# Interfaces y Clases Abstractas

# Clases Abstractas e Interfaces

JAVA provee 2 mecanismos para definir tipos de datos que admiten múltiples implementaciones:

**Clases abstractas**

**Interfaces**

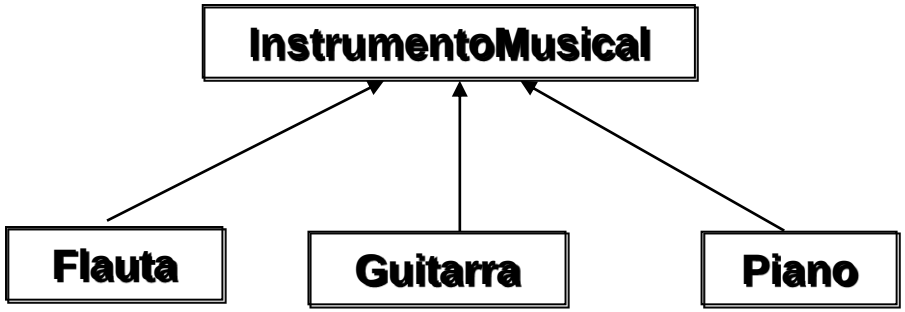
# Clases Abstractas

- Una clase abstracta representa un concepto abstracto que no es instanciable, expresa la interface de un objeto y no una implementación particular.
- Las clases abstractas permiten manipular un conjunto de clases a través de una interface común.
- Las clases abstractas se extienden, nunca se instancian. El compilador garantiza esta característica.
- Una clase abstracta se declara anteponiendo el modificador **abstract** a la palabra clave **class**.

  **abstract class InstrumentoMusical {}**
- Una clase abstracta puede contener métodos abstractos y métodos con implementación. Los métodos abstractos sólo tienen una declaración y carecen de cuerpo.

  **abstract void afinar();**
- Una clase que contiene métodos abstractos debe declararse abstracta (en otro caso, no compila).
- Las subclases de una clase abstracta de las que se desean crear objetos, deben proveer una implementación para todos los métodos abstractos definidos en la superclase. En caso de no hacerlo, las clases derivadas también son abstractas.
- Las clases abstractas y los métodos abstractos hacen explícita la característica abstracta de la clase e indican al usuario y al compilador cómo debe usarse.

# Clases Abstractas




No existe un *instrumento musical* genérico. Los instrumentos musicales hacen las mismas cosas pero de diferente manera. No es posible afinar genéricamente un instrumento musical: el piano se afina de una manera determinada, la flauta de otra, la guitarra de otra.

Se hereda comportamiento e implementación

```
public abstract class InstrumentoMusical {
  public String queEs() {
     System.out.println("Instrumento Musical");
  }
  public abstract void afinar();
  public abstract void sonar(Nota n);
}
```

Los métodos abstractos son incompletos: tienen declaración y no tienen cuerpo

```
public class Flauta extends InstrumentoMusical{
  public void sonar(Nota n) {System.out.println("sonar() de Flauta");}
  public String queEs() {System.out.println("Flauta");}
  public void afinar() {System.out.println("afinar() de Flauta");}
}
```

```
public class Guitarra extends InstrumentoMusical{
 public void sonar(Nota n) {System.out.println("sonar() de Guitarra");}
 public String queEs() {System.out.println("Guitarra");}
 public void afinar() {System.out.println("afinar() de Guitarra");}
}
```

```
public class Piano extends InstrumentoMusical{
 public void sonar(Nota n) {System.out.println("sonar() de Piano");}
 public String queEs() {System.out.println("Piano");}
 public void afinar() {System.out.println("afinar() de Piano");}
}
```

