

# Paquetes Especificadores de Acceso

# Paquetes y Espacios de Nombres

- Un **paquete en JAVA** es una colección de componentes de software (clases, interfaces, tipos enumerativos, anotaciones) con nombre. Los paquetes son útiles para **agrupar componentes de software** relacionados y definir un **espacio de nombres común** a las entidades contenidas en él.
- Las clases e interfaces esenciales (core) de la plataforma JAVA están ubicadas en un paquete cuyo nombre comienza con **java** y luego una serie de subnombres, resultando en un nombre jerárquico.
  - Las clases e interfaces fundamentales de JAVA están ubicadas en el paquete **java.lang**.
  - Las clases e interfaces utilitarias están en el paquete **java.util**.
  - Las clases e interfaces usadas para realizar I/O están en el paquete **java.io**.
  - Las clases e interfaces usadas para establecer conexiones con hosts remotos, que usan protocolos de comunicación de redes, etc. están en el paquete **java.net**.
  - Las clases e interfaces usadas para realizar funciones matemáticas y trigonométricas están en el paquete **java.math**.
  - Las clases e interfaces usadas para construir interfaces gráficas de usuario están en **java.awt**.

# Paquetes y Espacios de Nombres

- Los paquetes a su vez pueden contener subpaquetes, como por ej. el paquete `java.awt` contiene los subpaquetes `java.awt.event` y `java.awt.image`; el paquete `java.lang` contiene los subpaquetes `java.lang.reflect` y `java.util.regex`.
- Las extensiones de la plataforma JAVA están disponibles **a través de módulos** cuyos nombres comienzan con **jdk**, por ejemplo paquetes estándares como por ej. `org.w3c.dom.html` que implementan estándares definidos por la W3C está incluido en el módulo **`jdk.xml.dom`**.
- Todas las clases, interfaces, tipos enumerativos y anotaciones tienen un **nombre simple o no calificado** y un **nombre completo o calificado**. El **simple** es el que usamos para **definir la clase** y el **completo o calificado** es el que incluye como **prefijo el nombre del paquete** al que pertenece la clase. Por ej. la clase `String` es parte del paquete `java.lang`, su nombre simple es `String` y su nombre completo es `java.lang.String`.

# Nombres Únicos de Paquetes

Una de los propósitos más importantes de los paquetes es el de **dividir el espacio de nombres global de JAVA** y **evitar colisiones de nombres entre clases**. Por ej. el nombre del paquete permite diferenciar la interface **java.util.List** de la clase **java.awt.List**.

**¿Cómo hacemos para que los nombres de los paquetes sean distintos?**

Un esquema de nombres posible podría ser **usar el nombre invertido del dominio de Internet como prefijo de todos los nombres de paquetes**.

Por ej. si consideramos el nombre del dominio de Internet del LINTI, **linti.unlp.edu.ar**, los nombres de paquetes comenzarán con **ar.edu.unlp.linti**.

Luego, con algún criterio se debe decidir cómo particionar el nombre después de **ar.edu.unlp.linti**.

Tenemos que tener en cuenta que como el nombre del dominio de Internet es único, ningún otro desarrollador u organización que respete esta regla definirá un paquete con el mismo nombre.

**ar.edu.unlp.linti.graficos**

**ar.gov.gba.ec.rrhh.sueldos**

**ar.com.afip.estadisticas.reportes**

# Nombres Únicos de Paquetes

## Pautas para elegir nombres únicos de paquetes:

- Si somos desarrolladores de clases que serán usadas por otros programadores y que las combinarán con múltiples clases desconocidas por nosotros, es importante que los nombres de los paquetes sean globalmente únicos. Por ej. desarrolladores de compañías de software, de comunidades de software libre, etc.
- Si estamos desarrollando una aplicación JAVA, cuyas clases no están concebidas para ser usadas por otros programadores fuera de nuestro equipo de trabajo, podemos elegir un esquema de nombres de paquetes que se ajuste a nuestra conveniencia. En este caso conocemos el nombre completo del conjunto de clases que nuestra aplicación necesita para el *deployment* y no tendremos imprevistos por conflictos de nombres.

# La palabra clave *package*

En JAVA las clases e interfaces típicamente se agrupan en paquetes usando como primer *token* del archivo fuente la palabra clave **package** seguido por el nombre del paquete.

```
package ar.com.laplataautos;  
public class Vehiculo {  
    private String marca;  
    private String nroMotor;  
    public String getMarca(){  
        return marca;  
    }  
    public String getNroMotor(){  
        return nroMotor;  
    }  
}
```

Si se omite la palabra clave **package** en la definición de una clase, la misma se ubicará en el paquete predeterminado, conocido como **default package**. Este mecanismo resulta útil sólo para realizar pequeñas pruebas o testeos de algoritmos o lógica, sin embargo no es una buena práctica de programación.

# Importar Tipos de Datos

El mecanismo predeterminado para incluir nombres de clases o interfaces existentes es usar el **nombre completo** de la clase o interface.

Por ej. si estamos escribiendo código que manipula un archivo vamos a necesitar usar la clase **File** que pertenece al paquete **java.io**, entonces deberíamos escribir **java.io.File** cada vez que usamos la clase.

¿Cuándo podemos usar el **nombre simple** de un tipo de datos?

- Si usamos clases e interfaces del paquete **java.lang**. *Importación automática*
- Si usamos clases e interfaces que están definidas en el mismo paquete de la clase o interface que estamos escribiendo.
- Si incluimos tipos al espacio de nombres con la sentencia **import**.

Es posible explícitamente importar tipos de datos desde otros paquetes al espacio de nombres actual usando la sentencia **import**.

Importación de un tipo

```
import ar.edu.unlp.linti.graficos.Rectangulo;  
import java.io.File;  
import java.io.PrintWriter;
```

Es posible usar el nombre simple de las clases por ej.

Los tipos son importados a medida que se necesitan

Importación de tipos por demanda

```
import ar.edu.unlp.linti.graficos.*;  
import java.io.*;
```

Rectangulo, File,  
PrintWriter

La sintaxis de importación de tipos por demanda no se aplica a subpaquetes.



# Colisión de Nombres

La sentencia **import** puede generar conflictos.

Consideremos los paquetes **java.util** y **java.awt**.

Ambos paquetes contienen el tipo **List**: **java.util.List** es una interface comúnmente usada y **java.awt.List** es una clase.

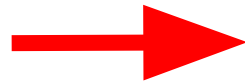
```
package pruebas;  
import java.util.List;  
import java.awt.List;
```



¿Se puede hacer?

```
public class ConflictoDeNombres {  
    //TODO  
}
```

```
package pruebas;  
import java.util.*;  
import java.awt.*;
```



¿Se puede hacer?

```
public class ConflictoDeNombres {  
    //TODO  
}
```

```
package pruebas;  
import java.util.*;  
import java.awt.*;  
public class ConflictoDeNombres {  
    List l;  
}
```



¿A qué List hace referencia?



# Colisión de Nombres

## ¿Cómo lo resolvemos?

Si la interface **java.util.List** es más usada que la clase **java.awt.List** se podría combinar importación de un tipo e importación por demanda y así quitamos la ambigüedad cuando hacemos referencia al tipo **List**.

```
package pruebas;
import java.util.List;
public class ConflictoDeNombres {
    List l;
    java.awt.List l2;
}
```

```
package pruebas;
import java.util.*;
public class ConflictoDeNombres {
    List<String> l=new ArrayList<String>();
    java.awt.List l2=new java.awt.List(4);
}
```

En estos casos a la clase **List** de **java.awt** la tenemos que usar con el nombre completo **java.awt.List** y a la interface de **java.util** con el nombre simple.

# Importar Miembros Estáticos

Es posible **importar miembros estáticos** de clases e interfaces usando la palabra clave **import static**.

El **import static** tiene 2 formas:

- Importación de un miembro estático
- Importación de miembros estáticos por demanda

Consideremos la siguiente situación: necesitamos imprimir texto en pantalla usando la salida estándar, **System.out**. Usamos la **importación de un miembro estático**:

```
package pruebas;  
import static java.lang.System.out;  
public class ImportOUT {  
    public static void main(String[] args) {  
        out.print("hola");  
    }  
}
```

Consideremos la siguiente situación: hacemos uso exhaustivo de funciones trigonométricas y otras operaciones de la clase **Math**. Usamos **importación de miembros estáticos por demanda**:

Nos evitamos escribir **System.out.print( )**

Podemos escribir expresiones concisas sin tener que usar **Math** como prefijo de cada método estático de la clase **Math**.

```
package pruebas;  
import static java.lang.Math.*;  
import static java.lang.System.out;  
public class ImportOUTBis {  
    public static void main(String[] args) {  
        out.println(sqrt(abs(sin(90))) );  
        out.println("Valor de PI: "+PI);  
    }  
}
```

# Importar Miembros Estáticos

La importación estática **importa nombres**, NO un miembro específico con dicho nombre. JAVA soporta sobrecarga de métodos y también permite que una clase defina atributos con el mismo nombre que un método -> al importar un miembro estático, podríamos importar más de un miembro (métodos y atributos).

Consideremos el siguiente ejemplo:

```
package pruebas;  
import static java.util.Arrays.sort;  
public class SobrecargaImportEstatico {  
    public static void main(String args[]) {  
        String varones[]{"Juan", "Pedro", "Luis", "Ernesto"};  
        sort(varones);  
    }  
}
```

Importa el nombre **sort** no uno de los 18 métodos **sort()** definidos en la clase Arrays

El compilador analizando el tipo, cantidad y orden de los argumentos y determina cuál de los métodos **sort()** queremos usar.

# Importar Miembros Estáticos

```
package pruebas;
import java.util.ArrayList;
import static java.util.Arrays.sort;
import static java.util.Collections.sort;
public class SobrecargaImportEstatico {
    public static void main(String args[]) {
        Integer numeros[]={30,2,44,18,3,23};
        String varones[]{"Juan", "Pedro", "Luis", "Ernesto"};
        ArrayList<String> mujeres=new ArrayList<String>();
        mujeres.add("Elena");
        mujeres.add("Pilar");
        mujeres.add("Juana");
        sort(varones);
        sort(numeros);
        sort(mujeres);
    }
}
```

Es legal importar métodos  
estáticos con el mismo nombre,  
pertenecientes a clases  
distintas, siempre que los  
métodos tengan diferentes  
firmas.

Este código no tiene errores de compilación porque los métodos **sort()** definidos en la clase **Collections** tienen firma diferente a todos los definidos en la clase **Arrays**. Cuando usamos el nombre **sort** el compilador determina cuál de los 20 posibles métodos importados deseamos invocar, analizando el tipo, orden y cantidad de argumentos.

# Ubicación de los Paquetes

Un paquete está formado por múltiples archivos **.class**.

JAVA aprovecha la estructura jerárquica de directorios del SO y ubica todos los **.class** de un mismo paquete en un mismo directorio. De esta manera se resuelven:

- El nombre único del paquete: no existen 2 *paths* con el mismo nombre
- La búsqueda de los **.class** y **.java**, evitando que estén diseminados en todo el filesystem

Los nombres de paquetes se resuelven en directorios del SO: en el nombre del paquete se codifica el *path* de la ubicación de los **.class**.

```
package ar.edu.unlp.linti.graficos;      ar/edu/unlp/lini/graficos/Rectangulo.class
import java.awt.*;

public class Rectangulo extends Graphics implements Draggable {
}
```

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 una variable *static*, procede de la siguiente manera:

- Busca en los **directorios estándares**: donde está instalado el JRE y en el directorio actual.
- Recupera la variable de entorno **CLASSPATH** que contiene la lista de directorios usados como raíces para buscar los archivos **.class**.
- Toma el nombre del paquete de la sentencia **import** y reemplaza cada “.” por una barra “\” o “/” (según el SO) para generar un *path* a partir de las entradas del **CLASSPATH**.

El compilador JAVA procede de manera similar al “intérprete”.

Las versiones actuales del JSE configuran automáticamente la variable de entorno **CLASSPATH**.

# Archivos JAR

Es posible **agrupar múltiples paquetes en un único archivo**:

## Usando archivos JAR (Java ARchive)

El formato JAR usa el formato ZIP. Los archivos JAR son multiplataforma, son totalmente portables.

En los archivos JAR pueden incluirse además de archivos **.class** recursos como archivos de imágenes y audio, etc.

La distribución estándar del JSE contiene una herramienta que permite crear archivos JAR desde la línea de comando: el utilitario **jar**.

El “intérprete” JAVA se encarga de buscar, descompactar, cargar y ejecutar los **.class** contenidos en el JAR.

En el CLASSPATH se codifica el nombre real del archivo JAR

**CLASSPATH=.;C:\cursoJava\lab;C:\laboratorio;c:\java12\utiles.jar**

# Archivos JAR

El uso de archivos **JAR** es la opción recomendada para la **entrega de aplicaciones o de librerías de componentes**.

Una **aplicación empaquetada en un JAR** es un **archivo ejecutable JAVA** que es posible lanzar directamente desde el sistema operativo.

Los archivos **JAR** además de contener todos los paquetes con sus archivos **.class** y los **recursos de la aplicación**, contienen 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 que define el método `main()`.

**Manifest-Version: 1.0**

**Created-By:**

**Main-Class: capitulo4.paquetes.TestOut.class**

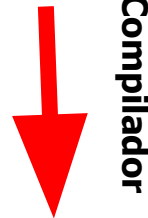


# Archivos JAVA

## Restricciones

```
package ar.edu.unlp.linti.graficos;  
public class Rectangulo extends Graphics implements Draggable { }  
class Helper { }
```

**Rectangulo.java**



**\ar\edu\unlp\linti\graficos\Helper.class**

**\ar\edu\unlp\linti\graficos\Rectangulo.class**

- El **archivo fuente JAVA** se llama **unidad de compilación** y tiene extensión **.java**.
- Cada **archivo fuente JAVA** puede contener a lo sumo una sola **clase o interface** declarada **public**. Si contiene clases o interfaces adicionales, éstas deben declararse no-public o package.
- El nombre del archivo fuente JAVA debe coincidir, incluyendo mayúsculas y minúsculas, con el nombre de la clase o interface declarada public.
- Si el archivo fuente contiene múltiples clases e interfaces, el compilador crea diferentes archivos **.class**. Cada uno de ellos tiene el nombre de la clase o interface y extensión **.class**.
- Es una buena práctica de programación definir una clase o interface por archivo fuente.



