IPOO

- Docentes
  - Profesores Adjuntos:
    - Carlos Di Cicco.
  - JTPs:
    - Federico Naso (Junín) .
      Nelson Di Grazia (Pergamino) .

Java – Herencia

- La programación orientada a objetos permite a las clases expresar similitudes entre objetos que tienen algunas características y comportamiento común.
- Estas similitudes pueden expresarse usando **herencia**.
- El término **herencia** se refiere al hecho de que una clase **hereda** los atributos (variables) y el comportamiento (métodos) de otra clase.
- La herencia toma una clase existente y construye una versión especializada => reusabilidad de código.

Java – Herencia

- Java posee:
  - Herencia simple de clases
  - Herencia múltiple de interfaces

Herencia de clases

- Consiste en crear una clase nueva, como una especialización de una clase existente. La clase nueva toma la forma de una clase existente y le agrega código (sin modificar la clase existente). Reutilización de código.
- Al ser un lenguaje compilado, el compilador es el que hace la mayor parte del trabajo.
- Para indicar herencia se utiliza la palabra clave **extends**
- La clase Object es la raíz de la jerarquía de clases en Java. Cualquier clase que no especifique un padre directo, será subclase directa de Object.

Herencia de clases

```
public class Persona {
        String apellido;
        String nombre;
        Date fechaNacimiento;
        public String getApellido(){
            return apellido;
        }
        public Persona (){
            System.out.println("constructor de persona");
        }
}
public class Alumno extends Persona{
    int legajo;
    public Alumno(){
        System.out.println("constructor de alumno");
    }
    public int getLegajo(){
      return legajo;
    }
}
```

Herencia de clases

- Contenido de la clase Alumno
  - Variables de instancia propias: legajo
  - Variables de instancia heredadas: nombre, apellido y fechaNacimiento
  - Métodos propios:
    - getLegajo()
  - Métodos heredados de Persona:
    - getApellido()
  - Métodos heredados de Object
    - equals(), toString()
  - Constructores
    - Alumno()

Herencia

- **Inicialización**
  - Un objeto de la clase derivada, contiene un sub-objeto de la superclase.
  - Es esencial que el sub-objeto de la clase base se inicialice correctamente y la manera de garantizarlo es realizar la inicialización en el constructor, invocando al constructor de la superclase quién tiene el conocimiento apropiado para hacerlo.
  - Es fundamental que sean invocados todos los constructores de la cadena de herencia, y de esta manera garantizar que el objeto está construido correctamente.
  - El compilador Java, invoca al constructor nulo o predeterminado de la superclase en el constructor de la clase derivada (si no se lo invocó)

Herencia

- **¿Cómo se construye un objeto Alumno?**
  - Recorriendo la jerarquía de herencia en forma ascendente e invocando al constructor de la superclase desde cada nivel de la jerarquía de clases, esto es: el constructor de Alumno invoca al constructor de Persona, y el de Persona al de Object.
- **La creación de un objeto Alumno, involucra:**
  - crear un objeto de tipo Object
  - crear un objeto de tipo Persona
  - crear un objeto de tipo Alumno
- Si no se codifica un constructor, el compilador insertará un constructor predeterminado que invoca al constructor de la superclase

Herencia – Inicialización

- **Constructores con Argumentos**
  - Para el compilador Java, es sencillo invocar al constructor sin argumentos, sin embargo si la clase no tiene constructor nulo (porque se escribió uno no-nulo) o si se quiere invocar a un constructor con argumentos, la invocación debe hacerse en forma explícita usando la palabra clave super y la lista de argumentos apropiada.

```
public class Persona{
  private String apellido;
  public Persona(String patas){
    System.out.println("Constructor de Persona con args");
  }
}
```

Herencia – Inicialización

```
public class Alumno extends Persona{
   public Alumno(String apellido){
      super(apellido);
      System.out.println("Constructor de Alumno con args");
   }
   public Alumno(){
      this("");
      System.out.println("Constructor de Alumno sin args");
   }}
```

- Si no se invoca al constructor de la superclase, el compilador intentará invocar al constructor nulo.
- Lo primero que se hace en el constructor de la clase derivada es invocar al constructor de la superclase
- Se puede invocar a un constructor de la misma clase a través de la palabra clave this()

Herencia - Upcasting

- Relación entre la clase derivada y la superclase: "la clase derivada es un tipo de la superclase" (es-un). Java soporta esta relación.
- Upcasting es la conversión de una referencia a un objeto de la clase derivada en una referencia a un objeto de la superclase.
- El upcasting es seguro: la clase derivada es un super conjunto de la superclase, podría contener más métodos que la superclase, pero seguro contiene los métodos de la super.

