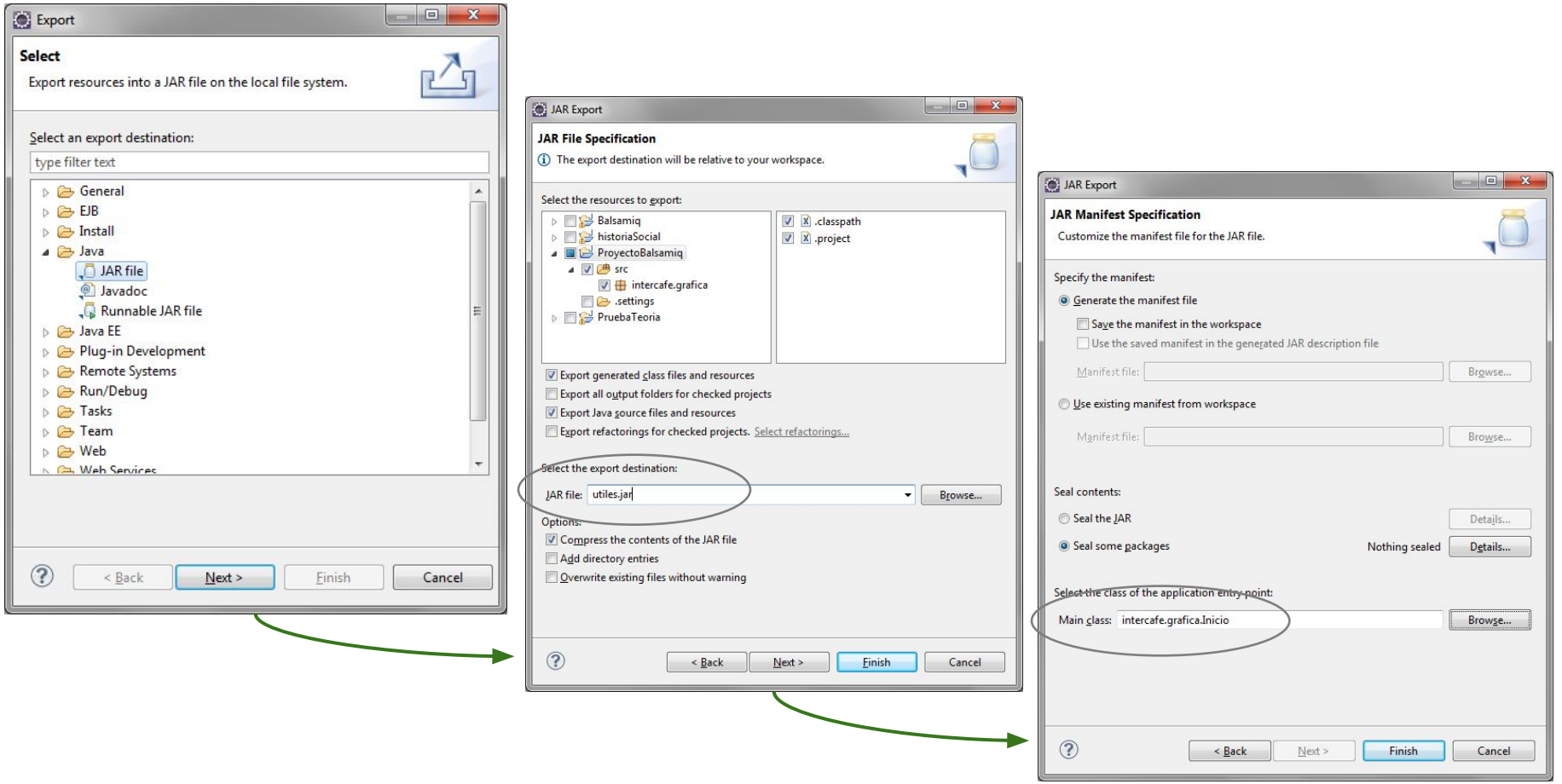
Los archivos jar pueden ubicarse en cualquier lugar del disco

- El "intérprete" JAVA se encarga de buscar, descomprimir, cargar e interpretar estos archivos.