# Especificadores de Acceso

JAVA dispone de facilidades para proveer **ocultamiento de información**: el control de acceso define la accesibilidad a clases, interfaces y a sus miembros, estableciendo **qué está disponible** y **qué no** para los programadores que utilizan estas entidades.

Los **especificadores de acceso** determinan la accesibilidad a clases, interfaces y sus miembros y proveen **diferentes niveles de ocultamiento**.

El control de acceso lo podemos utilizar, para:

- Clases e interfaces de nivel superior
- Miembros de clases (métodos, atributos) y constructores

Uno de los factores más importantes que distingue un módulo bien diseñado de uno pobremente diseñado es el nivel de ocultamiento de sus datos y de otros detalles de implementación. Un módulo bien diseñado oculta a los restantes módulos del sistema todos los detalles de implementación separando su interface pública de su implementación.

**Desacoplamiento**

**Reusabilidad de Código**

# Especificadores de Acceso en Clases

En JAVA los especificadores de acceso en las declaraciones de clases se usan para determinar **qué clases están disponibles para los programadores**. Da lo mismo para interfaces y anotaciones.

Una clase declarada **public** es parte de la **API que se exporta** y está disponible mediante la cláusula **import**.

```
package gui;  
public class Control{  
    //TODO  
}  
  
import gui.Control;
```

¿Qué pasa si en el paquete **gui** tengo una clase que se usa de soporte para la clase `Control` y para otras clases del paquete `gui`?

La definimos de acceso **package** y de esta manera solamente puede usarse en el paquete `gui`. Es **privada del paquete**. Es razonable que los miembros de una clase de acceso *package* tengan también acceso **package**.

```
package gui;  
class Soporte{ //TODO}
```

La clase **Soporte** es **privada del paquete gui**: sólo puede usarse en el paquete `gui`



# Especificadores de Acceso en Métodos, Variables y Constructores

Son **reglas de control de acceso** que restringen el uso de las variables, métodos y constructores de una clase. Permiten que el programador de la clase determine **qué está disponible** para el programador que usa dicha clase y **qué no**.

Los **especificadores de control de acceso** son:

<b>public</b>	<b>protected</b>	<b>package</b> (no tiene palabra clave)	<b>private</b>
---------------	------------------	---	----------------

Más libre

Más restrictivo

El **control de acceso** permite **ocultar la implementación**. Separa la **interface** de la **implementación**, permite hacer cambios que no afectan al código del usuario de la clase.

En JAVA los **especificadores de acceso** se ubican delante de la definición de cada variable, método y constructor de una clase. El especificador solamente controla el acceso a dicha definición.

¿Qué pasa si a un miembro de una clase no le definimos especificador de acceso?

Tiene acceso por defecto, no tiene palabra clave y comúnmente se lo llama **acceso package o friendly o privado del paquete**. Implica que tienen acceso a dicho miembro solamente las clases ubicadas en el mismo paquete que él. Para las clases declaradas en otro paquete, es un miembro privado.

El acceso **package** le da sentido a agrupar clases en un paquete.

# Especificadores de Acceso en Métodos, Variables y Constructores

**public**

El atributo, método o constructor declarado **public** está disponible para **TODOS**.

```
package labo12;  
public class Auto{  
    public String marca;  
    public Auto() {  
        System.out.println("Constructor de Auto");  
    }  
    void arrancar(){  
        System.out.println("arrancar");  
    }  
}
```

¿Qué observan en este código?

```
import labo12.*;  
public class Carrera{  
    public Carrera() {  
        System.out.println("Constructor de Carrera");  
    }  
    public static void main(String[] args){  
        Auto a=new Auto();  
        System.out.println("Marca: "+ a.marca);  
        a.arrancar();  
    }  
}
```

# Especificadores de Acceso en Métodos, Variables y Constructores

## Privado del paquete (package)

Las variables, métodos y constructores declarados **privados del paquete** son accesibles sólo desde clases pertenecientes al mismo paquete donde se declaran.

```
package labo12;  
public class Auto {  
public String marca;  
Auto() {  
    System.out.println("Constructor de Auto");  
}  
void arrancar(){  
    System.out.println("arrancar");  
}  
}
```

¿Qué observan en el código?

¿Qué relación encuentran entre el especificador de acceso **package** y la herencia?

```
import labo12.*;  
public class Carrera{  
    public Carrera() {  
        System.out.println("Constructor de Carrera");  
    }  
  
    public static void main(String[] args){  
        Auto a=new Auto();  
        System.out.println("Marca: "+ a.marca);  
        a.arrancar();  
    }  
}  
  
.....  
public class Sedan extends Auto {
```

# Especificadores de Acceso en Métodos, Variables y Constructores

## protected

La palabra clave **protected** está relacionada con la **herencia**:

- Si se define una **subclase** en un **paquete diferente** al de la **superclase**, la subclase solo tiene acceso a los miembros definidos **public**.
- Si se define una **subclase** en el **mismo paquete** que la **superclase**, la subclase tiene acceso a todos los miembros declarados **public** y **package**.
- El autor de la **clase base** podría **determinar** que **ciertos miembros pueden ser accedidos por las subclases**, pero no por todo el mundo. Esto es **protected**.
- Además el acceso **protected** provee acceso **package**: las clases declaradas en el mismo paquete que el miembro **protected** tienen acceso a dicho miembro.

```
package labo12;  
public class Auto{  
    public Auto() {  
        System.out.println("Constructor de Auto");  
    }  
    void arrancar(){  
        System.out.println("arrancar");  
    }  
}
```

¿Qué observan en el código?

```
import labo12.*;  
public class Sedan extends Auto{  
    public Sedan() {  
        System.out.println("Constructor de Sedan");  
    }  
    public static void main(String[] args){  
        Sedan x=new Sedan();  
        x.arrancar();  
    }  
}
```

# Especificadores de Acceso en Métodos, Variables y Constructores

## protected

Analizaremos el acceso **protected** aplicado a variables.

```
package labo12;
public class Auto{
    protected String marca;
    protected String nroMotor;
    public Auto() {
        System.out.println("Constructor de Auto");
    }
    protected void arrancar() {
        System.out.println("arrancar");
    }
}
```

¿Qué observan en el código?

```
import labo12.*;
public class Sedan extends Auto{
    private String identificador;
    public Sedan() {
        System.out.println("Constructor de Sedan");
    }
    public void setIdentificador(Auto v){
        this.identificador=this.marca+v.nroMotor;
    }
}
```

**El acceso protegido privilegia la relación de herencia:** las subclases heredan todas las variables y los métodos protegidos de sus superclases, independientemente del paquete donde estén definidas las clases y las subclases. Además el acceso protegido también es acceso privado del paquete.

**Un miembro declarado protegido es parte de la API que se exporta:** debe mantenerse siempre, es un compromiso público de un detalle de implementación.

# Especificadores de Acceso en Métodos, Variables y Constructores

## private

Las variables, métodos y constructores declarados **private** solamente están accesibles para la clase que los contiene. Están disponibles para usar dentro de los métodos de dicha clase.

```
public class Postre {  
    private Postre() {  
        System.out.println("Constructor de Auto");  
    }  
}
```

```
public class Helado{  
    public static void main(String[] args){  
        Postre p=new Postre();  
    }  
}
```

¿Qué observan en el código? ¿Qué relación encuentran entre el especificador de acceso **private** y la herencia?

Los **métodos privados** funcionan como **utilitarios de la clase**, sólo pueden invocarse desde otros métodos de la clase donde se declaran, **no puedan reemplazarse** a través del mecanismo de herencia y las modificaciones en su código no “rompen” el código que hace uso de la clase.

Las **variables privadas** forman parte de la **implementación de la clase**, son de uso exclusivo de la misma, sólo pueden manipularse directamente en los métodos de la clase y pueden modificarse sin perjudicar el código que hace uso de la clase.

Una buena práctica es declarar **private** todo lo posible.

Y con los constructores privados ¿qué pasa?



# Control de Acceso y Herencia

La especificación de JAVA establece que una subclase hereda de sus superclases todos los atributos y métodos de instancia **accesibles**:

- Si la subclase está definida en el mismo paquete que la superclase, hereda todos los atributos y métodos de instancia **no-privados**.
- Si la subclase está definida en un paquete diferente que su superclase, hereda solamente los atributos y métodos de instancia **protegidos** y **públicos**.
- Los atributos y métodos de instancia **privados nunca son heredados**.

Los **constructores no se heredan**, se encadenan.

Podría ser confuso: "una subclase NO HEREDA los atributos y métodos de su superclase inaccesibles para ella" -> NO IMPLICA que cuando se crea una instancia de la subclase, no se use memoria para los atributos privados o inaccesibles definidos en la superclase.

Todas las instancias de una subclase incluyen una instancia COMPLETA de la superclase, incluyendo los miembros inaccesibles.

Como los miembros inaccesibles no pueden usarse en la subclase, decimos NO SE HEREDAN.

# 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

- 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**.

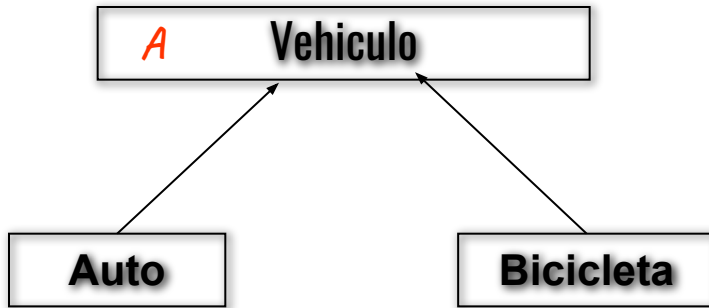
**abstract class Vehiculo {}**

- 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 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 *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() {  
        System.out.println("Vehiculo");  
    }  
    public abstract void arrancar();  
}
```

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

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

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

# 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.
- 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

```
public | "se omite"  
interface NombreInterface  
extends SuperInterface1, SuperInterface2, ..., SuperInterfacen  
{  
    cuerpo de la Interface  
}
```

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 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**, que se declaran **static** y **final**.
- Una interface NO puede ser instanciada, por lo tanto no define constructores.

# Ejemplo

## Definir una interface

### **public interface Centrabable**

```
{ 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 Centrabable** es una interface pública por lo tanto puede ser usada por cualquier clase e interface de cualquier paquete.
- La **interface Centrabable** define tres métodos de instancia. Los métodos son implícitamente **públicos**.
- Las clases que implementen **Centrabable** 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 Centrabable
{
    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.

# 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 sobreescrive 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 cláusulas throws que causen conflicto con alguno de los métodos que sobreescrive.

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

# Ejemplo (continuación)

## Implementar una interface

```
public class RectanguloCentrado extends Rectangulo implements Centrabale {  
    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.

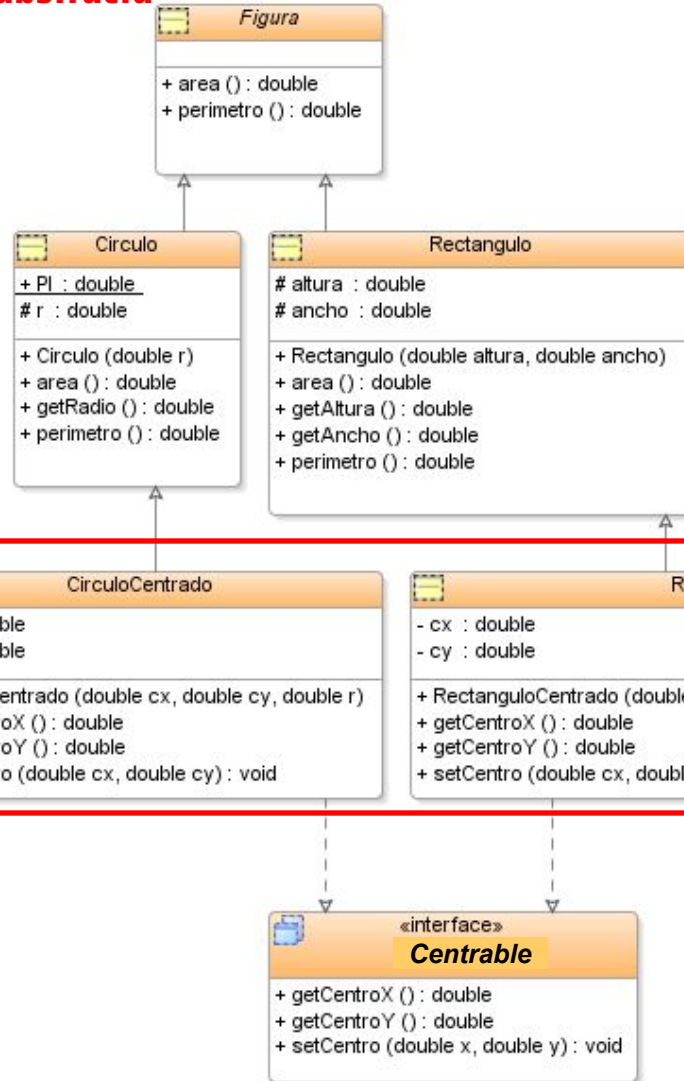
✓ Una clase que implementa una interface y **NO** provee una implementación para cada método de la interface debe declararse **abstract** (hereda los métodos abstractos y no los implementa).

✓ 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);
```