Herencia - Upcasting

```
public class Inscripcion {
    public void inscribir(Persona persona){ //puede recibir Persona o cualquiera de sus subclases
            int edad = persona.getEdad(); //funciona OK
            int legajo = persona.getLegajo(); //no compila
            int legajo = (Alumno) persona.getLegajo(); //casting de Persona a Alumno. Falla en (*)
    }
    …
    public static void main(String[] args){
            Alumno alumno = new Alumno("Perez");
            Inscripcion ins = new Inscripcion();
            ins.inscribir(alumno); //upcasting
            Persona persona = new Persona("Lopez");
            ins.inscribir(persona); //Compila pero falla en ejecución (*)
    }
}
```

Herencia - Upcasting

- El ejemplo anterior acepta como parámetro una referencia a una Persona o a cualquier derivado de la clase
- Una referencia a un objeto Alumno es pasada como parámetro al método inscribir(), sin hacer casting
- El Upcasting de Alumno a Persona limita la interface de Alumno a la de Persona

Polimorfismo

- ¿Puede el compilador saber que el objeto Persona pasado como parámetro, es una referencia a un objeto Alumno?
  - NO

- Conectar la invocación a un método con el cuerpo del método, se llama binding. Si el binding, se hace en compilación, se llama Early Binding y si se hace en ejecución Late Binding o Dynamic Binding.

Polimorfismo

- Java, normalmente usa Dynamic Binding: la resolución de las invocaciones a métodos, se resuelve en ejecución, basándose en el tipo del objeto receptor. Java provee un mecanismo para determinar en ejecución el tipo del objeto receptor e invocar al método apropiado.
- En Java el Dynamic Binding es automático y se utiliza para resolver métodos polimórficos.
- Java usa Early Binding para métodos declarados final o static (los declarados private son implícitamente final)

Constructores con polimorfismo

- Los constructores no son polimórficos, en realidad son implícitamente static.
- La invocación de constructores plantea un dilema : ¿Qué pasa si en el cuerpo de un constructor se invoca a un método del objeto que se está construyendo?
  - Conceptualmente, el constructor fabrica un objeto, pone a un objeto en un estado inicial.
  - En el cuerpo de un constructor, el objeto está parcialmente construido, solo se sabe que los subobjetos de las clases bases se inicializaron, pero no sabemos nada de las clases derivadas.
  - El binding dinámico busca el método a invocar "hacia fuera" en la cadena de herencia, pudiendo invocar a un método de la clase derivada. De está manera manipularía atributos aún no inicializados, lo cuál podría resultar en bugs difíciles de detectar, etc

Constructores con polimorfismo

- Buena práctica para constructores:
  - Inicializar el estado del objeto evitando en lo posible invocar a métodos.
  - Los únicos métodos que son seguros para invocar en el cuerpo de un constructor son los declarados final en la clase base o private (que son automáticamente final). Estos métodos no pueden sobreescribirse y por lo tanto funcionan correctamente.

Java – this

- La palabra clave **this** hace referencia al propio objeto y se utiliza para enviar mensajes al mismo
- Es muy común usar la palabra clave **this** dentro de los métodos de instancia de una clase, para referirse al objeto que está ejecutando el código.
- Cuando usar **this**
  - Cuando no puedo no usarlo: Dentro del cuerpo de un método es posible referir directamente a las variables miembros de un objeto por su nombre, sin embargo, a veces una variable miembro está oculta por un parámetro de un método que tiene el mismo nombre

Java – this

```
package ejemplo;
 public class Persona {
    private String apellido;
    . . .
    public void setApellido(String apellido){
        this.apellido = apellido;
    }
}
```

- si no usamos el **this**, no asignará al atributo de la clase Persona el valor que recibe como parámetro.