Paquetes en JAVA

Formato JAR

El archivo JAR también puede construirse desde un proyecto Eclipse, con al opción **`export`**.



Paquetes en JAVA

El formato JAR

Los archivos JAR contienen todos los paquetes con sus archivos .class, los recursos de la aplicación y un archivo **MANIFEST.MF** ubicado en el camino META-INF/MANIFEST.MF, cuyo **propósito es indicar cómo se usa el archivo JAR**.

Las **aplicaciones de escritorio** a diferencia de las librerías de componentes o utilitarias, **requieren que el archivo MANIFEST.MF** contenga una **entrada con el nombre de la clase** que actuará como punto de entrada de la aplicación (la clase que contiene método main).

Para especificar la **clase "principal"**, el archivo MANIFEST.MF debe contener la **entrada Main-Class**.

```
Manifest-Version: 1.0
Created-By: 1.6.0_12 (Sun Microsystems Inc.)
Main-Class: capitulo4.paquetes.TestOut.class
```

Clases Concretas & Abstractas

- En JAVA podemos definir **clases concretas** y **clases abstractas**.
- A partir de una **clase concreta** se crean instancias. Todos los métodos de una clase concreta tienen implementación (cuerpo).
- Una **clase abstracta** NO puede ser instanciada, sólo puede ser extendida. Sus métodos pueden ser abstractos y concretos.
- Una clase abstracta es ideal para representar un concepto, no es apropiada para crear instancias.

Clases Abstractas

Una clase abstracta representa un **concepto abstracto** que no es instanciable, expresa la **interfaz de comportamiento de un objeto** y **NO una implementación** particular. Definen un comportamiento común para todos los objetos de las subclases concretas

Se espera que las **clases abstractas** sean extendidas por **clases concretas**.

Una clase abstracta puede contener **métodos abstractos y métodos concretos**.

¿Qué es un método abstracto? Es un método que NO tiene implementación. Se define el nombre, el tipo de retorno, la lista de argumentos, termina con ";". Se antepone la palabra clave `abstract` al tipo de datos de retorno/void.

```
abstract void dibujar();
```

Solo las clases y métodos abstractos llevan la palabra clave abstract, no pueden llevar el modificador abstract:

- los constructores
- los métodos estáticos
- los métodos privados

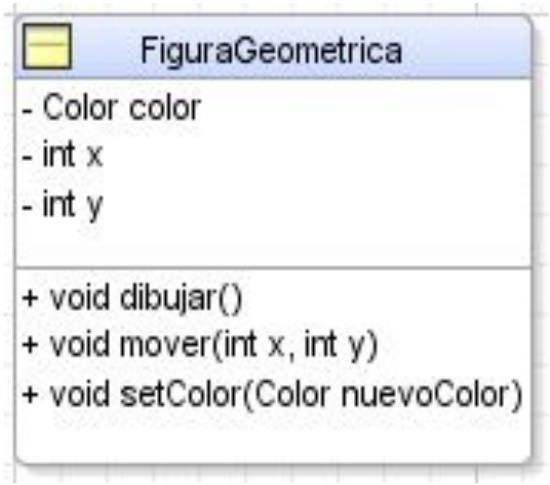
Clases Abstractas

En JAVA podemos modelar un **CONCEPTO** mediante una **clase abstracta**

Pensemos en una aplicación que manipula figuras geométricas en la pantalla. Podríamos dibujar **círculos**, **rectángulos**, **triángulos**, **líneas rectas**, etc.

Todas las figuras geométricas pueden **cambiar de color**, **dibujarse** en la pantalla, **moverse**, calcular la **superficie**, el **perímetro**, etc., pero cada una lo hace de una manera particular.

Una figura geométrica es un concepto abstracto: representa el comportamiento común de las todas las figuras geométricas, sin embargo no es posible dibujar o redimensionar una figura geométrica genérica, sólo sabemos que todas las figuras geométricas concretas, como los círculos, rectángulos, triángulos tienen esas capacidades.