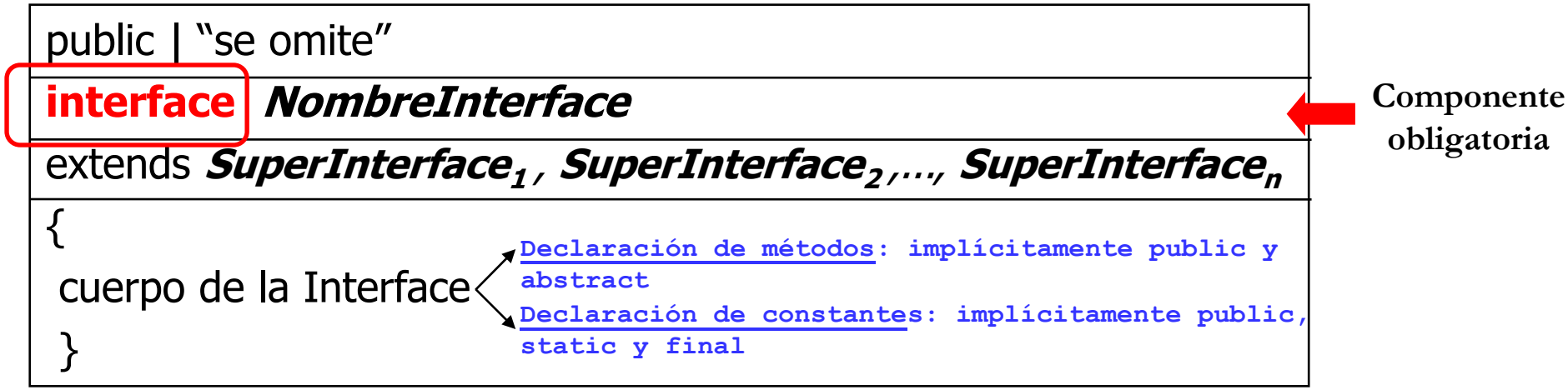
# Interfaces

**Conceptualmente una interface es un dispositivo o sistema que permite interactuar a entidades no relacionadas.**

**En JAVA una interface es una colección de definiciones de métodos sin implementación y de declaraciones de constantes agrupadas bajo un nombre.**

- **Las interfaces son completamente abstractas.**
- De igual manera que las clases, **las interfaces son tipos de datos**.
- A diferencia de las clases **las interfaces NO proveen implementación** para los tipos que ellas definen.
- **No es posible crear instancias de una interface**. Las **clases que implementan interfaces** proveen el comportamiento necesario para los métodos declarados en las interfaces. Una **interface establece qué** debe hacer la **clase que la implementa** sin especificar el **cómo**. Una instancia de dicha clase, es del **tipo de la clase y de la interface.**
- Las interfaces proveen un mecanismo de **herencia de comportamiento** y **NO de implementación**.
- Una **clase que implementa una interface tiene disponible las constantes** declaradas en la interface y **debe implementar cada uno de los métodos** declarados en la interface.
- Las **interfaces** permiten que objetos que no comparten la misma jerarquía de herencia sean del mismo tipo en virtud de implementar la misma interface.
- Una **interface** puede extender múltiples interfaces. Por lo tanto se tiene **herencia múltiple de interfaces**. No existe una interface de la cuál todas las interfaces sean extensiones: no hay un análogo a la clase Object en interfaces.
- Las interfaces proveen una **alternativa limitada y poderosa a la herencia múltiple**. Las clases en JAVA pueden heredar de una única clase pero pueden implementar múltiples interfaces.

# Declaración de una Interface




- La palabra clave **interface** permite crear interfaces.

- El especificador de acceso **public** establece que la interface puede ser usada por cualquier clase o interface de cualquier paquete. Si se omite el especificador de acceso, la interface solamente puede ser usada por las clases e interfaces contenidas en el mismo paquete que la interface declarada, visibilidad de default, NO es parte de la API que se exporta.

- Una **interface** puede extender múltiples interfaces. Por lo tanto se tiene herencia múltiple de interfaces.

**public interface I extends I1, I2, ..., IN{}**

- Una **interface** hereda todas las constantes y métodos de sus **superInterfaces**.

- Una **interface** no puede definir variables de instancia. Las variables de instancia son detalles de implementación y las interfaces son una especificación sin implementación. Las únicas **variables** permitidas en la definición de una interface son constantes de clase, **static** y **final**.