Java – this

- Cuando usar **this**
  - Cuando no debo no usarlo: ocultamiento y encapsulamiento!

```
public boolean isMayor(){
  return (edad >= 21);
}
public boolean isMayor(){
  return (this.getEdad() >= 21);//envía mensaje a si
    mismo
}
```

Java – this vs this()

- Entonces: **this** se usa para referenciar al objeto "actual", representa al objeto que estamos manipulando.
- Se puede usar en métodos de instancia y en constructores.
- Como vimos, sirve para quitar ambigüedad de nombres, por ej cuando los argumentos de los métodos o constructores son idénticos a los de las variables de instancia

Java – this vs this()

- A diferencia de **this**, la palabra clave **this()** se utiliza en constructores para referenciar a otro constructor de la misma clase.
- Se usa para invocar a un constructor desde el cuerpo de otro constructor.
- Cuando en una clase se declaran múltiples constructores, es posible que se invoquen entre ellos y de esta manera se evita duplicar código: **Re-uso de código**
- Se debe usar en la primera línea del cuerpo del constructor

Java – this vs this()

```
public class Persona{
    int dni;
    public void setDni(int dni){
        this.dni = dni;
    }
    public Persona(){
        //hace algo
    }
    public Persona(int dni){
        this();
        this.setDni(dni);
    }}
```

Java – super

- La palabra clave **super** hace referencia a la superclase del propio objeto y se utiliza para enviar mensajes definidos específicamente en la superclase
- Cuando usar **super**
  - Cuando quiero acceder a una variable o un método definido en la superclase que también está definido en la propia clase

Java – super

```
package ejemplo;
 public class Persona {
    . . .
    public String toString(){
        return this.getApellido();
    }
}
```

Java – super

```
package ejemplo;
 public class Alumno extends Persona {
    …
    public String toString(){
        return super.toString() + this.getLegajo();
    }
}
```

- Si pongo this.toString() o solo toString() hace referencia al mismo método de forma recursiva (se cuelga)

Java – super vs super()

- Entonces: la palabra **super** se usa para enviar mensajes (o acceder a variables) al objeto "actual", pero a métodos definidos en la superclase.

Java – super vs super()

- A diferencia de **super**, la palabra clave **super()** se utiliza en constructores para referenciar a constructores de la superclase.
- Se usa para invocar a un constructor de la superclase desde el cuerpo del constructor de la subclase.
- Se debe usar en la primera línea del cuerpo del constructor

Java – this, this(), super, super()

```
public class Persona{
    int dni;
    public void setDni(int dni){
        this.dni = dni;
    }
    public Persona(){
        //hace algo
    }
    public Persona(int dni){
        this();
        this.setDni(dni);
    }}
```

Java – this, this(), super, super()

```
public class Alumno extends Persona{
    int legajo;
    public Alumno(){
        super();
    }
    public Alumno(int dni){
        super(dni);
    }
    public Alumno(int dni, int legajo){
        this(dni);
        this.setLegajo(legajo);
    }
    public Alumno(String apellido){
        this();
        super.setApellido(apellido);
    }}
```

Sobrescritura y sobrecarga

- ¿Qué pasa si en una subclase se define un método que tenga exactamente la misma firma (nombre, tipo de retorno, y lista de argumentos (orden, cantidad y tipo))

  que un método de la superclase?:

  SOBREESCRITURA

- La sobreescritura permite definir un método en una subclase, que tenga exactamente la misma firma que un método de la superclase.

Recordemos: Sobrescritura de métodos

- Cualquier método que se herede (no privado y no final) de una superclase puede ser sobreescrito en las subclases.

- Los métodos sobreescritos en una subclase, deben tener el mismo nombre, la misma lista de argumentos (en cuanto a tipo y orden) y el mismo tipo de retorno que los de la clase padre.

Recordemos: Sobrescritura de métodos

- El nivel de acceso de un método sobreescrito debe ser igual o menos restrictivo que el declarado en la superclase. Por ejemplo: si en la superclase, el método es declarado public, entonces el método sobrescrito en la subclase debe declararse public. Si en la superclase el método es declarado protected o default, en la subclase puede declararse public.
- Los métodos sobreescritos deben disparar las mismas excepciones o subclases de las excepciones disparadas por el método de la superclase. No pueden disparar otras excepciones.