✓ **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 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){
```

tipo de una clase

```
        double areaTotal = 0;
```

```
        for(int i = 0; i < f.length; i++)
```

```
            areaTotal += f[i].area();
```

tipo de una interface

```
        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);
```

```
    }
```

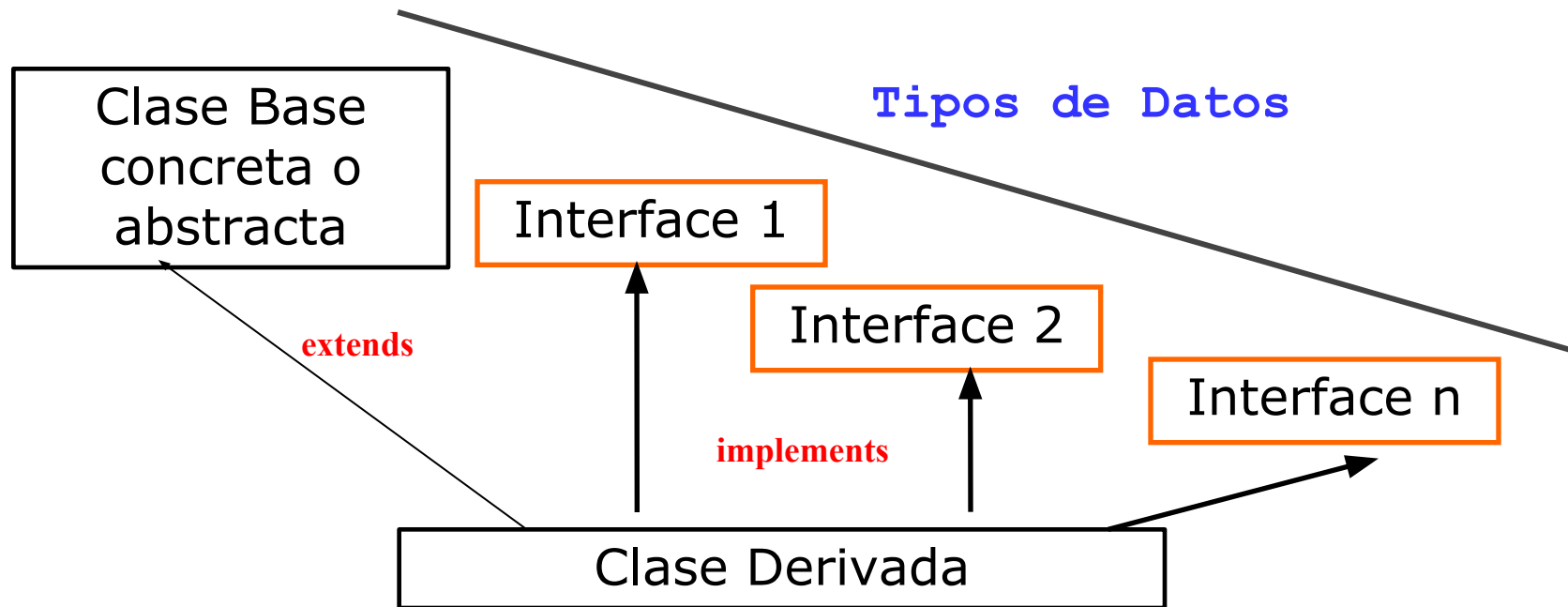
upcasting

*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 una HombreDeAccion
}

} // Fin de la clase Aventura
```

u  
p  
c  
a  
s  
t  
i  
n  
g

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.

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**.
- A partir de JAVA 8, las interfaces pueden proveer implementaciones para algunos métodos de instancia. 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 una interface 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 en forma segura agregar métodos no-abstractos sin romper el código de las clases que las extienden. **JAVA 8 soluciona el problema de incompatibilidad 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 subclasses aprovecharían (*skeletal implementation*). Patrón **Template Method**. Las interfaces de colecciones usan este patrón: **AbstractSet**, **AbstractList**, **AbstractMap**, etc.

```
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** 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**

- **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** pues son detalles de implementación, **ni métodos estáticos** pues 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.

# 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** ó 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 y que no implementan el método de default.

```
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. 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**

# Herencia Polimorfismo

# Herencia

Permite crear una **clase nueva** como un **subtipo de** una **clase existente**.

La **herencia** es una parte integral de Java, es el compilador el que hace la mayor parte del trabajo.

La **herencia es apropiada** solo cuando existe una **relación de subtipo genuina** entre la subclase y la superclase.

Es **seguro usar la herencia** dentro de un paquete, donde las **implementaciones de subclase y superclase** están bajo el control de los mismos programadores.

Hereda de  
Circulo las v.i y  
los métodos

```
public class Circulo {  
    public static final double PI= 3.14159;  
    public double r;  
    public Circulo(double r) { this.r = r; }  
    public static double radianesAgrados(double rads) {  
        return rads * 180 / PI;  
    }  
    public double area() {  
        return PI * r * r;  
    }  
    public double circunferencia() {  
        return 2 * PI * r;  
    }  
}
```

```
public class CirculoPlano extends Circulo {  
    public double cx, cy;  
    public CirculoPlano(double r, double x, double y)  
    {  
        super(r);  
        this.cx = x;  
        this.cy = y;  
    }  
    public boolean pertenece(double x, double y) {  
        double dx = x - cx, dy = y - cy;  
        double distancia = Math.sqrt(dx*dx + dy*dy);  
        return (distancia < r);  
    }  
}
```

Invoca al constructor de la  
superclase Circulo(r)

El atributo r se hereda de  
Circulo y se usa en  
CirculoPlano como si fuese  
propio



# La Clase Object

Definición del método **equals()** de la clase Object:

```
public boolean equals(Object obj) {  
    return (this == obj);  
}
```

Plaza plaza1 = **new** Plaza("Plaza Rocha");

Plaza plaza2 = **new** Plaza("Plaza Rocha");

System.out.println("Las plazas son iguales? "+ plaza1.equals(plaza2));

*¿Qué se imprime en pantalla?*      **Las plazas son iguales? false**

*¿Es el resultado esperado?*      **NO!! Esperamos true, dado que los valores de los objetos son iguales, ambos objetos representan a la plaza Rocha**

*¿Cuándo es apropiado sobrescribir el método **equals()** de **Object**?*

**Cuando las instancias de una clase tienen una noción de igualdad lógica que difiere de la identidad o referencias de esas instancias. Este es generalmente el caso de clases que representan valores, como por ejemplo las clases Integer, String de la API de JAVA.**

*¿Pasa lo mismo con las clases que tienen instancias controladas?* **NO**

# La Clase Object

El método **hashCode()** de la clase Object:

La sobrescritura del **hashCode()** está íntimamente relacionada con el framework de colecciones.

- **Toda clase que sobrescribe el método equals() debe sobrescribir el hashCode()** para asegurar un funcionamiento apropiado de sus objetos en contenedores basados en hashing como HashMap, HashSet y Hashtable.
- Una correcta sobrescritura del método **hashCode()** asegura que **dos instancias lógicamente iguales tengan el mismo hashcode** y en consecuencia las estructuras de datos basadas en hashing que almacenan y recuperan estos objetos, funcionarán correcta y eficientemente.

Es una **buena práctica sobrescribir** el método **hashCode()** cuando se sobreescribe el **equals()** y de esta manera se respeta el el contrato de la clase Object: *“si dos objetos son iguales según el método equals() entonces invocar al método hashCode() sobre cada uno de estos objetos, debe producir el mismo valor”*.

**ALWAYS OVERRIDE HASHCODE WHEN YOU OVERRIDE EQUALS**

**Effective JAVA (está en la biblioteca)**

# La Clase Object

El método **toString()** de la clase Object:

- La versión original del método `toString()` definida en `Object` produce un string formado por el nombre de la clase, seguido del símbolo `@` más la representación hexadecimal del código hash del objeto sobre el que se invoca al método `toString()`.

`capitulo2.laplata.Plaza@adbf1`

- El objetivo de este método es producir una representación textual, concisa, legible y expresiva del contenido del objeto.

*¿Cuándo se invoca al método `toString()`?*

Cuando se pasa un objeto como parámetro en los métodos `print()`, `println()`, `printf()`, `println()`; con el operador de concatenación. Internamente las distintas versiones del método `print()` invocan al método `toString()`.

El método `toString()` también es usado por el Debugger, lo que facilita la interpretación de los pasos de la ejecución de un programa

Es una buena práctica sobreescibir el método `toString()` para producir un resultado amigable que permite informar mejor sobre el objeto en cuestión.

# ¿Cómo implementa JAVA la herencia?

```
public class Vertebrado{  
    private int cantpatas;  
    public Vertebrado(){  
        System.out.println("Constructor de Vertebrado");  
    }  
    public void desplazar(){  
        System.out.println("Vertebrado.desplazar()");  
    }  
    public void comer(){  
        System.out.println("Vertebrado.comer()");  
    }  
}
```

```
public class Mamifero extends Vertebrado{  
    public Mamifero(){  
        System.out.println("Constructor de Mamifero");  
    }  
    public void comer(){  
        System.out.println("Mamifero.comer()");  
    }  
}
```

```
public class Perro extends Mamifero{  
    private String nom;  
    public Perro(){  
        System.out.println("Constructor de Perro");  
    }  
    public void setNombre(String n){this.nom=n;}  
    public String getNombre(){return nom;}  
    public void comer(){  
        System.out.println("Perro.comer()");  
    }  
    public void jugar(){  
        System.out.println("Perro.jugar()");  
    }  
    public static void main(String[] args){  
        Perro p=new Perro();  
    }  
}
```

# Encadenamiento de Constructores

- Un objeto de la clase derivada incluye un **objeto** de la clase base.
- Es esencial que el **objeto de la clase base se inicialice correctamente**. La manera de garantizarlo es realizar la inicialización en el constructor **invocando al constructor de la clase base** quién tiene el conocimiento apropiado para hacerlo. Sólo el constructor de la clase base tiene el conocimiento y los accesos necesarios para inicializar a sus miembros.
- Es fundamental que sean invocados **todos los constructores de la cadena de herencia** y de esta manera garantizar que el **objeto** quede construido correctamente.
- El compilador Java silenciosamente invoca al constructor nulo o de *default* de la clase base en el constructor de la clase derivada (si no se lo invocó explícitamente). La invocación es automática.

## ¿Cómo se construye un objeto Perro?

Recorriendo la jerarquía de herencia en forma ascendente e invocando al constructor de la superclase desde cada nivel de la jerarquía de clases: el constructor de **Perro** invoca al constructor de **Mamifero**, el de **Mamifero** al de **Vertebrado** y el de **Vertebrado** al de **Object**.

La creación de un objeto Perro involucra:

- Ejecutar el cuerpo del constructor de Object -> crear un objeto de tipo Object
- Ejecutar el cuerpo del constructor Vertebrado -> crear un objeto de tipo Vertebrado
- Ejecutar el cuerpo del constructor Mamifero -> crear un objeto de tipo Mamifero
- Ejecutar el cuerpo del constructor Perro -> crear un objeto de tipo Perro

Si NO se codifica un constructor, el compilador insertará un constructor de default que invoca al constructor de default de la superclase



# Encadenamiento de Constructores

## 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 con argumentos 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 apropiada de argumentos.

```
public class Vertebrado{
    private int cantpatas;
    public Vertebrado(int patas){
        System.out.println("Constructor de Vertebrado c/args");
    }
}
```

```
public class Mamifero extends Vertebrado{
    public Mamifero(int patas){
        super(patas);
        System.out.println("Constructor de Mamifero c/args");
    }
}
```

La invocación al constructor de la clase base es lo primero que debe hacerse en el cuerpo del constructor de la clase derivada

```
public class Perro extends Mamifero{
    private String nom;
    public Perro(){
        super(4);
        System.out.println("Constructor de Perro");
    }
    public static void main(String[] args){
        Perro p=new Perro();
    }
}
```

Si NO se invoca al constructor de la clase base explícitamente, el compilador insertará una invocación al constructor nulo. En nuestro ej. esta invocación implícita causaría un error de compilación.

Si una clase define constructores con argumentos y NO define al constructor nulo, todas sus subclasses deben definir constructores que invoquen explícitamente a los constructores con argumentos.

# Upcasting - Polimorfismo

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). 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 clase base.

**El upcasting es seguro:** la clase derivada es un super conjunto de la clase base, podría contener más métodos que la clase base, pero seguro contiene los métodos de la clase base.