Clases Abstractas

```
public class FiguraGeometrica {
   private Color color;
   private int x;
   private int y;

   public  void mover(int x, int y){
     this.x = x;
     this.y = y;
   }
   public void setColor(Color nuevoColor) {
     color = nuevoColor;
     dibujar();
   }
   public void dibujar() {
       ??
   }

   public int area(){
      ??
   }
}
```

La clase **FiguraGeometrica** NO representa una figura real y por lo tanto NO puede definir implementaciones para todos sus métodos.
¿Qué hacemos? La declaramos abstracta

Clases Abstractas

Para declarar una **clase abstracta** se antepone la palabra clave **abstract** a la palabra clave **class.** Una clase abstracta es una clase que solamente puede ser EXTENDIDA, no puede ser INSTANCIADA. El compilador garantiza esta característica.

```
public abstract class FiguraGeometrica {
 private Color color;
 private int x;
 private int y;

 public  void mover(int x, int y){
  this.x=x;
  this.y=y;
 }
 public void setColor(Color nuevoColor){
  color = nuevoColor;
  dibujar();
 }
 public abstract void dibujar();
 public abstract void area();
}
```

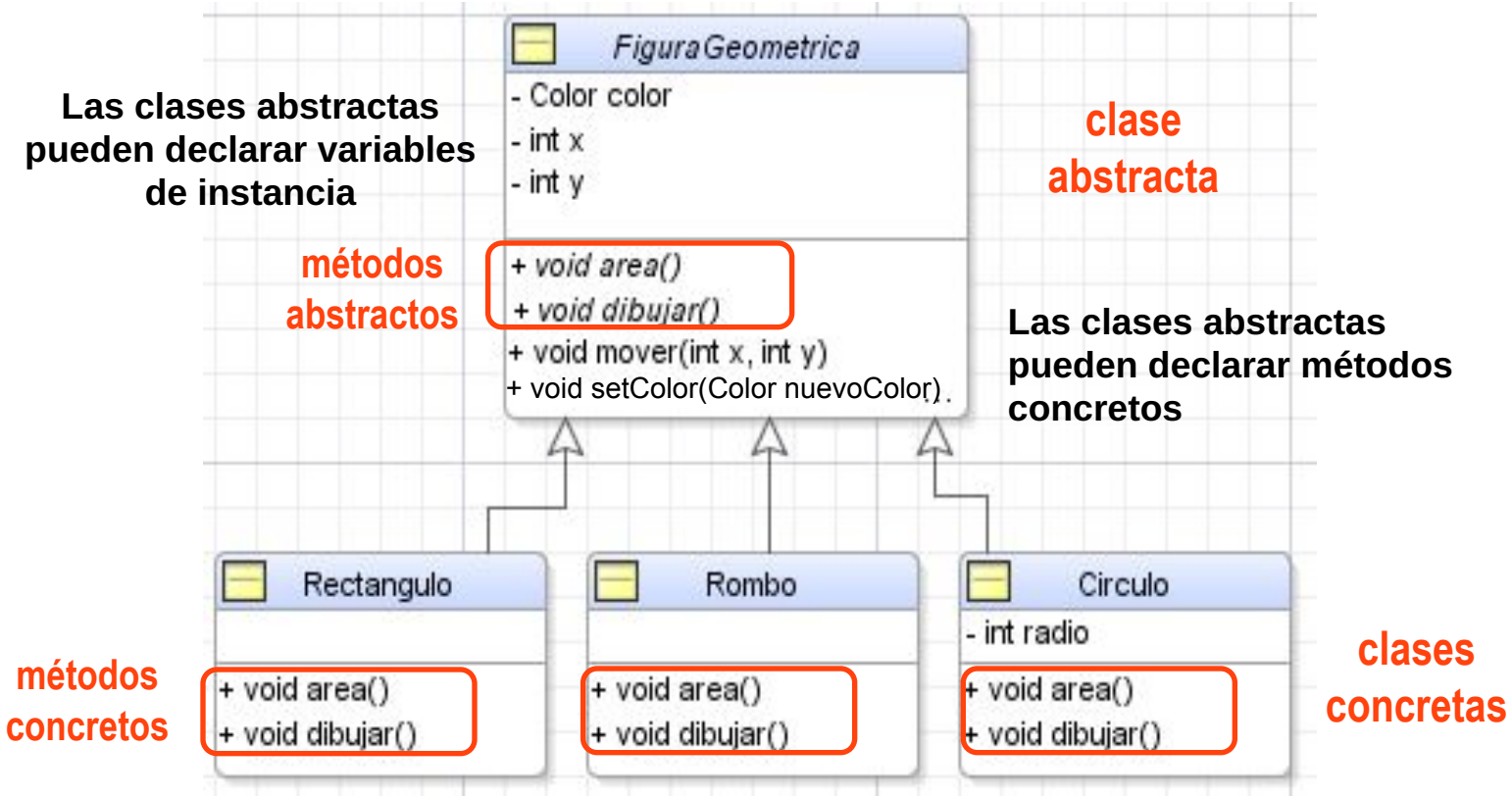
Los métodos abstractos no tiene cuerpo, su declaración termina con ";"

```
public class TestFiguras {
 public static void main(String args[]){
        new FigurasGeometricas();
    }
}
```

