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 e Interfaces

Encapsulamiento y Ocultamiento de Información

Diseñar clases e interfaces usables, robustas y flexibles: separar la API pública de la implementación. ¿Cómo?

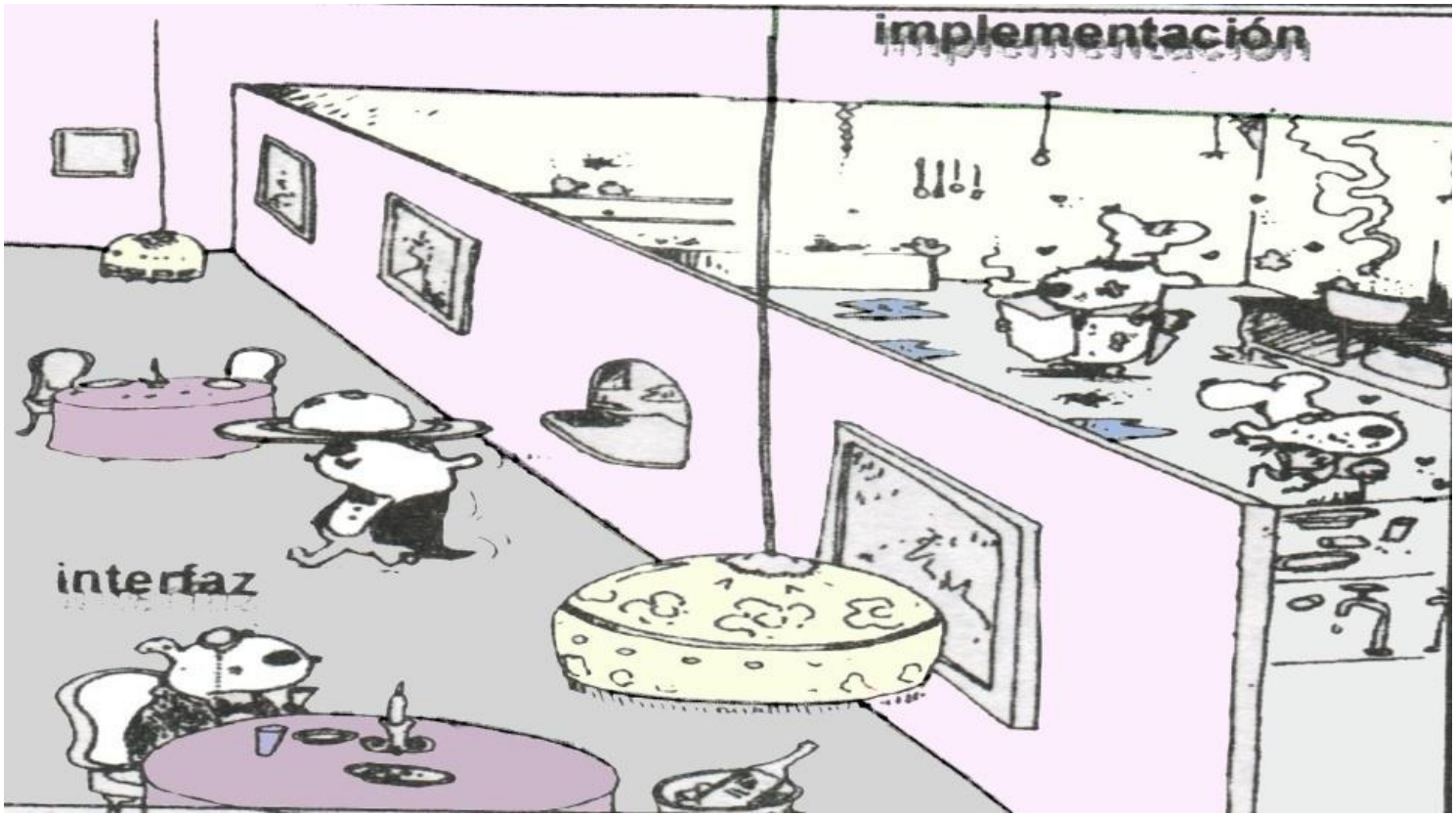


Imagen del libro “Object Oriented Analysis and Design” de Grady Booch

Clases Abstractas

- Representan un concepto abstracto, **no es instanciable**, expresa la **interface de un objeto** y **no una implementación particular**.
- Permiten manipular un **conjunto de clases** a través de una **interface común**.
- Se **extienden**, nunca se instancian. El compilador garantiza esta característica.
- Se declaran anteponiendo el modificador **abstract** a la palabra clave **class**.