*afinar(Nota)*

Vientos

Flauta

Oboe

Fagot

*Upcasting: Casting ascendente*

```
public class Musica{
    public static void dalepaly (Vientos v) {
        v.afinar(Nota.DO);
    }
    public static void main(String[] args){
        Flauta flauta=new Flauta();
        Fagot fagot=new Fagot();
        Oboe oboe=new Oboe();
        dalePlay(flauta);
        dalePlay(fagot);
        dalePlay(oboe);
    }
}
```

*afinar()* es polimórfico

*dalePlay()* acepta como parámetro una referencia a un objeto Vientos o a cualquier objeto derivado de Vientos

Una referencia a un objeto Flauta, Fagot y Oboe es pasada como parámetro al método dalePlay() sin hacer casting a Vientos

*afinar(Nota) afinar(Nota) afinar(Nota)*

El **Upcasting** de Flauta, Oboe y Fagot a **Vientos** limita la interface pública de estos objetos a la de **Vientos**

# Polimorfismo ¿Cómo se implementa en Java?

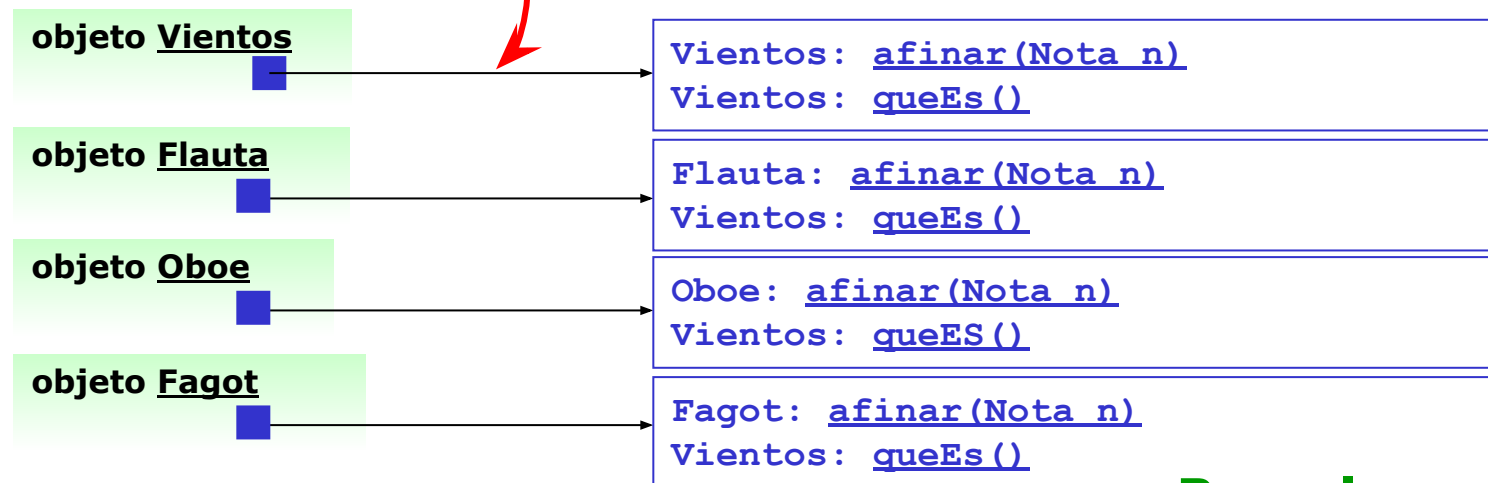
Cuando el “**intérprete**” ejecuta el código que produce el compilador busca el **afinar()** apropiado para invocar sobre cada objeto del arreglo. El intérprete chequea el tipo real del objeto referenciado por la variable y luego busca el método **afinar()** apropiado. El **intérprete** NO usa el método **afinar()** que está asociado estáticamente con la variable. **Dynamic Binding**

En JAVA la asociación entre la invocación a un método y el código que se ejecutará se resuelve **polimórficamente** a través del **binding dinámico**

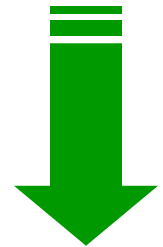
Objetos:

EJECUCIÓN

*Las invocaciones a afinar() y queEs() se resuelven mediante binding dinámico*



**Dynamic Binding**



**Resuelve métodos polimórficos**

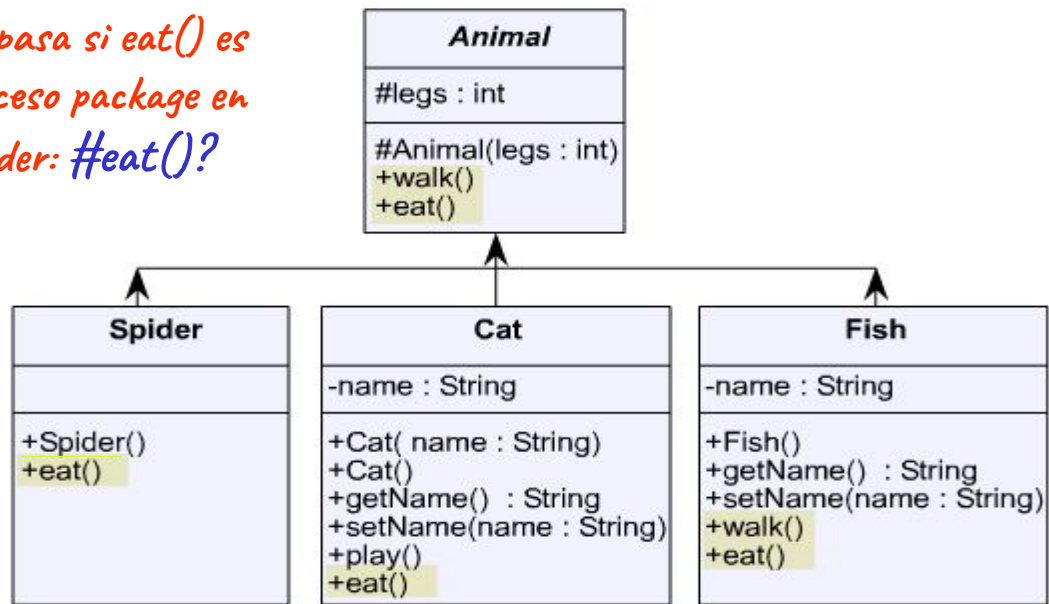
En Java el **Dynamic Binding** es automático.

Todos los **métodos de una clase final**, todos los **métodos declarados final**, **private** o **static** son invocados sin usar **Dynamic Binding**. Las invocaciones a estos métodos son candidatas a ser optimizadas (por ej. usar *inlining*)



# Sobreescritura y Control de Acceso

*¿Qué pasa si eat() es de acceso package en Spider: #eat()?*



paquete1

Animal, Spider, Cat y Fish son clases públicas

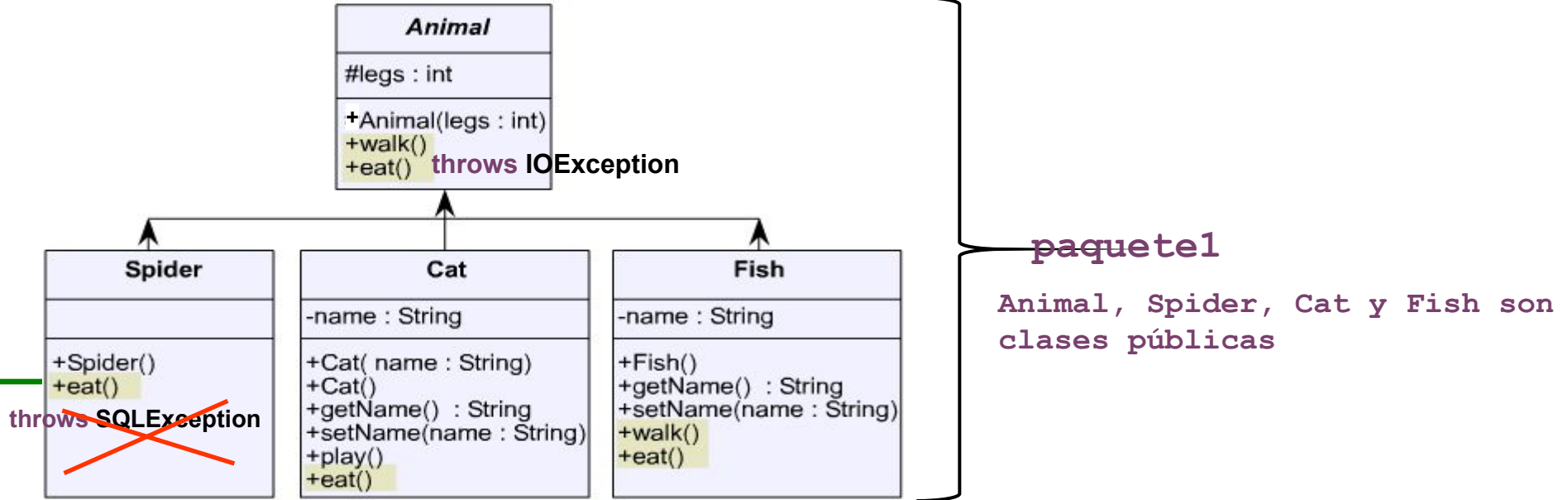
*eat()* es visible cuando se aplica sobre objetos Animal y deja de serlo cuando se aplica sobre una subclase (Spider), por lo tanto las subclases y las superclases dejan de tener una interface de comportamiento común

```

package paquete2;
import paquete1.*;
public class Fauna {
    private Animal[] ani={new Spider(), new Animal(4), new Cat("Violeta"), new Fish()};
    public static void main(String[] args) {
        Fauna f=new Fauna();
        for (int i=0;i<f.ani.length;i++){
            f.ani[i].eat();
        }
    }
}
  
```

Los métodos sobreescritos no pueden tener un control acceso más restrictivo que el declarado en la superclase

# Sobreescritura y Excepciones



```
package paquete2;
import paquete1.*;
import java.io.*;
public class Fauna {
    private Animal[] ani={new Spider(), new Animal(4), new Cat("Violeta"), new Fish()};
    public static void main(String[] args) throws IOException{
        Fauna f=new Fauna();
        for (int i=0;i<f.ani.length;i++){
            f.ani[i].eat();
        }
    }
}
```

Los métodos sobrescritos deben disparar las mismas excepciones que las del método de la superclase, subclases de ellas o ninguna excepción

# Reglas de Sobreescritura 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 declarados en la superclase. El tipo de retorno es *covariante*, de esta manera el tipo de retorno puede ser una subclase del tipo de retorno del método original.
- 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 sobreescrito en la subclase debe declararse **public**. Si en la superclase el método es declarado **protected** o **default**, en la subclase puede declararse **public**. ¿Y si es declarado **package**?
- Las **Excepciones** son clases especializadas que representan errores que pueden ocurrir durante la ejecución de un método. Los métodos sobreescritos deben disparar las mismas excepciones o subclases de las excepciones disparadas por el método original. NO pueden disparar otras excepciones.

# Constructores y Polimorfismo

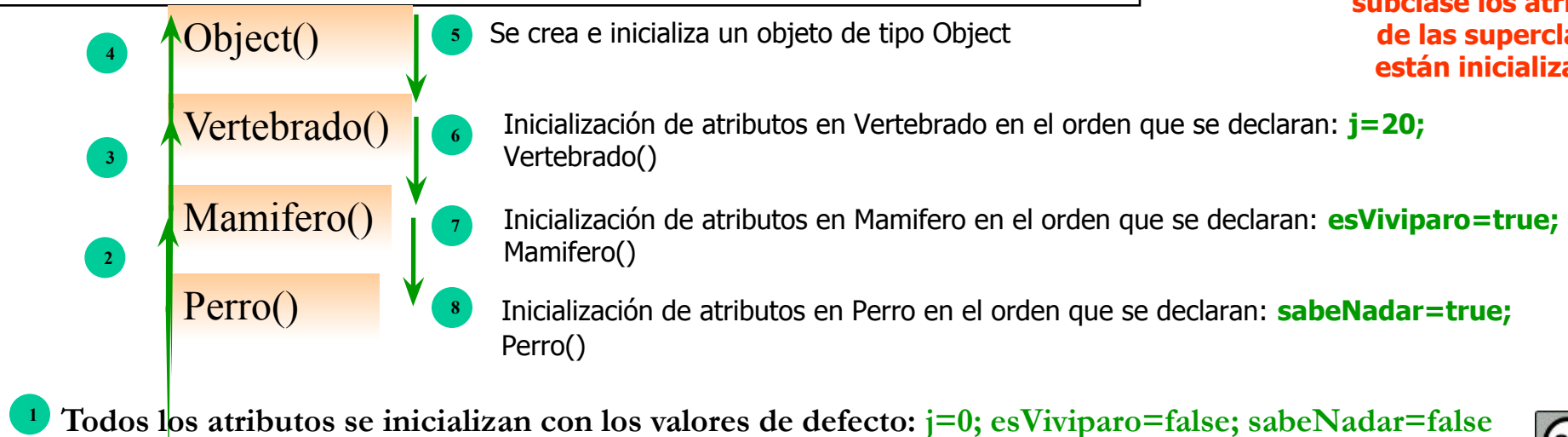
Los **constructores no son polimórficos** son implícitamente **static**.

```
public class Vertebrado {  
    private int j=20;  
  
    public Vertebrado() {  
        System.out.println("Vertebrado()");  
    }  
}
```

```
public class Mamifero extends Vertebrado {  
    private boolean esViviparo= true;  
    public Mamifero() {  
        System.out.println("Mamifero()");  
    }  
}
```

```
public class Perro extends Mamifero {  
    private boolean sabeNadar= true;  
  
    public Perro() {  
        System.out.println("Perro()");  
    }  
  
    public static void main(String[] args) {  
        Perro p=new Perro();  
    }  
}
```

Quando se ejecuta el constructor de una subclase los atributos de las superclases están inicializados



# Constructores y Polimorfismo

¿Qué ocurre si adentro de un constructor se invoca a un método sobrescrito?

```
public class Vertebrado {  
    private int j=20;  
  
    public Vertebrado() {  
        System.out.println("Vertebrado() antes de alimentar()");  
        alimentar();  
        System.out.println("Vertebrado() después de alimentar()");  
    }  
  
    public void alimentar() {  
        System.out.println("Vertebrado.alimentar()");  
    }  
}
```

```
public class Mamifero extends Vertebrado {  
    private boolean esViviparo= true;  
    public Mamifero() {  
        System.out.println("Mamifero()");  
    }  
    public void alimentar() {  
        System.out.println("Mamifero.alimentar(), vivíparo: "+ esViviparo);  
    }  
}
```

**j=20**

Vertebrado() antes de alimentar()

**Mamifero.alimentar(), esViviparo=false**

Vertebrado() después de alimentar()

```
public class Perro extends Mamifero {  
    private boolean sabeNadar= true;  
  
    public Perro() {  
        System.out.println("Perro()");  
    }  
  
    public static void main(String[] args) {  
        Perro p=new Perro();  
    }  
}
```

**esViviparo** aún NO se inicializó apropiadamente (de acuerdo a la declaración)

**j=0; esViviparo=false;**  
**sabeNadar=false**

# Constructores y Polimorfismo

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, sólo se sabe que los sub-objetos de las clases bases se inicializaron correctamente, pero no sabemos nada de las clases derivadas.
- Se estaría invocando a un método sobre un objeto que aún no se inicializó (su constructor NO se ejecutó), lo cual podría resultar en *bugs* difíciles de detectar, etc. Hay que tener en cuenta que el **binding dinámico** busca el método a invocar comenzando con la clase real del objeto y luego siguiendo por la cadena de herencia ascendentemente.