Sobrecarga de métodos

¿Qué pasa si en una misma clase o en una subclase se define un método que tenga exactamente el mismo nombre, tipo de retorno con diferente lista de argumentos? (orden, cantidad y tipo): **SOBRECARGA**.

Sobrecarga y sobreescritura

```
package ejemplo;
 public class Persona {
    . . .
    public String toString(){
        return this.getApellido();
    }
}
```

- Sobreescribe el método toString de la clase Object

Sobrecarga y sobreescritura

```
package ejemplo;
 public class Alumno extends Persona {
    public String toString(){
        //sobreescritura
        return super.toString() + " " + this.getLegajo();
    }
    public String toString(String inicio){
        //sobrecarga
        return inicio + super.toString() + " " + this.toString();
    }
}
```

Sobrecarga y sobreescritura

```
package ejemplo;
 public class Test {
    public static void main(String args[]){
        Alumno a = new Alumno();
        a.setApellido("Dominguez");
        a.setLegajo(14);
        System.out.println(a);
        System.out.println(a.toString("Alumno: "));
    }
}
```

static, final y abstract

- static denota “clase” o “estático”
- Un método static es un método de clase
- Una variable static es una variable de clase (valor común para todas las instancias)
- Se usa en clases abstractas (ej: Math)

static, final y abstract

- final denota “final” o “inmutable”
- final puede usarse para métodos, variables y clases
- Una clase final no puede tener subclases
- Un método final es un método que no puede ser sobreescrito en una subclase
- Una variable final es una variable que no puede ser modificada ni sobreescrita
- Una variable puede definirse como static y final: **CONSTANTE**

static, final y abstract

- abstract denota “abstracto”
- Una clase abstract no puede ser instanciada
- Un método abstract es un método que debe ser sobreescrito y solo puede declararse en clases abstractas
- Una clase que herede a una abstracta debe implementar todos sus métodos abstractos o declararse también abstracta
- Una clase o método no puede ser declarado final y abstract al mismo tiempo

Interfaces

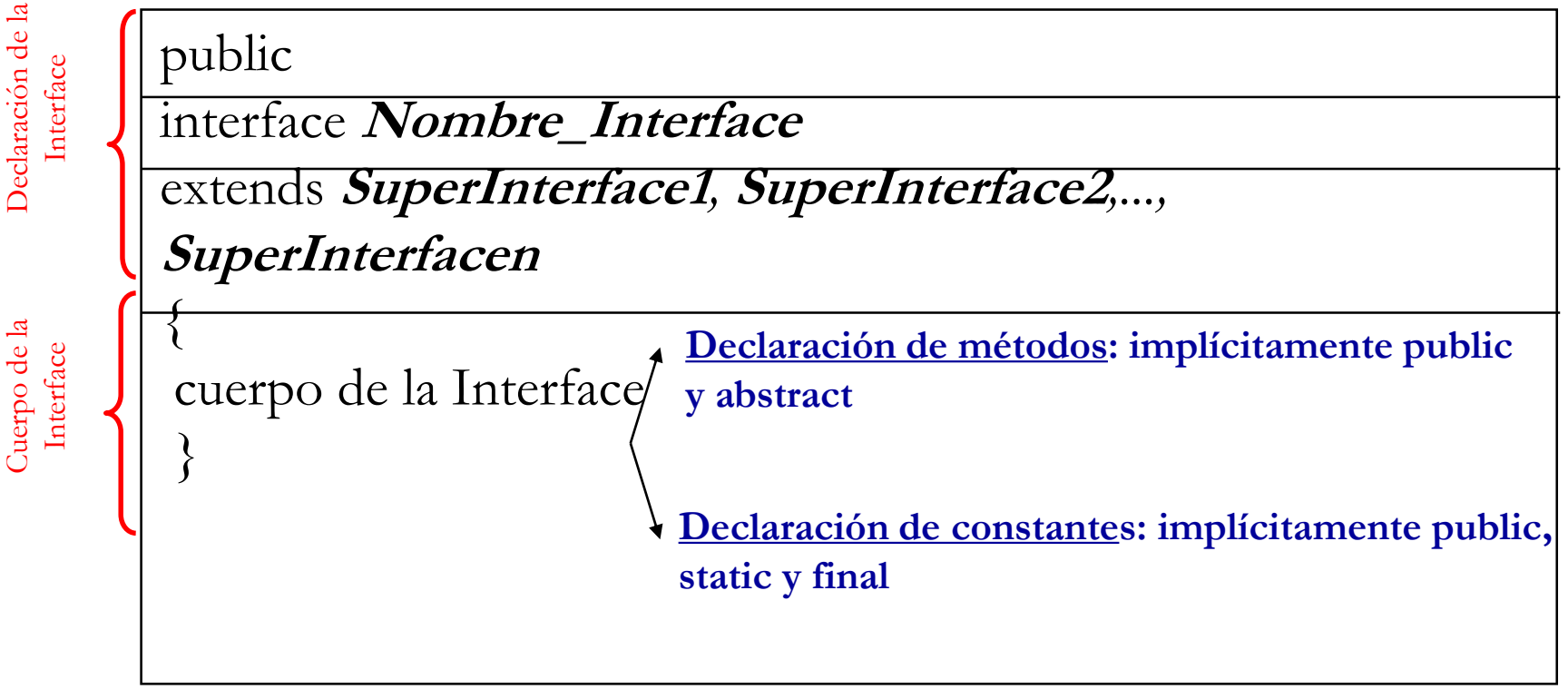
Conceptualmente una interface es un dispositivo o sistema que permite
que entidades no relacionadas interactúen.