- Una interface NO puede ser instanciada, por lo tanto no define constructores.

# Un Ejemplo

## Definir una interface

```
public interface Centrable
{   void setCentro(double x, double y);
    double getCentroX();
    double getCentroY();
}
```

Declaración de Métodos

Permite que las coordenadas del centro sean *seteadas* y recuperadas

-La **interface Centrable** es una interface pública por lo tanto puede ser usada por cualquier clase e interface de cualquier paquete.

- La **interface Centrable** define tres métodos de instancia. Los métodos son implícitamente **públicos**.

- Las clases que implementen **Centrable** deberán implementar los métodos:

**setCentro(double x, double y)**, **getCentroX()** y **getCentroY()**.

-Los **métodos** de una interface son implícitamente **public** y **abstract**; las **constantes** son implícitamente **public**, **static** y **final**.

# Un Ejemplo (continuación)

## Extender una interface

```
public interface Posicionable extends Centrable
{  void setEsquinaSupDer(double x, double y);
   double getEsquinaDerX();
   double getEsquinaDerY();
}
```

Declaración de Métodos

- Una **interface** puede **extender otras interfaces**, para ello es necesario incluir la cláusula **extends** en la declaración de la interface.

- Una **interface** que extiende a otras hereda todos los métodos abstractos y constantes de sus superinterfaces y puede definir nuevos métodos abstractos y constantes. A diferencia de las clases, la cláusula **extends** de una interfaces puede incluir más de una superinterface.

```
public interface Transformable extends Escalable, Trasladable, Rotable {..}
public interface SuperForma extends Posicionable, Transformable {...}
```

- Una clase que implementa una interface debe implementar los métodos abstractos definidos por la interface que implementa directamente, así como todos los métodos abstractos heredados de las superinterfaces.

# Un Ejemplo (continuación)

## Implementar una interface

```
public class RectanguloCentrado extends Rectangulo implements Centrable {
private double cx, cy;
```