Si se intenta crear objetos de una **clase abstracta**, fallará la compilación. El compilador NO permite crear instancias de clases abstractas.

Una **clase abstracta** puede contener **métodos abstractos** y **métodos concretos**.

Clases Abstractas y Herencia



FiguraGeometrica es una clase abstracta y los métodos **area()** y **dibujar()** son abstractos.

Para que las subclases **Rectangulo**, **Rombo** y **Circulo** sean concretas, deben proveer una implementación de cada uno de los método abstractos de la clase **FiguraGeometrica.**

Las clase **FiguraGeometrica** está parcialmente implementada.

Clases Abstractas y Herencia

```
public abstract class FiguraGeometrica {
private Color color;
private int x;
private int y;

public  void mover(int x, int y){
  this.x=x;
  this.y=y;
}
public void setColor(Color nuevoColor){
  color = nue<voColor;
  dibujar();
}
public abstract void dibujar();
public abstract void area();

}
```

```
public class Rombo extends FiguraGeometrica{
  public void dibujar() {
    System.out.println("dibujar() de Rombo");
  }
  public void area() {
    System.out.println("area() de Rombo");
   }
 }
```

```
public class Circulo extends FiguraGeometrica{
  public void dibujar() {
    System.out.println("dibujar() de Circulo");
  }
  public void area() {
    System.out.println("area() de Circulo");
   }
 }
```

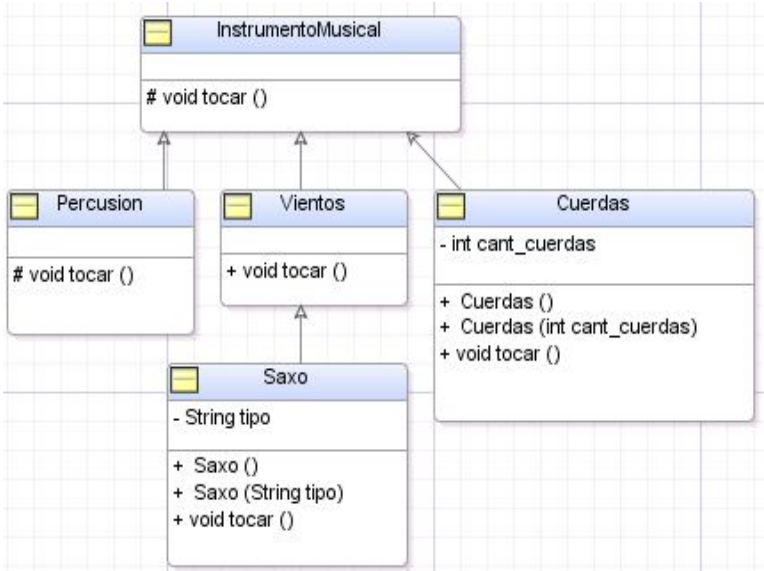
```
public class Rectangulo extends FiguraGeometrica{
  public void dibujar() {
    System.out.println("dibujar() de Rectangulo");
  }
  public void area() {
    System.out.println("area() de Rectangulo");
   }
 }
```