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

Interfaces

- Una interface permite a objetos no relacionados interactuar
.
- Una interface es un nuevo tipo de datos.
- Una interface puede extender múltiples interfaces. Por lo tanto, se tiene herencia múltiple de interfaces.
- Una interface establece qué debe hacer la clase que la implementa sin especificar el cómo.
- Una clase que implementa una interface hereda las constantes y debe implementar cada uno de los métodos declarados en la interface.
- Una clase puede implementar más de una interface y de esta manera provee un mecanismo similar a la herencia múltiple.

Interfaces. Declaración



Interfaces. Declaración

- El especificador de acceso ***public***, establece que la interface puede ser usada por cualquier clase o interface de cualquier paquete.

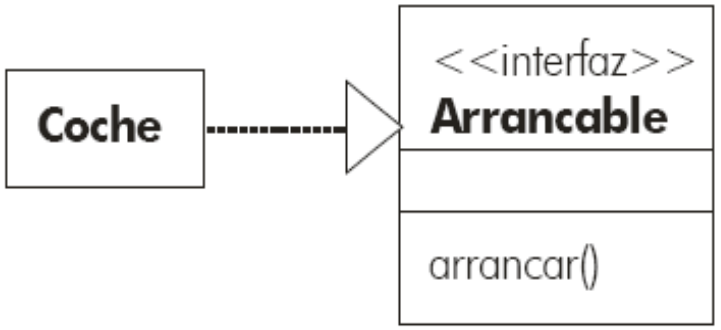
- Si se omite el especificador de acceso, la interface solamente podría ser usada por las clases e interfaces contenidas en el mismo paquete que la interface declarada (visibilidad de default).

- Una interface puede extender múltiples interfaces. Por lo tanto, se tiene herencia múltiple de interfaces.
- Ejemplo: public interface UnaInter extends OtraInter1, OtraInter2, ...

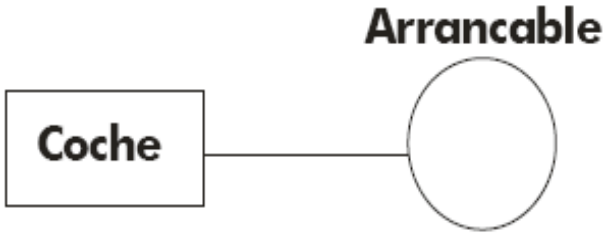
- Una interface hereda todas las constantes y métodos de sus *SuperInterfaces*.

```
interface arrancable(){
   boolean motorArrancado=false;
   void arrancar();
   void detenerMotor();
}
```

```
class Coche extends vehiculo implements arrancable {
   public void arrancar (){
         ....
   }
   public void detenerMotor(){
         ....
   }
```



Forma completa



Forma abreviada

Interfaces. Ejemplo

Declaración de la Interface

Cuerpo de la Interface

```
public interface InstrumentoMusical
{    void sonar();
     String queEs();
     void afinar();
     public long UN_SEGUNDO=1000;
     public long UN_MINUTO=60000;
}
```

Declaración de Métodos

Declaración de Constantes

Interfaces. Ejemplo

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

- La interface *InstrumentoMusical* define tres métodos y dos constantes.