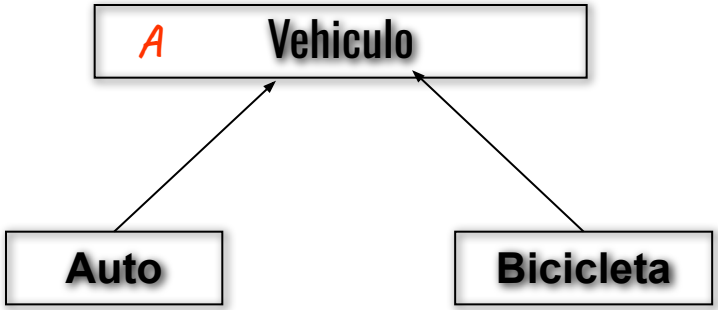
```
abstract class Vehiculo {}
```

- Puede contener **métodos abstractos** y **métodos con implementación**. Los métodos abstractos sólo tienen una declaración y carecen de cuerpo.

```
abstract void arrancar();
```

- 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 de 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 de la clase y al compilador cómo debe usarse.

Clases Abstractas



No existe un *vehículo* genérico, es un concepto general, sin embargo todos los vehículos comparten algunas características comunes, en el ejemplo tienen ruedas y se pueden arrancar. No es posible ***arrancar genéricamente un vehículo,*** pero podemos arrancar autos o bicicletas.

En este ejemplo se hereda comportamiento e implementación:

```
public abstract class Vehiculo{
  private int ruedas;

  public String queEs() {
     return "Vehículo";
   }
  public abstract void arrancar();
}
```

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

```
public class Auto extends Vehiculo{
  public void arrancar() {System.out.println("arrancar() de Auto");}
  public String queEs() {return "Auto";}
 }

public class Bicicleta extends Vehiculo{
  public void arrancar() {System.out.println("arrancar() de Bicicleta");}
  public String queEs() {return "Bicicleta";}
 }
```

```
public class Test {
  public static void main(String[] args) {
    var v1 = new Auto();                    Vehiculo v2 = new Bicicleta();
    System.out.println(v1.queEs());         System.out.println(v2.queEs());
    v1.arrancar();                          v2.arrancar();
   }
}
```

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.**
- Las **interfaces son tipos de datos**.
- L**as interfaces NO proveen implementación** para los tipos que ellas definen (hasta JAVA 8!)
- **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**.
- Las **clases que implementan interfaces tienen disponibles las constantes** declaradas en las interfaces y **debe implementar cada uno de los métodos** allí definidos.
- Las **interfaces** permiten que **objetos que no comparten la misma jerarquía de herencia** sean del **mismo tipo** en virtud de implementar la misma interface.
- **Herencia múltiple de interfaces:** una interface puede extender múltiples interfaces. No existe una interface de la cual todas las interfaces sean extensiones, no hay un análogo a la clase Object en interfaces.
- Las interfaces ofrecen una **alternativa limitada y poderosa a la herencia múltiple**. Las clases en JAVA pueden heredar de una única clase e implementar múltiples interfaces.

Declaración de una Interface

public | "se omite" (package)

interface *NombreInterface*

extends *SuperInterface* $_1$, *SuperInterface* $_2$,..., *SuperInterface* $_n$

{

cuerpo de la Interface

}

Declaración de métodos: implícitamente public y abstract

Declaración de constantes: implícitamente public, static y final

Componente obligatoria

- 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, es parte de la **API que se exporta**. 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, **es parte de la implementación**.
- 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 define 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**, que se declaran **static** y **final**.
- Una interface NO puede ser instanciada, por lo tanto no define constructores.