Permite nombrar las interfaces que implementa la clase

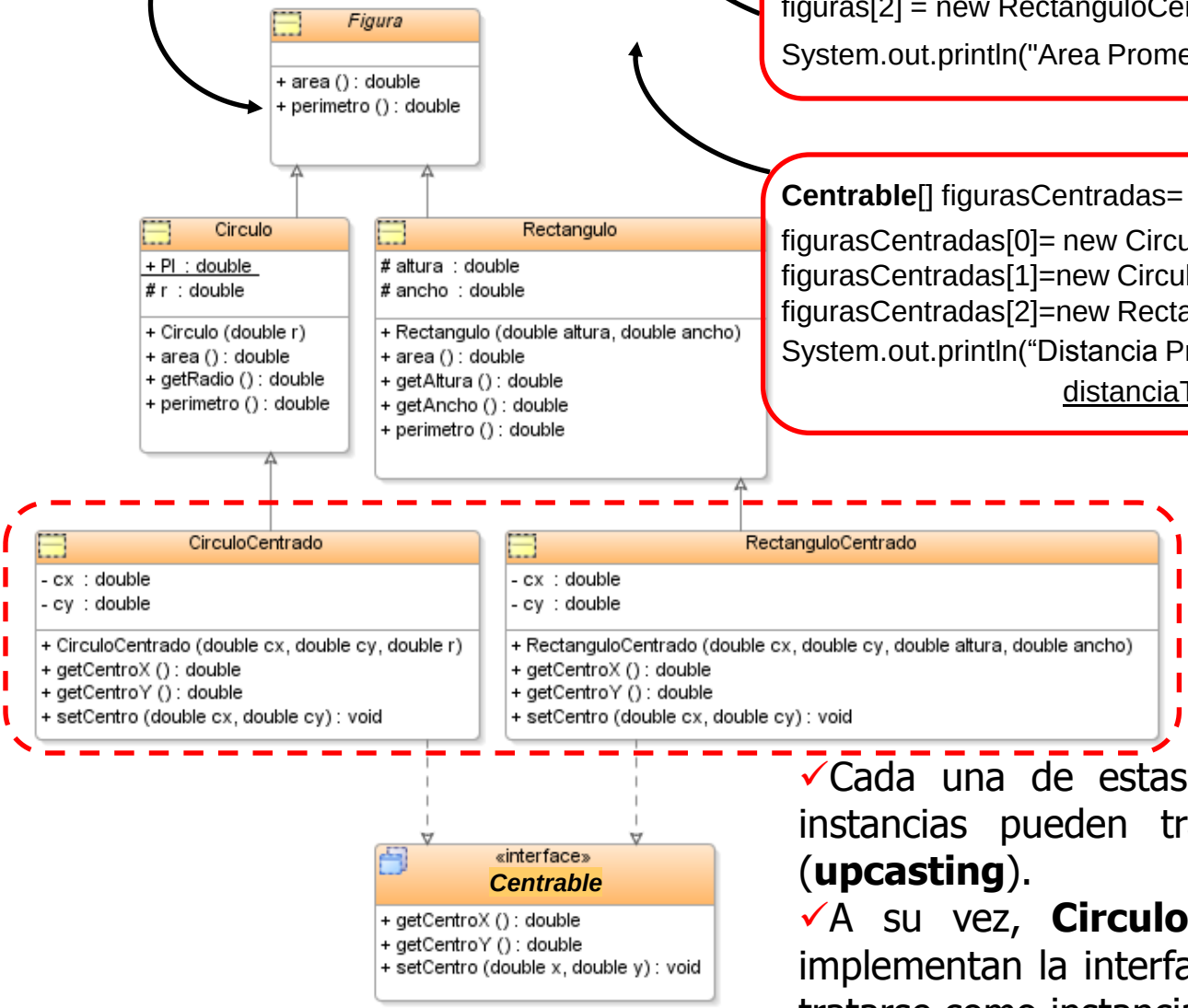
```
public RectanguloCentrado(double cx, double cy, double w, double h) {
super(w, h);
this.cx = cx;
this.cy = cy;
}

public void setCentro(double x, double y) {
cx = x; cy = y;
}
public double getCentroX() {
return cx;
}
public double getCentroY() {
return cy;
}
}
```

- ✓**Una clase que declara interfaces en su cláusula *implements* debe proveer una implementación para cada uno de los métodos definidos en dichas interfaces.**
- ✓**Una clase que implementa una interface y NO provee una implementación para cada método de la interface debe declararse abstract (hereda los métodos abstractos y no los implementa).**
- ✓**Una clase que implementa más de una interface debe implementar cada uno de los métodos de cada una de las interfaces o declararse abstract.**

Los mismos objetos son de 2 tipos diferentes: **Figura** y **Centrable**

clase abstracta




```
Figura[] figuras= new Figura[3];
figuras [0] = new CirculoCentrado(1.0, 1.0, 1.0);
figuras[1] = new CirculoCentrado(2.0, 2.0, 3);
figuras[2] = new RectanguloCentrado(2.3, 4.5, 3, 4);
System.out.println("Area Promedio:  "+areaTotal(figuras)/figuras.length);
```

```
Centrable[] figurasCentradas= new Centrable[3];
figurasCentradas[0]= new CirculoCentrado(1.0, 1.0, 1.0);
figurasCentradas[1]=new CirculoCentrado(2.0, 2.0, 3);
figurasCentradas[2]=new RectanguloCentrado(2.3, 4.5, 3, 4);
System.out.println("Distancia Promedio: " +
                    distanciaTotal(figurasCentradas)/figurasCentradas.length);
```

✓Cada una de estas clases es subclase de **Figura**, las instancias pueden tratarse como instancias de **Figura** (**upcasting**).

✓A su vez, **CirculoCentrado** y **RectanguloCentrado** implementan la interface **Centrable**, sus instancias pueden tratarse como instancias de tipo **Centrable (upcasting)**.