- Las clases que implementen *InstrumentoMusical* deberán implementar los métodos sonar(), queEs() y afinar() y podrán usar las constantes UN_SEGUNDO y UN_MINUTO.

- Los métodos de una interface son implícitamente public y abstract; las constantes son implícitamente public y final

Un ejemplo de una Interface (Continuación)

```
public interface InstrumentoMusical {
void sonar();
String queEs()
void afinar();
public long UN_SEGUNDO=1000;
public long UN_MINUTO=60000;
}
```

La claúsula **implements** indica que la clase se ***ajuste*** a la interface y por tal motivo implementa todos sus métodos

IMPLEMENTS

```
class IntrumentoDeCuerdas{
public void sonar(){
 System.out.println("Sonar Cuerdas");
}
public String queEs(){
  return "Instrumento de Cuerdas";
}
public void afinar(){….}
}
```

```
class InstrumentoDeViento {
public void sonar(){
 System.out.println("Sonar Vientos");
}
public String queEs() {
  return "Instrumento de Viento";
}
public void afinar(){…}
}
```

```
class IntrumentoDePercusion {
public void sonar(){
   System.out.println("Sonar Percusión");
}
public String queEs() {
  return "Instrumento de Percusión";
}
public void afinar(){…}
}
```

EXTENDS

```
class Flauta {
public void sonar(){…}
public String queEs(){….}
}
```

```
class Oboe {
public void sonar(){…}
public String afinar(){..}
}
```

Interface y Herencia Múltiple

- Las interfaces no tienen almacenamiento asociado ya que no tienen implementación, por lo tanto no hay problemas si se combinan.

- 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.**

- En Java, una clase puede implementar tantas interfaces como desee. Así se logra un mecanismo similar a la ***herencia múltiple*** de clases.

Interfaces y herencia múltiple

La clase Heroe combina la clase concreta Accion con las interfaces PuedeBoxear, PuedeNadar y PuedeVolar. Por lo tanto, un objeto Heroe es también un objeto Accion, PuedeBoxear, PuedeNadar y PuedeVolar

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

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

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

```
class Accion {
     public void boxear(){....}
}
```

```
class Heroe extends Accion implements
PuedeBoxear, PuedeNadar,

PuedeVolar{
 public void nadar(){….}
 public void volar(){}
}
```

Interfaces y clases abstractas

- De una clase abstracta no es posible crear instancias.

- Una clase abstracta es ideal para representar un concepto, para agrupar un conjunto de clases a través de una interfaz común. Define un tipo de dato genérico. En este sentido es similar a una Interface.

- Todas las subclases de una clase abstracta tienen una implementación particular de un conjunto de métodos (abstractos) que definen la interfaz común entre las subclases.

Interfaces y clases abstractas

- La interface InstrumentoMusical, podría haberse definido como una clase abstracta, con tres métodos abstractos: hacerSonar(), queEs() y afinar(). Las subclases concretas, proveerán la implementación apropiada a cada uno de ellos.

- Tanto las interfaces como las clases abstractas proveen una estructura organizacional.

- Las interfaces tienen una jerarquía propia, independiente y más flexible que la de las clases, ya que tienen permitida la herencia múltiple.

Interfaces y clases abstractas

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

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

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

Una clase solo puede extender una clase (concreta o abstracta). Java ***no soporta herencia múltiple de clases***, por lo tanto si se quiere que una clase sea además del tipo de la clase base de otro tipo diferente, entonces es necesario usar interfaces.

Interface Cloneable

Un interface como hemos estudiado declara un conjunto de funciones, pero sin implementarlas. El interface *Cloneable* es muy simple ya que no define ninguna función.

```
public interface Cloneable { }
```

Una clase que implemente este interface le indica al método *clone* de la clase base *Object* que puede hacer una copia miembro a miembro de las instancias de dicha clase.

Si una clase no implementa esta interface, e intenta hacer una duplicación del objeto a través de la llamada al método *clone* de la clase base *Object*, da como resultado una excepción del tipo *CloneNotSupportedException*.