Polimorfismo
Herencia y Upcasting

Se tiene esta jerarquía de instrumentos musicales y una clase CancionSimple, con un método **sonar()** sobrecargado -uno con argumento de tipo Vientos y otro de tipo Cuerdas-. ¿Es una buena solución?

```
public abstract class InstrumentoMusical{
 public abstract void tocar(Nota n);
}
```

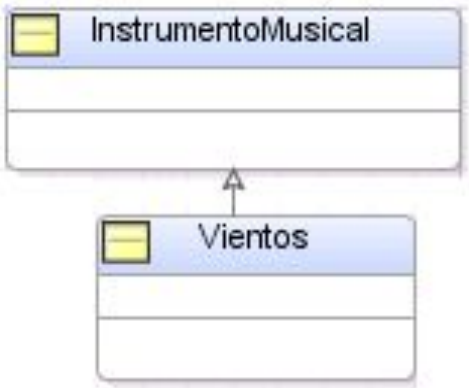
```
public class Vientos extends InstrumentoMus
 public void tocar(Nota n) {
  System.out.print("Vientos.tocar(): "+n);
 }
}
```




```
public class CancionSimple {

 private static Nota[] pentagrama=
                              new Nota[100];

 public static void sonar(Vientos i){
   for (Nota n:pentagrama)
      i.tocar(n);
 }
 public static void sonar(Cuerdas i){
   for (Nota n:pentagrama)
      i.tocar(n);
 }

 public static void main(String[] args){
   Vientos flauta = new Vientos();
   CancionSimple.sonar(flauta);
   Cuerdas piano = new Cuerdas();
   CancionSimple.sonar(piano);
 }
}
```

Polimorfismo
Herencia y Upcasting

Lo más interesante de la herencia es la relación entre la clase derivada y la clase base: "la clase derivada **es un tipo** de la clase base" (es-un o es-como-un). **Upcasting** es la conversión de una referencia de un objeto de la **clase derivada** a una referencia de la **clase base**.




```
public abstract class InstrumentoMusical{
 public abstract void tocar(Nota n);
}
```

```
public class Vientos extends InstrumentoMusical{
 public void tocar(Nota n) {
  System.out.print("Vientos.tocar():"+n);
 } }
```

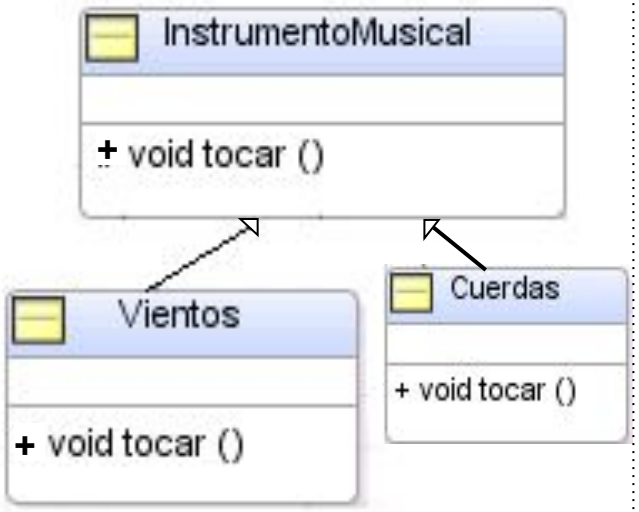
```
public class CancionSimple {
 private static Nota[] pentagrama=new Nota[100];

 public static void sonar(Instrumento Musical i){
   for (Nota n:pentagrama)
      i.tocar(n);
 }
 public static void main(String[] args){
   Vientos flauta = new Vientos();
   CancionSimple.sonar(flauta);
   Cuerdas piano = new Cuerdas();
   CancionSimple.sonar(piano);
 }
}
```

Este método puede aceptar como parámetro una referencia a un objeto `InstrumentoMusical` o a cualquier derivado de Instrumento

Se pasa como parámetro una referencia a un objeto Vientos o Cuerda sin necesidad de hacer **casting**, se hace un **upcasting** automático.