Las interfaces permiten crear clases que pueden ser "upcasteadas" a más de un tipo base

```
public class TestFiguras {
  static double areaTotal(Figura[] f){
      double areaTotal = 0;
      for(int i = 0; i < f.length; i++)
          areaTotal += f[i].area();
      return areaTotal;
  }
  static double distanciaTotal(Centrable[] c){
      double distanciaTotal = 0;
      double cx ;
      double cy ;
      for(int i = 0; i < c.length; i++) {
          cx = c[i].getCentroX();
           cy = c[i].getCentroY();
         distanciaTotal += Math.sqrt(cx*cx + cy*cy);
      }
      return distanciaTotal;
  }
  public static void main(String args[]){
      Figura[] figuras= new Figura[3];
      figuras [0] = new CirculoCentrado(1.0, 1.0, 1.0);
      figuras[1] = new CirculoCentrado(2.0, 2.0, 3);
      figuras[2] = new RectanguloCentrado(2.3, 4.5, 3, 4);
      System.out.println("Área Promedio: "+areaTotal(figuras)/figuras.length);

      Centrable[] figurasCentradas= new Centrable[3];
      figurasCentradas[0]=new CirculoCentrado(1.0, 1.0, 1.0);
      figurasCentradas[1]=new CirculoCentrado(2.0, 2.0, 3);
      figurasCentradas[2]= new RectanguloCentrado(2.3, 4.5, 3,4);
      System.out.println("Distancia Promedio: " + distanciaTotal(figurasCentradas)/figurasCentradas.length);

  }
}
```