Interface Cloneable
Duplicación de objeto simple

La clase base *Object* de todas las clases en el lenguaje Java, tiene una función miembro denominada *clone*, que se redefine en la clase derivada para realizar una duplicación de un objeto de dicha clase.

Sea la clase *Punto* ya estudiada en prácticas anteriores. Para hacer una copia de un objeto de esta clase, se ha de agregar a la misma el siguiente código:

- se ha de implementar el interface *Cloneable*

Interface Cloneable
Duplicación de objeto simple

- se ha de redefinir la función miembro *clone* de la clase base *Object*

```
public class Punto implements Cloneable{
      private int x; private int y;
      //constructores ...
      public Object clone(){
        Object obj=null;
        try{
              obj=super.clone(); }
        catch (CloneNotSupportedException ex){
              System.out.println(" no se puede duplicar"); }
         return obj;
      }
//otras funciones miembro
}
```

Interface Cloneable
Duplicación de objeto simple

En la redefinición de *clone*, se llama a la versión *clone* de la clase base desde **super**.

Esta llamada se ha de hacer forzosamente dentro de un bloque **try... catch**, para capturar la excepción *CloneNotSuportedException* que nunca se producirá si la clase implementa el interface *Cloneable*. Como vemos la llamada a la versión *clone* de la clase base devuelve un objeto de la clase base *Object*, que es a su vez devuelto por la versión *clone* de la clase derivada.

Para crear un objeto *pCopia* que es una copia de otro objeto *punto* se escribe.

- Punto punto=new Punto(20, 30);
- Punto pCopia=(Punto)punto.clone();

El casting es necesario ya que *clone* devuelve un objeto de la clase base *Object* que ha de ser casteado a la clase *Punto*.

Interface Cloneable Duplicación de objeto compuesto

Volvamos de nuevo sobre la clase *Rectangulo* estudiada en prácticas anteriores.

Los miembros dato de la clase *Rectangulo* son un objeto de la clase *Punto*, el *origen*, y las dimensiones: *ancho* y *alto* del rectángulo.

Veamos ahora el código que tenemos que agregar a la clase *Rectangulo* para que se puedan efectuar copias de los objetos de dicha clase.

Interface Cloneable Duplicación de objeto compuesto

```
public class Rectangulo implements Cloneable{
    private int ancho ;     private int alto ;       private Punto origen;
    //los constructores
    public Object clone(){
      Rectangulo obj=null;
      try{
            obj=(Rectangulo)super.clone();
      }
      catch(CloneNotSupportedException ex){
             System.out.println(" no se puede duplicar");
      }
      obj.origen=(Punto)obj.origen.clone();
      return obj;
    }       //otras funciones miembro }
```

Interface Cloneable Duplicación de objeto compuesto

Un objeto de la clase *Rectangulo* contiene un subobjeto de la clase *Punto*. En la redefinición de la función miembro *clone* de la clase *Rectangulo* se ha de efectuar una duplicación de dicho sub-objeto llamando a la versión *clone* definida en la clase *Punto*.

Recuérdese que la llamada a *clone* siempre devuelve un objeto de la clase base *Object* que ha de ser promocionado (casting) a la clase derivada adecuada.

Veamos ahora, como se crea un objeto de la clase *Rectangulo* y se duplica. Para crear un objeto *rCopia* que es una copia del objeto *rect* de la clase *Rectangulo*, se escribe.

- Rectangulo rect=new Rectangulo(new Punto(0, 0), 4, 5);
- Rectangulo rCopia=(Rectangulo)rect.clone();

Java Default Method

Los Default Method o Métodos default permiten agregar métodos implementados en las interfaces.

Antes de la versión 8 de Java era totalmente imposible definirles cuerpo a los métodos de las interfaces ya que nos marcaba error de compilación.