Polimorfismo

Esta solución es extensible. Se escribió código que habla con la clase `InstrumentoMusical` y con cualquiera de sus derivadas.




```
public class CancionSimple {
 private static Nota[] pentagrama =
         {new Nota("DO"),..., new Nota("RE")};

public static void sonar(InstrumentoMusical i) {
   for (Nota n: pentagrama)
      i.tocar(n);
}
```

El **polimorfismo** permite escribir código con la clase base, que funciona para las clases derivadas.

```
public static void main(String[] args) {
     InstrumentoMusical[] m={new Viento(),
                             new Cuerdas(),
                             new Saxo()};

     for (int j=0; j<m.length; j++)
       CancionSimple.sonar(m[j]);
    }
}
```

Polimorfismo y Binding Dinámico

¿Puede el compilador JAVA saber que el objeto **InstrumentoMusical** pasado como parámetro en **sonar()** es una referencia a un objeto Vientos o Cuerdas?

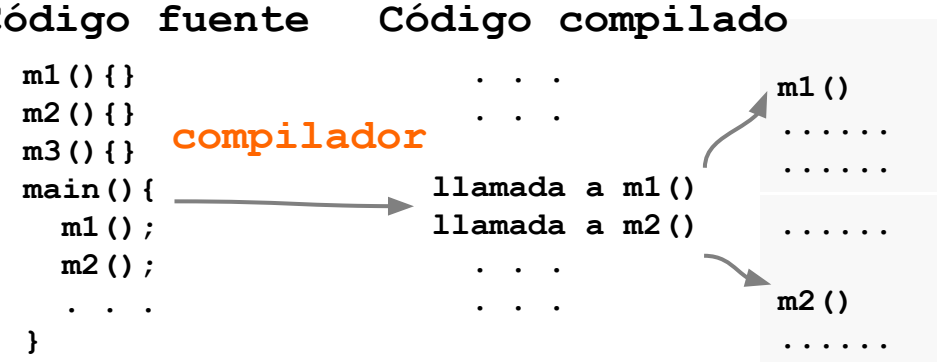
```
public class CancionSimple {
 private static Nota[] pentagrama = {new Nota("DO"),..., new Nota("RE")};
 public static void sonar(InstrumentoMusical i) {
   for (Nota n: pentagrama)
      i.tocar(n);
 }

 public static void main(String[] args) {
     if (args[0].equals("V"))
        Vientos instrumento=new Vientos();
       else
        Cuerdas instrumento=new Cuerdas();
     CancionSimple.sonar(instrumento);
 }
}
```

No hay manera de saber, mirando el código, que tipo de instrumento ejecutará el **`tocar()`**, depende **del valor que tome la variable i en ejecución**.

Conectar la invocación de un método con el cuerpo del método, se llama binding. Si el binding, se hace en compilación, se llama binding estático o temprano (early binding) y si se hace en ejecución se llam binding dinámico (dynamic binding) o tardío (late binding).