tipo de una clase

tipo de una interface

upcasting

upcasting

***Castear** al tipo de la clase base*

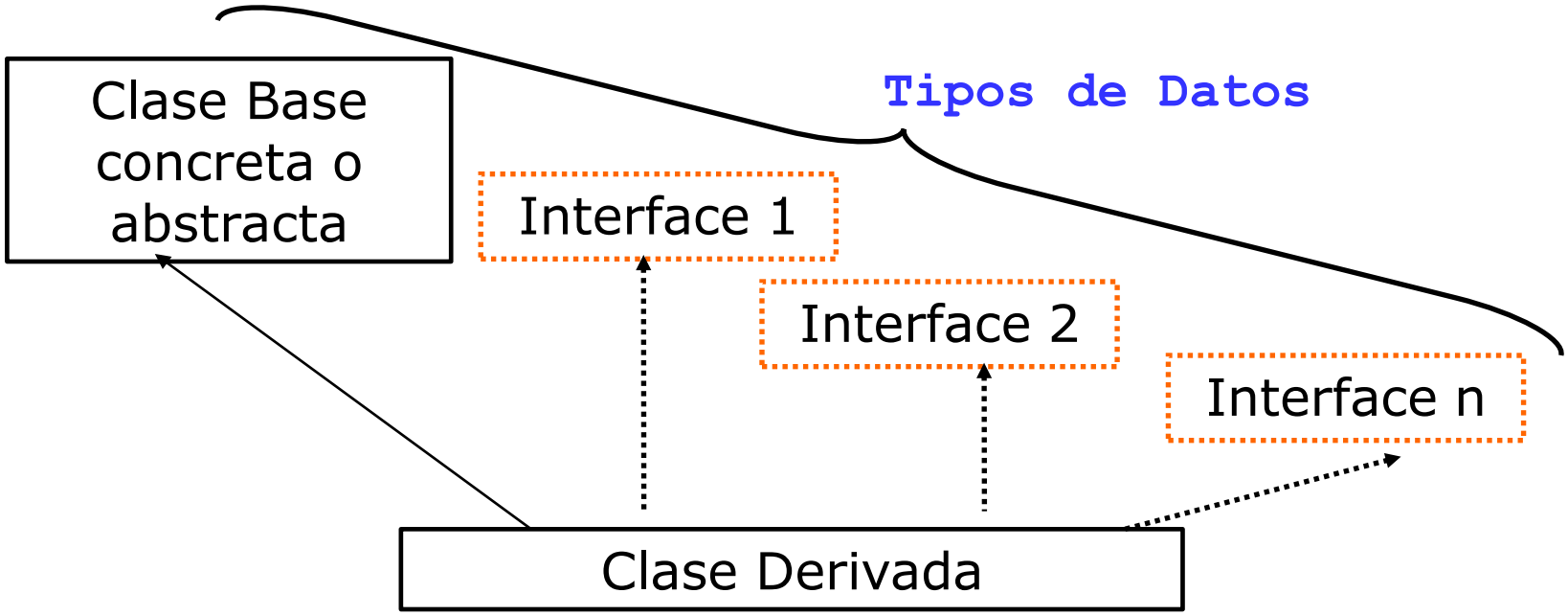
***Castear** al tipo de una interface*

# Interfaces y Herencia Múltiple

- Las **interfaces** no tienen implementación, por ende no tienen almacenamiento asociado y en consecuencia no causa ningún problema combinarlas.

- En C++ el mecanismo de combinación de clases se llama **herencia múltiple** y así es posible decir que un objeto x **es un** A y un B y un C. Pero como cada clase tiene una implementación propia, la combinación podría tener ciertos inconvenientes.

- En JAVA una clase puede implementar tantas interfaces como desee. Cada una de estas interfaces provee de un tipo de dato y solamente la clase tiene implementación. Por lo tanto se logra un mecanismo de combinación de interfaces sin complicaciones. Las interfaces son una alternativa a la **herencia múltiple**.



# Interfaces y Herencia Múltiple

El siguiente ejemplo muestra una clase concreta combinada con múltiples interfaces para producir una clase nueva:

```
interface Boxeador{
      void boxear();
}
```

```
interface Nadador{
    void nadar();
}
```

```
interface Volador{
    void volar();
}
```

```
class HombreDeAccion {
    public void boxear(){}
}
```

```
class Heroe extends HombreDeAccion implements Boxeador,
Nadador, Volador{
 public void nadar(){}
 public void volar(){}
}
```

La clase **Heroe** es creada a partir de la combinación de la clase concreta **HombreDeAccion** con las interfaces **Boxeador**, **Nadador** y **Volador**. Por lo tanto, un objeto **Heroe** es también un objeto **HombreDeAccion**, **Boxeador**, **Nadador** y **Volador**.

# Interfaces y Herencia Múltiple

```
public class Aventura{
    static void t(Boxeador x) {
       x.boxear();
    }
    static void u(Nadador x) {
       x.nadar();
    }
    static void v(Volador x) {
       x.volar();
    }
    static void w(HombreDeAccion x) {
       x.boxear();
    }
  public static void main(String[] args){
  Heroe e=new Heroe();
  t(e); // e es un Boxeador
  u(e); // e es un Nadador
  v(e); // e es un Volador
  w(e); // e es una HombreDeAccion

  }
} // Fin de la clase Aventura
```

upcasting

**El principal objetivo de las interfaces es poder hacer "upcasting" a más de un tipo base. De esta manera se logra una variación a la herencia múltiple.**

**Otro objetivo del uso de interfaces es similar al de las clases abstractas: evitar que el programador cliente cree objetos y establecer solamente una "interface de comportamiento común".**

- Un objeto Heroe es creado y pasado como parámetro a los métodos t(), u(), v(), w() que reciben como parámetro objetos del tipo de una interface y de una clase concreta.
- El objeto Heroe es *upcasteado* automáticamente al tipo de interface o de la clase según corresponde.

# Interfaces Marker

- Son interfaces completamente vacías.
- Una clase que implementa una interface *marker* simplemente debe nombrarla en su cláusula **implements** sin implementar ningún método. Cualquier instancia de la clase es una instancia válida del tipo de la interface.
- Es una técnica útil para proveer información adicional sobre un objeto. Es posible verificar si un objeto es del tipo de la interface usando el operador *instanceof*.