## Buena práctica para constructores:

- Los constructores **no deben invocar a métodos sobre-escribibles**.
- 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.

# Ocultar Atributos de la Superclase

```
public class Circulo {  
    public static final double PI= 3.14159;  
    public double r;  
    public Circulo(double r) { this.r = r; }  
    public static double radianesAgradados(double rads) {  
        return rads * 180 / PI;  
    }  
    public double area() {  
        return PI * r * r;  
    }  
    public double circunferencia() {  
        return 2 * PI * r;  
    }  
}
```

```
public class CirculoPlano extends Circulo {  
    public double r;  
    public double cx, cy;  
    public CirculoPlano(double r, double x, double y) {  
        super(r);  
        this.cx = x;  
        this.cy = y;  
        this.r = Math.sqrt(cx*cx + cy*cy);  
    }  
    public boolean pertenece(double x, double y) {  
        double dx = x - cx, dy = y - cy;  
        double distancia = Math.sqrt(dx*dx + dy*dy);  
        return (distancia < r);  
    }  
}
```

Se agrega a la clase **CirculoPlano** un atributo que guardará la distancia(**r**) entre el centro del círculo y el origen (0,0):

**r**

El atributo **r** de **CirculoPlano** **oculta** el atributo **r** de **Circulo**.  
Cuando en los métodos de **CirculoPlano**, nos referimos a:

**r** → es el atributo de **CirculoPlano**  
**this.r** → es el atributo de **CirculoPlano**  
**super.r** → es el atributo radio de **Circulo**: ((Circulo)this).r

¿A qué **r** hacen referencia los métodos **area()** y **circunferencia()** cuándo los invocamos sobre una instancia de **CirculoPlano**?

```
CirculoPlano cp=new CirculoPlano(5,10,10);  
cp.area();  
cp.circunferencia();
```

**Al radio definido en Circulo**



# Ocultar Atributos de la Superclase

```
public class Circulo {  
    public static final double PI= 3.14159;  
    public double r;  
    public Circulo(double r) { this.r = r; }  
    public static double radianesAgradados(double rads) {  
        return rads * 180 / PI;  
    }  
    public double area() {  
        return PI * r * r;  
    }  
    public double circunferencia() {  
        return 2 * PI * r;  
    }  
}
```

```
public class CirculoPlano extends Circulo {  
    public static final double PI = 3.14159265358979323846;  
    public double cx, cy;  
    public CirculoPlano(double r, double x, double y) {  
        super(r);  
        this.cx = x;  
        this.cy = y;  
    }  
    public boolean pertenece(double x, double y) {  
        double dx = x - cx, dy = y - cy;  
        double distancia = Math.sqrt(dx*dx + dy*dy);  
        return (distancia < r);  
    }  
}
```

¿Es posible **ocultar variables de Clase**? **SI !!**

Vamos a agregar una constante PI a **CirculoPlano**.

¿A qué PI hacen referencia area() y circunferencia() ?

**A la definida en Circulo, PI= 3.14159**

```
import static java.lang.System.out;  
public class TestOcultamiento {  
    public static void main(String args[]){  
        CirculoPlano cp=new CirculoPlano(10, 20, 10);  
        out.println("Area : " + cp.area());  
        out.println("Circunferencia: " + cp.circunferencia());  
    }  
}
```

**Area: 314.159**

**Circunferencia:**

**62.8318**



# Ocultar Métodos de Clase

Los métodos de clase de la misma manera que los atributos pueden **ocultarse** por una subclase, pero **NO sobrescribirse**. **NO son un reemplazo**.

```
class A {  
    int i = 1;  
    int f() { return i; }  
    static char g() { return 'A'; }  
}  
class B extends A {  
    int i = 2;  
    int f() { return -i; }  
    static char g() { return 'B'; }  
}
```

**Sobrescribe**      **Ocultar**

```
public class TestSobrescritura {  
    public static void main(String args[]) {  
        B b = new B();  
        System.out.println(b.i); // B.i; imprime 2  
        System.out.println(b.f()); // B.f(); imprime - 2  
        System.out.println(b.g()); // B.g(); imprime B  
        System.out.println(B.g()); // Imprime B  
                                   // Es la mejor manera de invocar a g()  
  
        A a = b; // Upcasting  
        System.out.println(a.i); // A.i; imprime 1 –OCULTAMIENTO-  
        System.out.println(a.f()); // B.f(); imprime -2 –SOBREESCRITURA-  
        System.out.println(a.g()); // A.g(); imprime A –OCULTAMIENTO-  
        System.out.println(A.g()); // Imprime A.  
                                   // Es la mejor manera de invocar a g()  
    }  
}
```

# Clases Anidadas

# Clases Anidadas

Las **clases** que vimos hasta ahora son clases de nivel superior: **son miembros directos de paquetes** y se definen en forma **independiente de otras clases**.

- Los **clases anidadas** son clases definidas **dentro de otras clases**.
- Las **clases anidadas** deben existir sólo para **servir a la clase que la anida**. Si son útiles en otros contextos, entonces deben definirse como clases de nivel superior.
- Las **clases anidadas NO definen una relación de composición** entre objetos.
- Las **clases anidadas** pueden declararse **private** o **protected** a diferencia de las de nivel superior que sólo pueden ser **public** o tener accesibilidad de **default** o **package**.
- Hay 4 tipos de **clases anidadas**: clases miembro no-estáticas, clases miembro estáticas, clases anónimas y clases locales.
- A las **clases miembro estáticas** también se las conoce como **clases internas**.

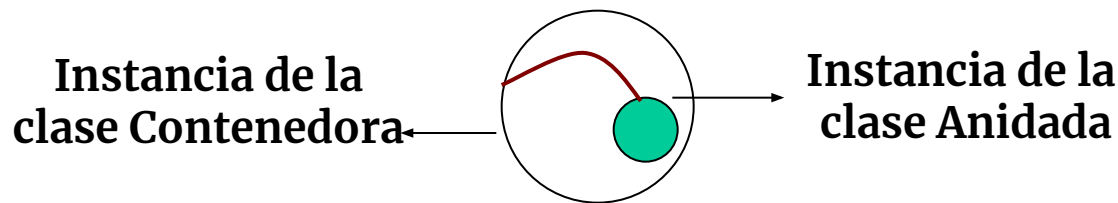
```
class Contenedora{  
    private int x=1;  
    static class Interna {  
        //TODO  
    }  
    class Anidada {  
        //TODO  
    }  
}
```

La **clase Interna** tiene acceso a todos los miembros declarados “static” de la clase que la anida, aún aquellos que son privados. **No está ligada a ninguna instancia particular**. Sólo existe para la clase que la contiene.

La **clase Anidada** tiene acceso a la implementación de la clase que la contiene (variables de instancia, de clase y métodos) como si fuesen propios. **Es una relación entre objetos**.

# Clases Anidadas

- Las **clases anidadas** del tipo **miembro no-estático** son similares a los métodos de instancia o a las variables de instancia. **Sus instancias se asocian a cada instancia de la clase que la contiene.**
- Un objeto de una **clase anidada** tiene **acceso ilimitado a la implementación** del objeto que lo anida, inclusive aquellos declarados **private**.
- Un objeto de una **clase anidada** tiene una **referencia implícita al objeto de la clase que lo instanció** (clase contenedora). A través de esta referencia tiene acceso al estado completo del objeto contenedor, inclusive a sus datos privados. Por lo tanto las **clases anidadas** tienen **más privilegios de acceso que las de nivel superior.**
- El compilador agrega la **referencia implícita en el constructor** de la **clase anidada**. Es invisible en la definición de la clase interna.



Una instancia de una clase anidada está siempre asociada con una instancia de la clase contenedora

- Las **clases internas son miembros estáticos**, no tienen acceso a esta referencia implícita, sólo pueden acceder a los miembros declarados estáticos (inclusive aquellos que son privados).
- Sintácticamente las clases miembro no-estáticas y estáticas son similares, difieren en que las estáticas tienen el modificador **static** en su declaración.

# Ejemplo de clase anidada

```
public class Paquete {  
    class Contenido {  
        private int i = 11;  
        public int valor() {  
            return i;  
        }  
    }  
    class Destino {  
        private String etiqueta;  
        Destino(String donde) {  
            etiqueta = donde;  
        }  
        String leerEtiqueta() {  
            return etiqueta;  
        }  
    }  
    public void vender(String dest) {  
        Contenido c = new Contenido();  
        Destino d = new Destino(dest);  
        System.out.println(d.leerEtiqueta());  
    }  
    public static void main(String[] args) {  
        Paquete p = new Paquete();  
        p.vender("Roma");  
    }  
}
```

## Clases Anidadas

En los métodos de instancia, las clases anidadas se usan de la misma manera que en las clases de nivel superior

Declaración de objetos de la clase anidada

```
public class Paquete {  
    class Contenido {  
        //código  
    }  
    class Destino {  
        //código  
    }  
    public Destino hacia(String s) {  
        return new Destino(s);  
    }  
    public Contenido cont() {  
        return new Contenido();  
    }  
    public void vender(String dest) {  
        Contenido c = cont();  
        Destino d = hacia(dest);  
        System.out.println(d.leerEtiqueta());  
    }  
} // Fin de la clase Paquete
```

La clase contenedora define métodos que devuelven referencias a objetos de la clase anidada

```
public class Viaje {  
    public static void main(String[] args) {  
        Paquete p = new Paquete();  
        p.vender("Roma");  
        Paquete q = new Paquete();  
        Paquete.Contenido c = q.cont();  
        Paquete.Destino d = q.hacia("Buenos Aires");  
    }  
}
```

# Clases anidadas & ocultamiento de implementación

¿Puedo ocultar una clase sin usar clases anidadas? **SI!!!** ¿Cómo?

Definiendo a la clase con acceso de **default** o **package**: la clase es visible solamente dentro del paquete donde se declaró.

¿Para qué usamos clases anidadas?

```
public interface Contenido {
    int valor();
}
public interface Destino {
    String leerEtiqueta();
}
```

Clase anidada privada: es accesible solamente desde la clase Paquete

```
package turismo;
public class Viaje {
    public static void main(String[] args) {
        Paquete p = new Paquete();
        Contenido c = p.cont();
        Destino d = p.hacia("Buenos Aires");
        System.out.println(d.leerEtiqueta());
        System.out.println(c.valor());
    }
}
```

Para proveer **ocultamiento de detalles de implementación**

Solamente a través **Upcasting/Generalización** a una clase base o interface pública, se obtiene una referencia

```
public class Paquete {
    private class PContenido implements Contenido{
        private int i=11;
        public int valor() {return i;}
    }
    private class PDestino implements Destino{
        private String etiqueta;
        private PDestino(String donde){
            etiqueta=donde;
        }
        public String leerEtiqueta() {
            return etiqueta;
        }
    }
    public Destino hacia(String s) {
        return new PDestino(s);
    }
    public Contenido cont() {
        return new PContenido();
    }
} // Fin de la clase Paquete
```

**Upcasting** al tipo de la interface

Las **clases anidadas privadas que implementan interfaces** son completamente invisibles e inaccesibles y de esta manera se **oculta la implementación**. Se evitan dependencias de tipos. Desde afuera de la clase **Paquete** se obtiene una referencia al tipo de la interface.

# Acceso a los Miembros de la Clase Contenedora

En la clase anidada **SIterador** se hace referencia a la variable **objetos** que es un atributo privado de la clase contenedora **Secuencia**.

```
public interface Iterador
{
    boolean fin();
    Object actual();
    void siguiente();
}
```

La interface **Iterador** se usa en la clase **Secuencia** para recorrer secuencias de objetos

```
public class TestSecuencia{
    public static void main(String[] args) {
        Secuencia s = new Secuencia(10);
        for (int i=0; i<10 ; i++)
            s.agregar(Integer.toString(i));
        Iterador it = s.getIterador();
        while (!it.fin()) {
            System.out.println(it.actual());
            it.siguiente();
        }
    }
}
```

Recorreremos una **Secuencia**

```
public class Secuencia{
    private Object[] objetos;
    private int sig = 0;
    public Secuencia(int tamaño){
        objetos = new Object[tamaño];
    }
    public void agregar(Object x){
        if (sig < objetos.length)
            objetos[sig++] = x;
    }
```

La clase **SIterador** se declaró privada: es inaccesible para los usuarios de la clase **Secuencia**.

```
private class SIterador implements Iterador{
    private int i = 0;
    public boolean fin(){ return (i== objetos.length); }
    public Object actual(){ return objetos[i]; }
    public void siguiente(){
        if (i< objetos.length) i++; }
}
```

```
    public Iterador getIterador(){return new SIterador(); }
} // Fin de la clase Secuencia
```

Se crea un objeto **Iterador** asociado a un objeto **Secuencia**

Las clases anidadas pueden acceder a métodos y atributos de la clase contenedora como si fuesen propios.