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**: puede usarse en cualquier clase e interface de cualquier paquete.

- La **interface Centrable** define tres métodos de instancia, 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**.

¿Podrían definirse métodos de clase (static) en una interface?

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 y constantes. A diferencia de las clases, la cláusula **extends** de las 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.

Herencia y Sobreescritura

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

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

¿Hay conflicto?

¡¡Sí!! Si el tipo de retorno del método f() es primitivo, el método sobreescrito debe tener el mismo tipo de retorno. **Hay conflicto**.

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

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

¿Hay conflicto?

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 sobreescrito 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.

¡¡No!! El tipo de retorno del método sobreescrito podría ser una subclase del tipo de retorno del método original. **Tipo de retorno covariante**.

Ejemplo (continuación)

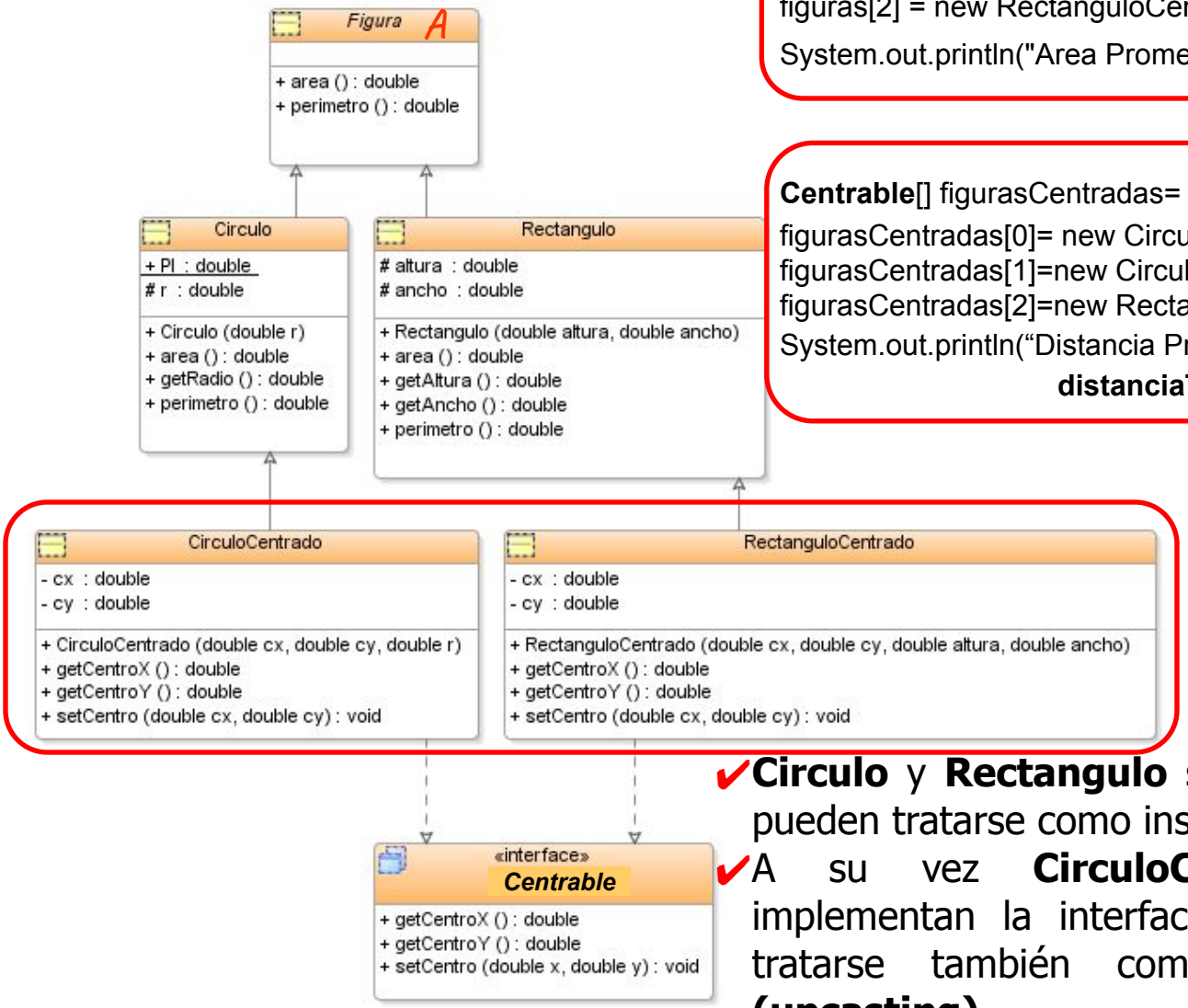
Implementar una interface

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

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;
}
}
```

Permite nombrar las interfaces que implementa la clase

- ✔ 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**




```
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);
```

✔**Circulo** y **Rectangulo** son 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 también 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

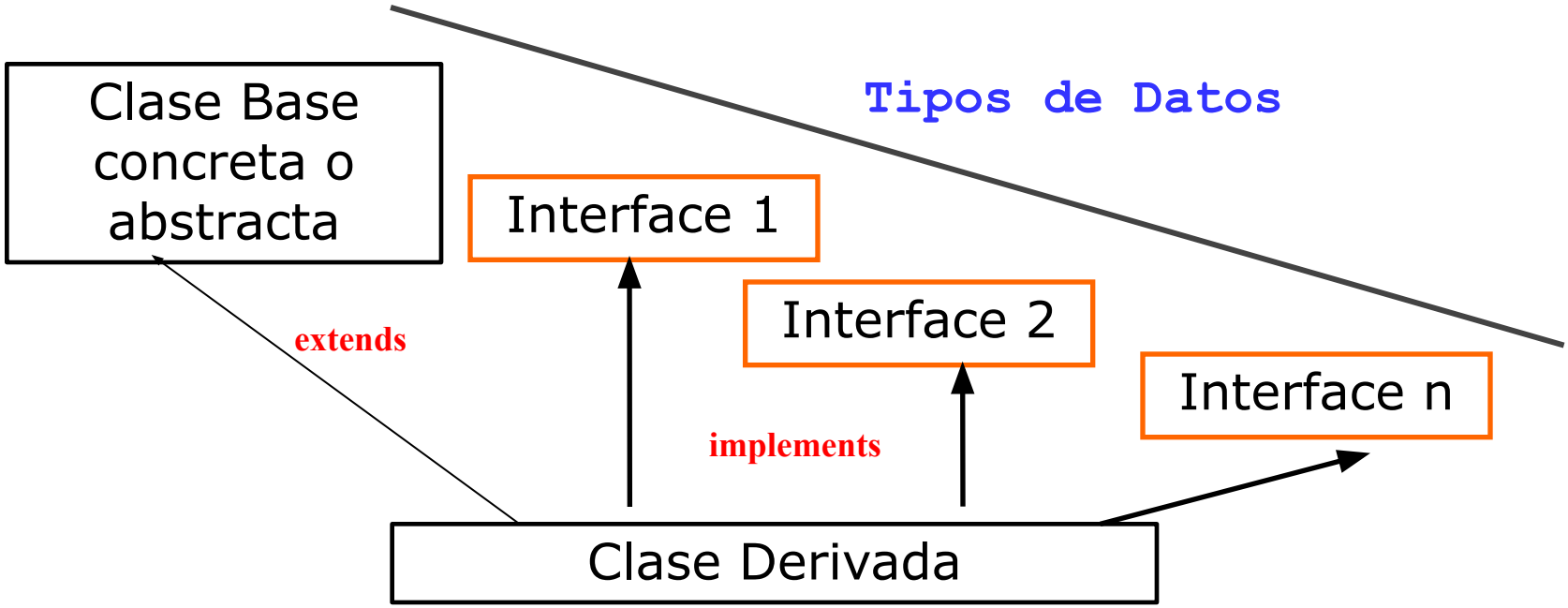
***Castear** al tipo de la clase base*

***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 JAVA una clase puede implementar tantas interfaces como desee. Cada una de estas interfaces provee de un tipo de dato nuevo 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** y las interfaces **Boxeador**, **Nadador** y **Volador**. Por lo tanto, un objeto **Heroe** es también un objeto **HombreDeAccion**, **Boxeador**, **Nadador** y **Volador**.

¿Qué métodos implementa Heroe? ¿se podrían declarar de alcance package los métodos de Heroe?