La interface **java.io.Serializable** es una interface *marker*: una clase que implementa la interface Serializable le indica al objeto ObjectOutputStream que sus instancias pueden persistirse en forma segura.
La interface **java.util.RandomAccess** también es una interface *marker*: algunas implementaciones de **java.util.List** la implementan para indicar que proveen acceso *random* a los elementos de la lista.
Los algoritmos a los que les interesa la *performance* de las operaciones con acceso *random* pueden testearlo de esta manera:

```
public class ArrayList<E> extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, java.io.Serializable {}

public class LinkedList<E> extends AbstractSequentialList<E>
                    implements List<E>, Queue<E>, Cloneable, java.io.Serializable {}
```

```
List l = ...; // Alguna implementación de List
if (l.size() > 2 && !(l instanceof RandomAccess)) l = new ArrayList(l);
OrdenarLista(l);
```

Antes de ordenar una lista de gran cantidad de elementos, nos aseguramos que la lista provea acceso *random*. Si no lo provee hacemos una copia de la lista en un ArrayList antes de ordenarla (ArrayList sí provee acceso random).

# Interfaces y Clases Abstractas

-Las **interfaces** y las **clases abstractas** proveen una **interface de comportamiento común**.

- Las **interfaces** son **completamente abstractas**, no tienen ninguna implementación. Las **clases abstractas NO** necesariamente son **completamente abstractas**, las subclases pueden aprovechar de esta implementación parcial.

- Con **interfaces** NO hay herencia de métodos ni de variables de instancia. Proveen "herencia de comportamiento y no de implementación". Las clases abstractas proveen "herencia de comportamiento e implementación"

- De una **clase abstracta** no es posible crear instancias; de las **interfaces** tampoco.

- Una clase solamente puede extender una clase abstracta pero puede implementar más de una interface.

**¿Interfaces o clases Abstractas?**

- Una interface es útil pués permite ser implementada por cualquier clase aún si dichas clases extienden otra clase no relacionada. Una **interface es especificación pura**, NO contiene implementación. Si una interface declara muchos métodos puede llegar a ser tedioso implementarla.

- Las clases abstractas fuerzan relaciones de herencia, las interfaces NO.

-Agregar nuevas definiciones de métodos en interfaces que forman parte de la API pública ocasiona problemas de incompatibilidad con el código existente. Las clases abstractas pueden en forma segura agregar métodos no-abstractos sin romper el código de las clases que las extienden. **JAVA 8 soluciona el problema de incompatilidad de las interfaces**

- En algunas situaciones la elección es de diseño.

- Es posible combinar clases abstractas e interfaces: definir un tipo como una interface y luego una clase abstracta que la implementa parcialmente, proveyendo implementaciones de defecto que las subclases aprovecharían (*skeletal implementation*).

```
abstract class Mamifero {
public abstract void comer();
}
```

```
abstract class Mascota{
public void respirar(){…}
public abstract void entretener();}
```

```
class Perro extends Mamifero, Mascota {} ERROR!!!
```

**Java NO soporta herencia múltiple de clases, por lo tanto si se desea que una clase sea además del tipo de su superclase de otro tipo diferente es necesario usar interfaces.**

# Propiedades de las Interfaces

- Las interfaces definen un tipo de dato, por lo tanto es posible declarar variables con el nombre de la interface. Ejemplo: **Centrable f;**
- La variable **f** del tipo de la interface **Centrable** hace referencia a un objeto de una clase que implementa dicha interface. Ejemplo: **f=new CirculoCentrado(); //CirculoCentrado implementa la interface Centrable**.
- Herencia múltiple de interfaces: es posible definir una nueva interface **heredando** de otras ya existentes.

```
interface Monstruo {
    void amenazar();
}
```

```
interface MonstruoPeligroso extends Monstruo{
    void destruir();
}
```

```
interface Letal {
    void matar();
}
```

```
interface Vampiro extends MonstruoPeligroso, Letal{
  void beberSangre();
}
```

**Herencia múltiple de interfaces**

- **Métodos de la interface MonstruoPeligroso**: destruir() y amenazar().
- **Métodos de la interface Vampiro**: beberSangre(), matar(), destruir() y amenazar().