# Implementación de patrones con clases anidadas

Las clases anidadas se usan en el framework de colecciones para implementar:

- **iteradores** que acceden a los elementos de la colección secuencialmente sin exponer la representación interna. Usados en implementaciones de List y Set.
- **adaptadores** que permiten a las implementaciones de Map implementar sus vistas de colección, mediante algunos métodos de Map.

```
public class HashMap<K,V> extends AbstractMap<K,V>
implements Map<K,V>, Cloneable, Serializable {
```

```
    public Set<K> keySet() {
        Set<K> ks = keySet;
        return (ks != null ? ks : (keySet = new KeySet()));
    }
```

```
    private final class KeySet extends AbstractSet<K> {
        // código de la clase anidada
    }
```

**Adapter**

**keySet()** devuelve un **Set** y así un **HashMap** puede tratarse como un **Set**. El Set está soportado por el Map.

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

```
    public Iterator<E> iterator() {
        return new Itr();
    }
```

```
    private class Itr implements Iterator<E> {
        // código de la clase anidada
    }
```

**Iterator**

**Iterator** es una interface pública del framework de colecciones. **ArrayList** delega en el iterador el recorrido de la lista.



# Clases Locales

Las **clases locales** se definen dentro de un método o dentro de un bloque de código JAVA. Se declaran sin especificador de acceso dado que su alcance está restringido al bloque de código donde se definió. **NO pueden declararse *public*, *protected*, *private* ni *static***. Las interfaces y los tipos enumerativos NO pueden definirse localmente. Una **clase local** es similar a una variable local: es visible solamente dentro del bloque de código donde se definió

```
public class Paquete {
    public Destino hacia(String s) {
        class PDestino implements Destino {
            private String etiqueta;
            private PDestino(String donde){
                etiqueta=donde;
            }
            public String leerEtiqueta(){
                return etiqueta;
            }
        }

        return new PDestino(s);
    }
} // Fin de la clase Paquete
```

```
package turismo;
public class Viaje {
    public static void main(String[] args) {
        Paquete p = new Paquete();
        Destino d = p.hacia("Buenos Aires");
        System.out.println(d.leerEtiqueta());
    }
}
```

Las instancias de **clases locales** están asociadas con instancias de la clase que la contiene, por lo tanto pueden acceder a **TODOS** sus miembros incluyendo los privados.

-**PDestino es una clase interna local.**

-**Solamente adentro del método *hacia()* se pueden crear objetos PDestino.**

-**PDestino forma parte del método *hacia()* en vez de ser parte de la clase Paquete. La clase PDestino NO puede ser accedida afuera del método *hacia()*, excepto a través de una referencia a la interface Destino**

Lo único que sale del método ***hacia()*** es una referencia a **Destino**.  
Upcasting

-Una vez que el método ***hacia()*** terminó de ejecutarse, el objeto **PDestino** (*upcasteado* a Destino) es un objeto válido, es accesible.


-La clase **PDestino** a pesar de estar definida localmente en el método ***hacia()*** se compila con el resto de la clase (es un .class separado).

-La clase **PDestino** no está disponible afuera del método, está fuera de alcance (no se pueden crear objetos **PDestino** afuera del método ***hacia()***), es un nombre inválido.

# Clases Locales

Anidar una clase **en un alcance arbitrario**:

```
public class PaqueteCondicional {  
    private void tramoInterno(boolean b) {  
        if(b) {  
  
            class UnPaseo {  
                private String id;  
                UnPaseo(String s) {  
                    id = s;  
                }  
                String getPaseo() { return id; }  
            }  
  
            UnPaseo up = new UnPaseo("Villa Traful");  
            String s = up.getPaseo();  
        }  
    }  
    public void tramo() { tramoInterno(true); }  
}
```



**UnPaseo** es una **clase local** definida dentro del **bloque if**. No implica que la clase se cree condicionalmente: la clase se compila con el resto de la clase, sin embargo no está disponible fuera del alcance donde se definió.

La clase **UnPaseo** está fuera de alcance. No se reconoce el nombre.

```
package turismo;  
public class Viaje {  
    public static void main(String[] args) {  
        PaqueteCondicional p = new PaqueteCondicional();  
        p.tramo();  
    }  
}
```

# Clases Anónimas

Las **clases anónimas** son **clases locales** sin nombre. Se crean extendiendo una clase o implementando una interface. Combinan la sintaxis de definición de clases con la de instanciación de objetos. Las interfaces y los tipos enumerativos NO pueden definirse anónimamente.

El método **cont()** combina la creación del valor de retorno con la definición de la clase que representa el valor retornado.

- Crea un objeto de una clase anónima que implementa la interface **Contenido**.
- La referencia que devuelve el método **cont()** es automáticamente **upcasteada** a una referencia a **Contenido**.

```
public class Paquete {  
    public Contenido cont() {  
        return new Contenido(){  
            private int i=11;  
            public int valor(){  
                return i;  
            }  
        };  
    }  
} // Fin de la clase Paquete
```

Es una **abreviatura** de la declaración de una clase que implementa la interface **Contenido**

```
public interface Contenido {  
    int valor();  
}
```

```
package turismo;  
public class Viaje {  
    public static void main(String[] args) {  
        Paquete p = new Paquete();  
        Contenido c = p.cont();  
    }  
}
```

```
class PContenido implements Contenido {  
    private int i=11;  
    public int valor(){ return i;}  
}
```

-----  
return new PContenido();

# Clases Anónimas

Las clases anónimas son simultáneamente declaradas e instanciadas en el punto en que se van a usar.

```
public class Datos{  
    private int i;  
    public Datos(int x){i=x;}  
    public int valor(){return i;}  
}
```

```
public class Paquete {  
    public Datos info(int x) {  
        return new Datos(x){  
            public int valor(){  
                return super.valor()*50;  
            }  
        };  
    }  
} // Fin de la clase Paquete
```

```
package turismo;  
public class Viaje {  
    public static void main(String[] args) {  
        Paquete p = new Paquete();  
        Datos d = p.info(10);  
        System.out.println(d.valor());  
    }  
}
```

La clase base, **Datos**, requiere un constructor con un argumento.

El método **info()** crea un objeto de una clase anónima que es subclase de **Datos** usando el constructor con un argumento de la superclase.

Es una **abreviatura** de la declaración de una clase que extiende la clase **Datos**

```
class MisDatos extends Datos {  
    public MisDatos(int y){super(y);}  
    public int valor(){  
        return super.valor()*50;  
    }  
}  
  
return new MisDatos(x);
```

# Clases Anónimas

- Las **clases locales** y las **anónimas** tienen acceso a las **variables locales** del bloque de código donde están declaradas.

En este ejemplo, la clase anónima accede al parámetro String del método **hacia()**: inicializa la variable de instancia **etiqueta** con el valor del parámetro **donde**.

- Las **variables locales**, **parámetros de métodos** y **parámetros de manejadores de excepciones** que se usan en las clases locales y anónimas son **final**. ¿Por qué? El tiempo de vida de una instancia de una clase local o anónima es mayor que el tiempo de vida del método en el que se declaran y por ello es necesario preservar el estado de las variables locales a las que accede. Para asegurar esto se crean copias privadas de todas las variables locales que se usan (lo hace automáticamente el compilador) y se reemplazan todas las referencias a las variables locales por referencias a las copias. La única manera de garantizar que las variables locales y sus copias contengan el mismo valor es obligando a las variables locales a ser **final**.

- El compilador se encarga de agregar un parámetro extra al constructor de la clase anónima (del tipo de la variable local) y una variable de instancia, en la que se mantendrá la copia.

```
public interface Destino
{
    String leerEtiqueta();
}
```

```
public class Paquete {
    public Destino hacia(String donde) {
        return new Destino(){
            private String etiqueta=donde;
            public String leerEtiqueta(){
                return etiqueta;
            }
        };
    }
} // Fin de la clase Paquete
```

```
package turismo;
public class Viaje {
    public static void main(String[] args) {
        Paquete p = new Paquete();
        Destino d = p.hacia("Buenos Aires");
        System.out.println(d.leerEtiqueta());
    }
}
```

# Clases Anónimas

¿Puede definirse un constructor en una clase anónima?

**NO!!!**


No es posible pues la clase no tiene nombre y el constructor debe tener el mismo nombre que la clase.

¿Cómo podemos realizar una inicialización al estilo de un constructor?

## Bloques de inicialización (de instancia)

```
public class Paquete {  
    public Destino hacia(String donde, float precio) {  
        return new Destino() {  
            private int costo;  
            private String etiqueta=donde;  
            {costo=Math.round(precio);  
                if (costo>100)  
                    System.out.println("Muy caro!!!");  
            }  
            public String leerEtiqueta(){return etiqueta;}  
        };  
    }  
} // Fin de la clase Paquete
```

**Bloque de Inicialización**



- El bloque de inicialización funciona como un constructor para la clase anónima. Se ejecuta cada vez que se crea una instancia.
- El uso del bloque de inicialización para definir constructores es limitado dado que NO es posible definir constructores sobrecargados.

Las clases anónimas son limitadas dado que sólo pueden extender una clase o implementar una interface, no pueden hacer ambas cosas a la vez ni tampoco pueden implementar más de una interface.

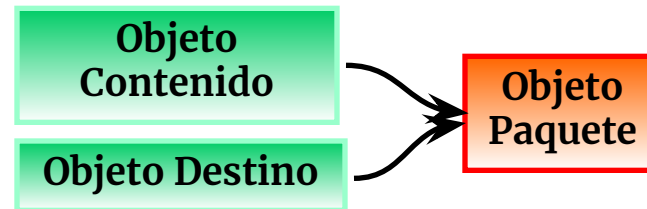
# ¿Cómo pueden las clases anidadas acceder a los miembros de la clase contenedora?

Cada **objeto de la clase anidada** mantiene una **referencia al objeto de la clase contenedora** que lo creó. De esta manera cuando nos referimos a un miembro de la clase contenedora (atributo o método) esta referencia “oculta” es usada.

El compilador se encarga de todos los detalles: agrega en el constructor de la clase anidada una referencia al objeto de la clase contenedora.

**SIEMPRE** un objeto de una clase anidada está asociado con un objeto de la clase contenedora: la construcción de un objeto de la clase anidada requiere de la referencia al objeto de la clase contenedora.

```
public class Paquete {
    class Contenido {
        private int i = 11;
        public int valor() {return i;}
    }
    class Destino {
        private String etiqueta;
        Destino(String donde) {etiqueta = donde;}
        String leerEtiqueta() {return etiqueta;}
    }
}
```



```
public class Viaje {
    public static void main(String[] args) {
        Paquete p = new Paquete();
        Paquete.Contenido c = p.new Contenido();
        Paquete.Destino d = p.new Destino("Buenos Aires");
    }
}
```

Para crear un objeto de la clase anidada es necesario usar un objeto de la clase contenedora (asociación manual)

# ¿Cómo nombrar al objeto de la clase Contenedora?

```
private class SIterador implements Iterador{  
    private int i = 0;  
  
    public boolean fin(){return ( this.i == Secuencia.this.objetos.length );}  
  
    public Object actual(){return Secuencia.this.objetos[i];}  
  
    public void siguiente(){ if ( this.i < Secuencia.this.objetos.length) this.i++ ; }  
}
```

Es necesario usar esta sintaxis sólo si el nombre del atributo de la clase contenedora está **ocultado** por un atributo con el mismo nombre en la clase anidada

La sintaxis para nombrar al objeto de la clase contenedora es: **NombreDeLaClaseContenedora.this**

## ¿Cómo crear instancias de una clase anidada?

```
public Iterador getIterador(){return new SIterador();}
```

Al invocar al constructor de la clase anidada, automáticamente la instancia de la clase anidada se asocia con el objeto **this** de la clase contenedora.

```
public Iterador getIterador(){return this.new SIterador();}
```

Es posible **explicitar la instancia contenedora** cuando se crea el objeto de la clase anidada.

```
Paquete p=new Paquete();  
Paquete.Destino d=p.new Destino("Roma");  
Paquete.Contenido c=p.new Contenido();
```

Dependiendo de la visibilidad de las clases anidadas, es posible crear instancias fuera de la clase contenedora. En el ejemplo, es posible crear instancias de las clases Destino y Contenido en clases ubicadas en el paquete por *default*.

La sintaxis **.new** produce el alcance correcto, no es necesario calificar el nombre de la clase contenedora en la invocación al constructor



# Identificadores de Clases Anidadas

Las clases JAVA de alto nivel generan archivos **.class** en donde se guarda toda la información necesaria para crear objetos. Esta información produce una “meta-clase” llamada **objeto Class**.

Las **clases internas** también producen archivos **.class** que contienen la información de sus **objetos Class**. Los nombres de estos archivos cumplen la siguiente regla:

```
interface Contador {  
    int siguiente();  
}
```



Contador.class

ClaseAltoNivel.class

ClaseAltoNivel\$1ContadorLocal.class

ClaseAltoNivel\$2.class



```
public class ClaseAltoNivel{  
    Contador getContador(final String nom){  
        class ContadorLocal{  
            public ContadorLocal(){...}  
            public int siguiente(){....}  
        }  
        return new ContadorLocal();  
    }  
    Contador getContador2(final String nom){  
        return new Contador(){  
            public int siguiente(){....}  
        };  
    }  
}
```

# Objetos Función

Java no provee funciones o referencias a funciones.

```
class StringLengthComparator {  
    public int compare(String s1, String s2) {  
        return s1.length() - s2.length();  
    }  
}
```

Un objeto  
`StringLengthComparator`  
puede expresar  
**Objetos-Función**

Una instancia de una clase JAVA que exporta métodos que realizan operaciones sobre otros objetos pasados como parámetros, es un **objeto-función**.

Un objeto **StringLengthComparator** exporta un único método que toma 2 strings y devuelve un número negativo si el primer string es más corto que el segundo, cero si ambos strings tienen la misma longitud y un número positivo si el primer string es más largo que el segundo.

El método **compare()** permite ordenar strings de acuerdo a su longitud.

Un objeto **StringLengthComparator** es un “objeto función” o un “puntero” a un comparador, pudiendo ser invocado con un par de strings arbitrarios.

```
Lista.ordenar(stringArray[], new StringLengthComparator())
```

# Objetos Función

Los objetos **StringLengthComparator** representan una **estrategia concreta** para comparar strings por longitud.