Para agregar un método default solo será necesario marcar un método con el operador default , veamos un ejemplo de cómo se implementan:

Los métodos default se utilizan cuando es necesario implementar un cuerpo sin la necesidad de tener que implementar por separado una clase abstracta, además los métodos por default se pueden sobrescribir en caso de que la implementación por default no cumpla con lo que requerimos

Java Default Method

```
package com.osb.defaultmethod;

import java.util.List;

/**
 * @author Oscar Blancarte
 */
public interface IOrder {

    public List<IOrderItem> getOrderItem();

    public default double getTotal(){
        double total = 0;
        for (IOrderItem item : getOrderItem()) {
            total += item.getTotal();
        }
        return total;
    }

    public default void addOrderItem(IOrderItem orderItem){
        getOrderItem().add(orderItem);
    }
}
```

Java Static Import

Java Static Import es una de las características menos conocidas del core del lenguaje, pero a veces puede ser muy útil. La mayoría de los desarrolladores usan la palabra reservada "import" para importar las diferentes clases que va a utilizar en su programa.

Existen situaciones en el que el uso de los imports no es suficiente. Por ejemplo en la siguiente situación en la que se usa la clase Math para realizar diferentes cálculos.

```
package com.arquitecturajava;

import java.util.ArrayList;

public class PrincipalStatic {

public static void main (String[] args) {

ArrayList<String> lista= new ArrayList<String>();
lista.add("hola");
lista.add("que");
lista.add("tal");
double resultado= Math.pow(2, 2);
double resultado2= Math.pow(3,3);
```

Java Static Import

Continuamente tenemos que escribir la palabra “Math” para invocar los diferentes métodos estáticos de la clase. Los Java Static Import, permiten solventar este problema. Ya que al importar una clase de forma estática será suficiente con invocar los métodos.

```
package com.arquitecturajava;

import java.util.ArrayList;
import static java.lang.Math.*;
public class PrincipalStatic3 {

 public static void main (String[] args) {

 ArrayList<String> lista= new ArrayList<String>();
 lista.add("hola");
 lista.add("que");
 lista.add("tal");
 double resultado= pow(2, 2);
 double resultado2= pow(3,3);
 double resultado3= PI*2;
 System.out.println(resultado);
```

Java Default Method

Ahora no es necesario invocar continuamente a la clase Math, de forma similar se puede aplicar sobre System.out:

```
package com.arquitecturajava;

import java.util.ArrayList;
import static java.lang.Math.*;
import static java.lang.System.*;
public class PrincipalStatic4 {

public static void main (String[] args) {

ArrayList<String> lista= new ArrayList<String>();
lista.add("hola");
lista.add("que");
lista.add("tal");
double resultado= pow(2, 2);
double resultado2= pow(3,3);
double resultado3= PI*2;
out.println(resultado);
out.println(resultado2);
out.println(resultado3);
```

@notaciones

Las anotaciones tienen una serie de usos, entre ellos:

- **Información para el compilador**: El compilador puede utilizar las anotaciones para detectar errores o suprimir las advertencias.
- **Procesamiento en tiempo de compilación y tiempo de implementación**: Las herramientas de software pueden procesar información de anotaciones para generar código, archivos XML, etc.
- **Procesamiento de tiempo de ejecución:** algunas anotaciones están disponibles para ser examinadas en tiempo de ejecución.

@notaciones. Formato

En su forma más simple, una anotación se parece a la siguiente:

@Entidad

El carácter de signo (@) indica al compilador que lo que sigue es una anotación. En el siguiente ejemplo, el nombre de la anotación es Override:

```
@Override
void mySuperMethod() { ... }
```

@notaciones

La anotación puede incluir elementos, que pueden ser nombrados o sin nombre, y hay valores para esos elementos:

```
@Author(
   name = "Benjamin Franklin",
   date = "3/27/2003"
)
class MyClass() { ... }
```

or

```
@SuppressWarnings(value = "unchecked")
void myMethod() { ... }
```

Si sólo hay un elemento denominado valor, el nombre puede omitirse, como en:

```
@SuppressWarnings("unchecked")
void myMethod() { ... }
```

@notaciones

Si la anotación no tiene elementos, los paréntesis se pueden omitir, como se muestra en el ejemplo anterior de @Override.

También es posible utilizar múltiples anotaciones en la misma declaración:

```
@Author(name = "Jane Doe")
@EBook
class MyClass { ... }
```

Si las anotaciones tienen el mismo tipo, entonces se denomina anotación repetitiva:

```
@Author(name = "Jane Doe")
@Author(name = "John Smith")
class MyClass { ... }
```

Las anotaciones repetidas se admiten desde la versión de Java SE 8