- En una **interface NO pueden definirse variables de instancia** pués son detalles de implementación, **ni métodos estáticos** pués no pueden declararse abstractos.
- Los métodos de una interface son automáticamente **public y abstract** y las constantes son **public**, **static** y **final**

# Colisión de Nombres

¿Es posible que una interface herede de sus super-interfaces un atributo con igual nombre?

```
public interface I{
int W=5;
}
```

```
public interface J {
int W=10;
}
```

¿Hay conflicto de nombres?

```
public interface K extends I, J {
 int Z=J.W+5;

}
```

- Si en la interface K **NO** se hace referencia al atributo W, no hay conflicto de nombres, no hay error de compilación.

- Si en la interface K se hace referencia a W por su nombre simple, entonces hay una referencia ambigua que el compilador no sabe resolver. La ambigüedad de nombres se resuelve usando el nombre completo del atributo.

- Si un mismo atributo se hereda múltiples veces desde la misma interface, por ejemplo si tanto la interface que estamos definiendo como alguna de sus superinterfaces extienden la interface que declara el atributo en cuestión, entonces el atributo se hereda una única vez.

# Herencia y Sobreescritura

```
public interface I {
int f();
}
```

```
public interface J extends I {
int f();
}
```

¿Hay conflicto?

Si el tipo de retorno es primitivo: el método f() sobreescrito en la interface J debe tener el mismo tipo de retorno que el definido en I. En otro caso, hay conflicto.

```
public interface I {
Figura f();
}
```

```
public interface J extends I {
Circulo f();
}
```

**La declaración de un método en una interface sobreescribe a todos aquellos que tienen la misma firma que en sus superinterfaces.**

**Hay error de compilación si el tipo de retorno del método sobre-escrito no es subtipo del retornado por el método original.**

**El método sobreescrito no debe tener claúsulas throws que causen conflicto con alguno de los métodos que sobreescribe.**

Soporta tipo de retorno *covariante*: el tipo de retorno del método sobreescrito podría ser una subclase del tipo de retorno del método original.

# Interfaces
# Java 8

```
public interface SimpleI{
      public void hacer ();
}

class ClaseSimple implements SimpleI {
     @Override
      public void hacer() {
        System.out.println("hacer algo en la clase");
       }

      public static void main(String[] args) {
         ClaseSimple s = new ClaseSimple ();
         s.hacer();
      }
}
```

**¿Y si necesitamos agregar un método nuevo a la interface SimpleI?**

```
public interface SimpleI{
      public void hacer ();
      public void  hacerOtraCosa();
}
```

**ClaseSimple deja de compilar!**

-Esta limitación hace que sea **casi imposible extender y mejorar interfaces** existentes y APIs.

- Los diseñadores de JAVA se enfrentaron a esta situación al intentar extender la API de Colecciones en JAVA 8.

**¿Qué solución se adoptó en JAVA 8?**

Incorporó **métodos de default** ó métodos "Defender" o métodos virtuales.

# Interfaces Java 8

Los métodos de default contienen una implementación predeterminada y colaboran en la evolución de las interfaces sin “romper” el código existente.

```
public interface SimpleI{
      public void hacer ();
}

class ClaseSimple implements SimpleI {
    @Override
     public void hacer() {
       System.out.println(“hacer algo en la clase");
     }
  /*
    * no requiere proveer una implementación
    * para hacerOtraCosa
    */
     public static void main(String[] args) {
        ClaseSimple s = new ClaseSimple ();
        s.hacer();
        s.hacerOtraCosa();
     }
}
```

```
public interface SimpleI{
     public void hacer ();
     default public void  hacerOtraCosa() {
         System.out.println(“hacerOtraCosa en la interface ");
      }
}
```

**Los métodos de default permiten agregar funcionalidad nueva a las interfaces asegurando compatibilidad binaria con el código escrito para versiones previas de la misma interface**

Es preferible evitar el uso de métodos de default en las interfaces