La clase **StringLengthComparator** **no tiene estado**: no tiene variables de instancia, todas las instancias son funcionalmente equivalentes. La **estrategia de comparación** debe definirse como un singleton.

```
class StringLengthComparator {  
    private StringLengthComparator() { }  
    public static final StringLengthComparator  
        INSTANCE = new StringLengthComparator();  
    public int compare(String s1, String s2) {  
        return s1.length() - s2.length();  
    }  
}
```

**Estrategia Concreta**

```
Lista.ordenar(args, StringLengthComparator.INSTANCE);
```

Para pasar la instancia de **StringLengthComparator** a un método se necesita contar con un **tipo** apropiado como parámetro. No es recomendable usar el tipo **StringLengthComparator** porque no permite intercambiar estrategias de comparación. **¿Cómo intercambio estrategias?**

Definimos una **interface genérica** para la estrategia: **Estrategia Abstracta**

```
public interface Comparator<T> {  
    public int compare(T t1, T t2);  
}
```

# Objetos Función

```
class StringLengthComparator implements Comparator<String> {  
    private StringLengthComparator() { }  
    public static final Comparator<String>  
        INSTANCE = new StringLengthComparator();  
    public int compare(String s1, String s2) {  
        return s1.length() - s2.length();  
    }  
}
```

**Estrategia Concreta**

Definimos las **estrategias abstractas** como interfaces y las **concretas** como implementaciones de dichas interfaces

## ¿Cómo usamos la estrategia de comparación de strings?

```
public class Lista {  
    public static void ordenar(String[] stringArray, Comparator<String> comparador)  
    {  
        for (int i = 0; i < stringArray.length-1; i++) {  
  
            for (int j = i+1; j < stringArray.length; j++)  
                if (comparador.compare(stringArray[i], stringArray[j]) > 0) {  
                    String aux = stringArray[i];  
                    stringArray[i] = stringArray[j];  
                    stringArray[j] = aux;  
                }  
        }  
    }  
}
```

# Objetos Función

## ¿Cómo usamos la estrategia de comparación de strings?

Objeto Comparator

```
Listas.ordenar(listaDeStrings, StringLengthComparator.INSTANCE);
```

Las clases que representan **estrategias concretas** frecuentemente son definidas anónimas:

```
Listas.ordenar(listaDeStrings, new Comparator<String> () {  
    public int compare(String s1, String s2) {  
        return s1.length() - s2.length();  
    }  
});
```

En este último caso se crea una nueva instancia cada vez que se invoca al `ordenar()`

# Objetos Función

Redefinimos el ejemplo usando la interface **java.util.Comparator** que es genérica, entonces es aplicable a cualquier tipo de comparadores:

```
package java.util;  
public interface Comparator<T> {  
    int compare(T o1, T o2);  
    boolean equals(Object obj);  
}
```

**Estrategia Abstracta**

```
package anidadas;  
import java.util.Comparator;  
class StringLengthComparator implements Comparator<String>{  
    private StringLengthComparator() { }  
    public static final Comparator<String>  
        INSTANCE = new StringLengthComparator();  
    public int compare(String s1, String s2) {  
        return s1.length() - s2.length();  
    }  
}
```

**Estrategia Concreta**

Método `sort()` de la clase `Arrays`:

```
public static <T> void sort(T[] a, Comparator<? super T> c)  
Arrays.sort(stringArray, StringLengthComparator.INSTANCE);
```

# Objetos Función

```
package java.util;  
public interface Comparator<T> {  
    int compare(T o1, T o2);  
    boolean equals(Object obj);  
}
```

**La estrategia concreta  
definida como una  
Clase Anónima**

```
Arrays.sort(stringArray, new java.util.Comparator<String>() {  
    public int compare(String s1, String s2) {  
        return s1.length() - s2.length();  
    }  
});
```

# Objetos Función

```
package anidadas;
import java.util.Arrays;
public class TestAnonimas {
    public static void main(String[] args) {
        String[] stringArray= {"hola", "chau", "hi", "goodbye"};
        Arrays.sort(stringArray, new java.util.Comparator<String>() {
            public int compare(String s1, String s2) {
                return s1.length() - s2.length();
            }
        });
        for (String s: stringArray)
            System.out.println(s);
    }
}
```

¿Cuál es la salida?

hi  
hola  
chau  
goodbye



# Objetos Función

Usar una clase anónima en algunas circunstancias creará un objeto nuevo cada vez, por ejemplo si se ejecuta repetitivamente. Una solución más eficiente consiste en guardar la referencia al objeto función en una constante de clase y reusarla cada vez que se necesita.

La **interface** que representa la estrategia sirve como **tipo para todas las instancias de estrategias concretas**, por ello las clases que implementan estrategias concretas no necesitan ser públicas y, esto permite intercambiarlas. La **clase Host** exporta constantes de clase del tipo de la interface de la estrategia y las **clases que implementan las estrategias** pueden ser **clases anidadas privadas** de dicha clase.

```
package anidadas;
```

## Patrón Strategy

```
public class Host {  
    private static class StrLenCmp implements java.util.Comparator<String> {  
        public int compare(String s1, String s2) {  
            return s1.length() - s2.length();  
        }  
    }  
  
    public static final java.util.Comparator<String>  
        STRING_LENGTH_COMPARATOR = new StrLenCmp();  
}
```

# Objetos Función

```
package anidadas;
import java.util.Arrays;

public class TestAnidadas2 {
    public static void main(String[] args) {
        String[] stringArray= {"hola", "chau", "hi", "goodbye"};

        Arrays.sort(stringArray, Host.STRING_LENGTH_COMPARATOR) ;
        for (String s: stringArray)
            System.out.println(s);
    }
}
```

**Síntesis**: los objetos-función permiten implementar el **patrón Strategy**. En JAVA este patrón se implementa declarando una **interface que representa la estrategia** y diferentes **clases que implementan dicha interface**, las **estrategias concretas**. Si la **estrategia concreta se usa sólo una vez**, entonces se declara e instancia como una **clase anónima**. Si una **estrategia concreta se usa repetitivamente** es conveniente definirla como una **clase interna privada y exportar la estrategia mediante una constante pública de clase** del tipo de la interface

De esta manera es posible **intercambiar** en ejecución las estrategias.

# Java 8, lambdas y clases anónimas

A partir de JAVA 8 se formalizó la noción que las interfaces con un único método son especiales y merecen un tratamiento especial.

Estas interfaces se conocen como **interfaces funcionales** y JAVA permite crear implementaciones de estas interfaces usando **expresiones lambda o lambdas**.

Las lambdas son similares a las clases anónimas, en cuanto a su función, pero son mucho más concisas.

```
Collections.sort(palabras,  
    (s1, s2) -> Integer.compare(s1.length(), s2.length()));
```

# Tipos Enumerativos

# Tipos Enumerativos

JAVA soporta 2 categorías de tipos de datos de propósitos específico:

Una categoría de clases: **Tipos Enumerativos**

Una categoría de interfaces: **Anotaciones**

# Tipos Enumerativos

Los **tipos enumerativos** se incorporaron a la plataforma JAVA partir de JAVA 5.0. Constituyen una **categoría especial de clases**.

Un tipo enumerativo es un tipo de datos que tiene asociado un **conjunto de valores finito y acotado**.

La palabra clave **enum** se usa para definir un nuevo tipo enumerativo:

**package labo;**

El cuerpo del tipo enum es una lista separada por comas de los valores posibles

**public enum Estados {CONECTANDO, LEYENDO, LISTO, ERROR ;}**

Los **valores** son **constantes públicas de clase (public static final)** y se hace referencia a ellas de la siguiente manera: **Estados.CONECTANDO**, **Estados.LEYENDO**. A una variable de tipo **Estados** se le puede asignar uno de los 4 valores definidos o **null**. Los valores de un tipo enumerado se llaman **valores enumerados** y también **constantes enum**.

**El tipo enumerativo es una clase y sus valores son instancias de dicha clase.**

Garantiza **seguridad de tipos**. Es una diferencia fundamental con usar constantes de tipo primitivo. El compilador puede chequear si a un método se le pasa un objeto de tipo **Estado**.

Por **convención**, los valores de los tipos enumerativos se escriben en **mayúsculas** como cualquier otra constante de clase.

# Características de los Tipos Enum

Cuando se crea un tipo enumerado el compilador crea una clase que es subclase de `java.lang.Enum`. No es posible extender la clase `Enum` para crear un tipo enumerativo propio. La única manera de crear un tipo enumerativo es usando la palabra clave `enum`.

`public abstract class Enum<E> extends Enum<E>> extends Object implements Comparable<E>, Serializable`

Constructors		
Modifier	Constructor	Description
protected	<code>Enum(String name, int ordinal)</code>	Sole constructor.

Los tipos enumerativos no tienen constructores públicos. Las únicas instancias son las declaradas por el tipo `enum`.

All Methods	Static Methods	Instance Methods	Concrete Methods
Modifier and Type		Method	
protected final Object		<code>clone()</code>	
final int		<code>compareTo(E o)</code>	
final Optional<Enum.EnumDesc<E>>		<code>describeConstable()</code>	
final boolean		<code>equals(Object other)</code>	
protected final void		<code>finalize()</code>	
final Class<E>		<code>getDeclaringClass()</code>	
final int		<code>hashCode()</code>	
final String		<code>name()</code>	
final int		<code>ordinal()</code>	
String		<code>toString()</code>	
static <T extends Enum<T>>		<code>valueOf(Class&lt;T&gt; enumClass, String name)</code>	

- Los tipos enumerativos implementan la interface `java.lang.Comparable` y `java.io.Serializable`.
- El método `compareTo()` establece un orden entre los valores enumerados de acuerdo al orden en que aparecen en la declaración del `enum`. Es **final**.
- Es seguro comparar valores enumerativos usando el operador `==` en lugar de el método `equals()` dado que el conjunto de valores posible es limitado. El método `equals()` internamente usa el operador `==` y además es **final**.
- El método `name()` devuelve un `String` con el nombre de la constante `enum`. Es **final**.
- El método `ordinal()` devuelve un entero con la posición del `enum` según está declarado. Es **final**.
- El método `toString()` puede sobreescribirse. Por defecto retorna el nombre de la instancia del enumerativo.

<sup>T</sup> No es posible extender un tipo enumerativo, son implícitamente **final**. El compilador define **final** a la clase que lo soporta.

# Usos de Tipos Enumerativos

## Tipos Enum y sentencia Switch

```
Estados misEstados=Estados.LEYENDO;
```

```
switch(misEstados) {  
  case CONECTANDO: {  
    System.out.println(misEstados);  
    break;  
  }  
  case LEYENDO: {  
    System.out.println(misEstados);  
    break;  
  }  
  case LISTO: {  
    System.out.println(misEstados);  
    break;  
  }  
  case ERROR:  
    throw new IOException("error");  
}
```

La sentencia **switch** soporta **tipos enumerativos**.

Si el tipo de la declaración de la expresión **switch** es un tipo enumerativo, las etiquetas de los **case** deben ser todas **instancias sin calificación** de dicho tipo.

Es ilegal usar **null** como valor de una etiqueta **case**.

Si NO se incluyen todos los valores posibles del tipo enumerativo en las etiquetas de los **case** o la etiqueta **default**, el compilador emite una advertencia.



# Usos de Tipos Enumerativos

## EnumMap y EnumSet

La clase `java.util.EnumMap` es una implementación especializada de un **Map** que requiere como clave un tipo **Enumerativo** y la clase `java.util.EnumSet` es una implementación de **Set** adaptada a valores de tipo **Enumerativo**. Ambas estructuras de datos están optimizadas para tipos Enumerativos.

```
package labo;
import java.util.EnumMap;
public class TestEnumHash {
    public static void main(String[] args) {
```

```
        EnumMap<Estados,String> mensajes = new EnumMap<>(Estados.class);
        mensajes.put(Estados.CONECTANDO, "Conectando...");
        mensajes.put(Estados.LEYENDO, "Leyendo...");
        mensajes.put(Estados.LISTO, "Listo!!...");
        mensajes.put(Estados.ERROR, "Falla en la descarga....");
```

```
        Estados miEstado= getEstado();
        String unMensaje = mensajes.get(miEstado);
        System.out.print(unMensaje);
```

```
    }
    public static Estados getEstado(){
        return Estados. CONECTANDO;
    }
}
```

Crea un Map vacío cuyas claves son del tipo enum Estados



# Tipos Enumerativos Enriquecidos

Los tipos enumerativos pueden incluir métodos y propiedades.

```
package labo;  
public enum Prefijo {  
    MM("m", .001),  
    CM("c", .01),  
    DM("d", .1),  
    DAM("D", 10.0),  
    HM("h", 100.0),  
    KM("k", 1000.0);
```

Las instancias se declaran al principio

El constructor y los métodos se declaran igual que en las clases

Cada constante se declara con valores para la **abreviatura** y para el factor **multiplicador**

Cuando se declaran **propiedades** y **métodos**, la lista de constantes enumerativas termina en ;

```
    private String abrev;  
    private double multiplicador;
```

**Propiedades** de los Prefijos: abreviatura y factor multiplicador

```
    Prefijo(String abrev, double multiplicador) {  
        this.abrev = abrev;  
        this.multiplicador = multiplicador;  
    }
```

Se debe proveer de un **constructor**.

Los **valores declarados para las propiedades** se pasan al constructor cuando se crean las constantes.