Binding temprano

En compilación se resuelven todas las invocaciones a métodos

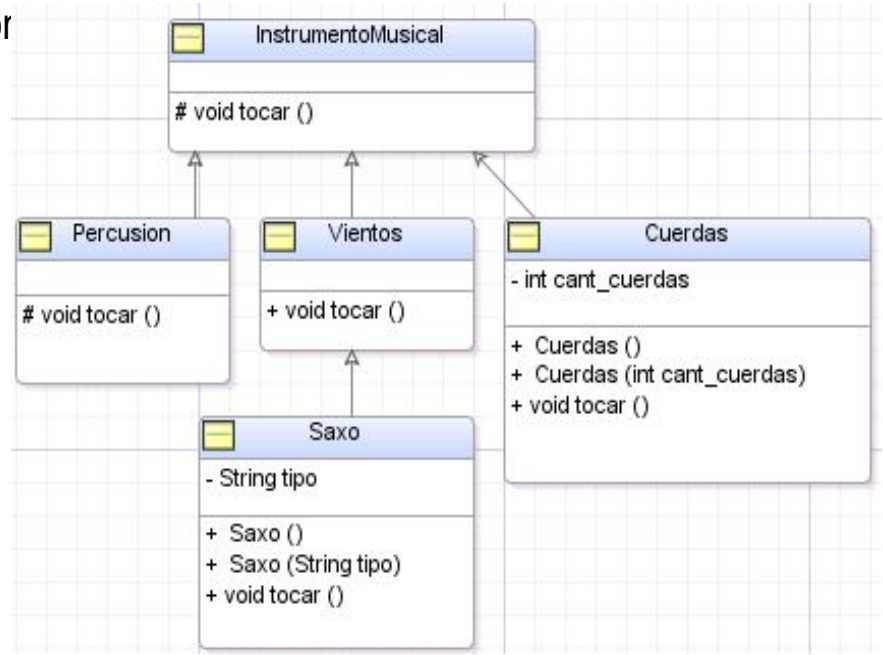


Binding Dinámico

El compilador no conoce qué método invocará cuando recibe la referencia a un objeto **InstrumentoMusical**. El binding se resuelve en ejecución.

Polimorfismo y Binding Dinámico

- JAVA usa **binding dinámico** para la resolución de las invocaciones a métodos, se hace en ejecución (run-time), basándose en el tipo del objeto receptor. Este tipo de enlace es la base de la **sobreescritura de métodos** en Java y permite la implementación de **polimorfismo**.
- JAVA provee un mecanismo para **determinar en ejecución el tipo del objeto receptor** e invocar al método apropiado.
- Aprovechando que JAVA usa binding dinámico, el programador puede escribir **código que hable con la clase base**, conociendo que las clases derivadas funcior
- Java aplica el **binding estático** para métodos de clase, finales o privados.
- El polimorfismo promueve la construcción de código **extensible**: un programador podría agregar subclases a **InstrumentoMusical** sin tocar el método **sonar(InstrumentoMusical i)**.



Polimorfismo y Binding Dinámico

El polimorfismo es un principio fundamental de la programación orientada a objetos (POO) que permite que una sola interfaz o método sea utilizado para representar múltiples comportamientos. El término "polimorfismo" proviene del griego y significa "muchas formas". Un mismo mensaje puede tener diferentes comportamientos según el objeto que o esté ejecutando.

En POO se tiene una interfaz común en la clase base y diferentes versiones de métodos en las subclases que se ligan dinámicamente: **binding dinámico**.

El **binding dinámico** resuelve a qué método invocar, cuando más de una clase en una jerarquía de herencia implementó un método (en este caso **tocar()**).

La JVM busca la implementación de los métodos invocados de acuerdo al tipo real del objeto receptor en tiempo de ejecución. B*inding* dinámico.

La declaración del tipo de la variable (InstrumentoMusical) es utilizada por el compilador de java para chequeo de errores durante la fase de compilación, pero el compilador no puede resolver qué método que se ejecutará.

Si se agregan nuevas clases a la jerarquía de herencia, el método **sonar(InstrumentoMucical i)**, no se ve afectado esto es lo que provee el polimorfismo.

```
public class CancionSimple {
 private static Nota[] pentagrama = {. . };

 public static void sonar(InstrumentoMusical i){
    for (Nota n: pentagrama)
      i.tocar(n);
 }

 public static void main(String[] args) {
    InstrumentoMusical[] m={new Viento(),
                            new Cuerdas(),
                            new Saxo()};

    for (int j=0; j<m.length; j++)
       CancionSimple.sonar(m[j]);
  }
}
```