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 un HombreDeAccion

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

upcasting

Las interfaces permiten 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: 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 corresponda.

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. **¿Qué tiene que ver serialVersionUID con Serializable? ¿Qué clases son candidatas a implementar Serializable?**

La interface **java.util.RandomAccess** 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**.
- JAVA 8 y JAVA 9 introducen interfaces con implementaciones de default para métodos de instancia, métodos de clase y métodos privados. En este sentido podríamos decir que tanto las **interfaces** como las clases abstractas proveen **herencia de comportamiento** y de **implementación**.
- De una **clase abstracta** no es posible crear instancias; de las **interfaces** tampoco.
- Para implementar un tipo definido por una **clase abstracta** es necesario extender a la clase abstracta, se **fuerza una relación de herencia**. Sin embargo cualquier clase puede implementar interfaces independientemente de la jerarquía de herencia a la que pertenezca.

¿Interfaces o clases Abstractas?

- Las **interfaces** son ideales para **definir "mezclas"**: una "mezcla" es un tipo que una clase puede implementar además de su tipo primario. Ejemplo: La interface **Comparable** permite que una funcionalidad (ordenación) se puede mezclar con su función primaria. Con clases abstractas no se pueden definir este tipo de comportamiento.
- 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 agregar métodos no-abstractos sin romper el código de las clases que las extienden (a veces lo rompen!). **JAVA 8 soluciona el problema de incompatilidad de las interfaces (¿polémico?)**
- 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, minimizando el esfuerzo de implementar una clase (*skeletal implementation*). Patrón **Template Method.** En colecciones se usa este patrón: **AbstractSet, AbstractList, AbstractMap, etc.**

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

}
```

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

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.

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

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** hace referencia a un objeto de una clase que implementa la interface **Centrable**. 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**

¿Cuáles son los métodos de la interface MonstruoPeligroso? destruir() y amenazar()

¿Cuáles son los métodos de la interface Vampiro? beberSangre(), matar(), destruir(), amenazar()

- En una **interface NO pueden definirse variables de instancia** pues son detalles de implementación.
- Los métodos de una interface son automáticamente **public y abstract** y las constantes son **public, static** y **final.** Algo de esto cambio

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.

Interfaces para la posteridad Java 8

Antes de Java 8 era imposible agregar métodos a una interface sin romper las implementaciones existentes.

```
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** o métodos "Defender" o métodos virtuales.

Interfaces para la posteridad
Java 8

Los **métodos de default** contienen una implementación predeterminada que utilizan todas las clases que implementan la interface que no implementan.

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

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. Sin embargo no hay garantía que funcione para todas las implementaciones.

Los **métodos predeterminados se "inyectan"** en las implementaciones existentes sin el conocimiento y consentimiento de sus desarrolladores.

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

Conflictos

```
interface Bar {
    default void bar() {
        System.out.println("Método bar() de Baz");
    }
}
```

```
public class MiClase implements Foo, Bar {
    // conflicto entre Foo y Baz
     @Override
        public void bar() {
       }

    public static void main(String[] args) {
        MiClase obj = new MiClase();
        obj.bar();
     // Llamada al método estático de la interfaz Foo
        Foo.buzz();
    }

}
```

```
public interface Foo {
    default void bar() {
        System.out.println("Método bar() de Foo");
        baz();
    }
    private void baz() {
        System.out.println(" Hola mundo!");
    }
    static void buzz() {
        System.out.print("Hola");
        staticBaz();
    }
    private static void staticBaz() {
        System.out.println(" mundo estático!");
    }
}
```

A partir de Java 9, se pueden definir **métodos privados** dentro de las interfaces, estáticos o de instancia, ¿para qué?

Cuando una clase implementa múltiples interfaces con métodos default con la misma firma, el compilador no sabe cuál implementar. **¿Cómo se puede solucionar?**