El **constructor** de un tipo enumerativo tiene acceso privado o privado del paquete. Automáticamente crea las instancias y **NO puede ser invocado**.

```
    public String abrev() { return abrev; }  
    public double multiplicador() { return multiplicador; }
```

Métodos que permiten **recuperar la abreviatura** y el **factor multiplicador** de cada Prefijo



# Tipos Enumerativos Enriquecidos

```
package labo;
```

```
public class TestPrefijo {
```

```
public static void main(String[] args) {
```

```
    double longTablaM= Double.parseDouble(args[0]);
```

```
    for (Prefijo p : Prefijo.values() )
```

```
        System.out.println("La longitud de la tabla en "+ p+ " "+longTablaM*p.multiplicador());
```

```
    }
```

```
}
```

**values()** es un método de Clase que inserta el compilador y que permite **recuperar en un arreglo todos los valores del enumerativo** en el orden en que fueron declarados.

```
java TestPrefijo 15
```

La longitud de la tabla en MM 0.015

La longitud de la tabla en CM 0.15

La longitud de la tabla en DM 1.5

La longitud de la tabla en DAM 150.0

La longitud de la tabla en HM 1500.0

La longitud de la tabla en KM 15000.0

# Tipos Enumerativos Enriquecidos

**Sobreescritura** del método **toString()** de una enumeración:

```
package labo;
```

```
public enum Señales {  
    VERDE, ROJO, AMARILLO;
```

```
    public String toString() {  
        String id = name();    Recupera el nombre de la instancia  
        String minuscula = id.substring(1).toLowerCase();  
        return id.charAt(0) + minuscula;  
    }  
}
```

Verde  
Rojo  
Amarillo

```
package labo;  
import static java.lang.System.out;
```

```
public class PruebaSeñales {  
    public static void main (String args[]){  
        for (Señales s: Señales.values())  
            out.println(s);  
    }  
}
```

# Tipos Enumerativos Enriquecidos

Es posible asociar **comportamiento diferente** con cada constante enumerativa.

```
package enumerativos;
public enum Operation {
    PLUS, MINUS, TIMES, DIVIDE;
    public double apply(double x, double y){
        switch (this){
            case PLUS: return x+y;
            case MINUS: return x-y;
            case TIMES: return x*y;
            case DIVIDE: return x/y;
        }
        throw new AssertionError("Operación desconocida: " + this);
    }
}
```

Este código funciona bien, pero...

- **NO compilará** si no le ponemos la sentencia **throw** porque el final del método es técnicamente alcanzable a pesar que nunca se alcanzará (están los **cases** de todos los enumerativos).
- El **código es frágil**: si se agrega una nueva constante enumerativa y nos olvidamos de agregar el **case** correspondiente en el switch, el enumerativo compilará pero dará un error en ejecución cuando intenta aplicar la nueva operación.

# Tipos Enumerativos Enriquecidos

Es posible asociar **comportamiento diferente** con cada constante enumerativa.

```
package enumerativos;
```

```
public enum Operation {
```

```
    PLUS("+") {double apply(double x, double y) {return x + y;} },
```

```
    MINUS("-") {double apply(double x, double y) {return x - y;} },
```

```
    TIMES("*") {double apply(double x, double y) {return x * y;} },
```

```
    DIVIDE("/") {double apply(double x, double y) {return x / y;} };
```

```
    private final String symbol;
```

```
    Operation(String symbol) {
```

```
        this.symbol = symbol;
```

```
    }
```

```
    public String toString() {
```

```
        return symbol;
```

```
    }
```

```
    abstract double apply(double x, double y);
```

Esta es una mejor solución para **asociar comportamiento específico con cada constante enumerativa**:

- Declarar un **método abstracto** en el tipo enumerativo
- **Sobreescribirlo** con un **método concreto** en cada constante.

Son **implementaciones de métodos de constantes específicas**.

**El código es más robusto:** si nos olvidamos de agregar el método apply() cuando agregamos una nueva constante, el compilador lo recordará, **los métodos abstractos en un tipo enum deben sobreescribirse con cada constante.**

# Tipos Enumerativos Enriquecidos

Combinar implementaciones de métodos y las constantes enum (apply() y toString())

```
package enumerativos;

public class TestEnum{
    public static void main(String[] args) {
        double x = Double.parseDouble(args[0]);
        double y = Double.parseDouble(args[1]);
        for (Operation op : Operation.values())
            System.out.printf("%f %s %f = %f%n",x, op, y, op.apply(x, y));
    }
}
```

¿Cuál es la salida de ejecutar TestEnum 12 5 ?

**12,000000 + 5,000000 = 17,000000**

**12,000000 - 5,000000 = 7,000000**

**12,000000 \* 5,000000 = 60,000000**

**12,000000 / 5,000000 = 2,400000**

# Tipos Enumerativos Enriquecidos

Los tipos Enumerativos **NO pueden extenderse** pero si pueden **implementar interfaces**.

```
package labo;
```

```
public interface Abreivable {  
    String abrev();  
}
```

Se define la interface  
**Abreivable** para cualquier  
objeto que pueda abreviarse

Cualquier método que acepte como  
parámetro un objeto **Abreivable**  
aceptará una instancia de  
**UnidadesTemperatura**.

```
package labo;
```

```
public enum UnidadesTemperatura implements Abreivable {
```

```
    GRADOCELSIUS("°C"),  
    GRADOFARENHEIT("°F"),  
    GRADOREAUMUR("°R"),  
    GRADONEWTON("°N");  
    private String abrev;  
    UnidadesTemperatura(String abrev){  
        this.abrev=abrev;  
    }  
    public String abrev() {  
        return abrev;  
    }  
}
```

```
public static void imprimirAbreviatura(Abreivable s){  
    out.println(s.abrev());  
}  
  
public static void main(String args[]){  
    for (UnidadesTemperatura u :  
        UnidadesTemperatura.values() )  
        imprimirAbreviatura(u);  
    }  
}
```



# Genéricos en JAVA

# Introducción

- En JAVA se denomina "Genéricos" a la capacidad del lenguaje de **definir** y **usar tipos** (clases e interfaces) y **métodos genéricos**.
- Los **tipos** y **métodos genéricos** difieren de los "regulares" en que contienen **tipos de datos** como **parámetros formales**.
- Los **tipos** y **los métodos genéricos** se incorporan en JSE 5.0 para proveer **chequeo de tipos en compilación**.
- No pueden declararse genéricos los tipos enumerativos, las clases anónimas y subclases de excepciones.

```
public class LinkedList <E> extends AbstractSequentialList <E> implements List<E>, Queue<E>, Cloneable, Serializable
```

**LinkedList es un tipo Genérico**

**E es un parámetro formal que denota un tipo de dato**

**Los elementos que se almacenan en la lista encadenada son del tipo desconocido E**

Con tipos genéricos podemos definir: `LinkedList<String>` y `LinkedList<Integer>`

- Una **clase genérica** tiene el mismo comportamiento para todos sus posibles tipos de parámetros.
- Los **tipos parametrizados** se forman al asignarle **tipos reales a los parámetros formales** que denotan un tipo de datos:

`LinkedList<E>`, `Comparator<T>`      **Tipos Genéricos**

`LinkedList<String> listaStr;`      **Lista de Strings**

`LinkedList<Integer> listaInt = new LinkedList<Integer>();`      **Lista de Enteros**

`Comparator<String> compara;`      **Comparador de Strings**

**Tipos  
Parametrizados**

- El **framework de colecciones** del paquete **java.util** es **genérico** a partir de Java 5.0

# ¿Qué problemas resuelven los “Genéricos”?

Se **evitan los errores en ejecución** causados por el uso de *casting*.

```
List list = new ArrayList(); // Lista que contiene Strings
```

```
list.add("abc");  
list.add(new Integer(5));
```

**El compilador devuelve una advertencia “unchecked call”**

```
for(Object obj : list){
```

```
String str=(String) obj;
```

```
}
```

**El casting dispara el siguiente error en ejecución: “ClassCastException”**

Los **Genéricos** proveen una **mejora para el sistema de tipos**:

- permite operar sobre objetos de múltiples tipos.
- provee seguridad en compilación pudiendo detectar *errores* en compilación.

Los programas que usan genéricos son seguros, reusables, es un código limpio.

La **inserción errónea genera un mensaje de error en compilación** que indica exactamente qué es lo que está mal.

```
List<String> list1 = new ArrayList<>();
```

```
list1.add("abc");
```

```
list1.add(new Integer(5)); // error de compilación  
//“error: incompatible types: Integer cannot be converted”
```

```
for(String str : list1){  
    System.out.print(str);
```

```
}
```



**No es necesario hacer casting**

# Tipos Genéricos y Tipos Parametrizados

Un **tipo genérico** es un **tipo de datos con parámetros formales que denotan tipos de datos**.

Un **tipo parametrizado** es una **instanciación de un tipo genérico con argumentos que son tipos reales**.

```
public interface Collection<E> extends Iterable<E> {  
    int size();  
    boolean isEmpty();  
    boolean contains(Object o);  
    Iterator<E> iterator();  
    Object[] toArray();  
    <T> T[] toArray(T[] a);  
    boolean add(E e);  
    boolean remove(Object o);  
    boolean containsAll(Collection<?> c);  
    boolean addAll(Collection<? extends E> c);  
    boolean removeAll(Collection<?> c);  
    boolean retainAll(Collection<?> c);  
    void clear();  
    boolean equals(Object o);  
    int hashCode();  
}
```

## INTERFACE GENÉRICA

Un **tipo genérico** es un tipo de datos que **tiene uno o más parámetros formales** que representan tipos de datos.

Cuando el tipo genérico es instanciado o declarado, se reemplazan los parámetros formales por argumentos que representan tipos reales.

La **interface Collection** tiene un parámetro formal **E** que indica un tipo de datos. En la declaración de una colección específica, **E** es reemplazado por un tipo real.

La **instanciación de un tipo genérico** se denomina **tipo parametrizado**.

```
Collection <String> col=new LinkedList<String>();  
List <String> list= new ArrayList<>(); // a partir de JAVA 7  
Collection <? extends Number> col=new LinkedList<Integer>();
```

## TIPO PARAMETRIZADO

# Declaración de tipos Genéricos

En la definición de los **tipos genéricos** la sección correspondiente a los parámetros continúa al nombre de la clase o interface. Es una lista separada por comas y delimitada por los símbolos <>.

```
package genericos.definicion;  
public class ParOrdenado <X,Y> {  
    private X a;  
    private Y b;  
    public ParOrdenado (X a, Y b){  
        this.a=a;  
        this.b=b;  
    }  
    public X getA() { return a; }  
    public void setA(X a) { this.a = a; }  
    public Y getB() { return b; }  
    public void setB(Y b) { this.b = b; }  
}
```

El alcance de los identificadores **X** e **Y** es toda la clase **ParOrdenado**.

En el ejemplo, **X** e **Y** son usados en la declaración de variables de instancia y como argumentos y tipos de retorno de los métodos de instancia.

Los parámetros formales (tipos de datos) pueden declararse con cotas. Las cotas proveen acceso a métodos del tipo definido en el parámetro formal (que es desconocido) .

En el ejemplo de la clase **ParOrdenado** no invocamos a ningún método sobre los tipos desconocidos X e Y, es por esta razón que los 2 tipos son sin cotas.

# Tipos Parametrizados Concretos

Para usar un **tipo genérico** se deben especificar los **argumentos** que reemplazarán a los **parámetros formales** que denotan tipos de datos.

Los argumentos pueden ser referencias a **tipos concretos** como String, Long, Date, etc.

```
package genericos.definicion;
import java.awt.Color;
public class TestParOrdenado {
public static void main(String[] args){
```

```
    ParOrdenado<String, Long> par = new ParOrdenado<>("hola", 23L);
```

```
    System.out.println("(" + par.getA() + ", " + par.getB() + ")");
```

```
    ParOrdenado<String, Color> nombreColor = new ParOrdenado<String, Color>("Rojo", Color.RED);
```

```
    System.out.println("(" + nombreColor.getA() + ", " + nombreColor.getB() + ")");
```


```
    ParOrdenado<Double, Double> coordenadas = new ParOrdenado<Double, Double>(17.3, 42.8);
```

```
    System.out.println("(" + coordenadas.getA() + ", " + coordenadas.getB() + ")"); (hola, 23)
```

```
}
```

```
}
```

Autoboxing: el 23 es  
automáticamente convertido  
a Long



(Rojo, java.awt.Color[r=255,g=0,b=0])

(17.3, 42.8)

La **instanciación** de ParOrdenado<String, Long>, ParOrdenado<String, Color>, ParOrdenado<Double, Double> son **tipos parametrizados concretos** y se usan como un tipo regular. Podemos usar **tipos parametrizados** como **argumentos de métodos**, para **declarar variables** y en la expresión **new** para crear un objeto.

# Tipos Parametrizados con Comodines

Son **instanciaciones de tipos genéricos** que contienen al menos un comodín (?) como tipo de argumento en **oposición a los tipos concretos**.

Un comodín es una construcción sintáctica “?” que denota la familia de “todos los tipos”.

```
static int cantElemEnComun(Set<?> s1, Set<?> s2) {  
    int result = 0;  
    for (Object o1 : s1)  
        if (s2.contains(o1)) result++;  
    return result;  
}
```

**Tipos parametrizados comodines sin cota**

El ? indica un conjunto de “algún tipo desconocido”.

Es el **conjunto parametrizado más general**, capaz de contener cualquier tipo de elemento. No interesa cuál es el parámetro del tipo real.

Puede ser Set<String>, Set<A>, Set<Long>, etc.

**Set<?>** representa la flia de todos los conjuntos de “cualquier tipo”. No nos interesa el tipo real, es un conjunto que puede contener cualquier tipo.

Se pueden usar para **declarar variables o parámetros de métodos**, como s1 y s2, pero no en una sentencia **new** para crear objetos.

Como no sabemos nada del tipo de los elementos del Set, **solo se puede leer y tratar los objetos leídos como instancias de Object**.

**Los tipos parametrizados con comodines sin cota son útiles en situaciones en las que no es necesario conocer nada sobre el tipo del argumento.**

¿Cuál es la diferencia entre Set<?> y Set?

# Tipos Parametrizados con Comodines

Un comodín con una cota superior “? extends T” es la familia de todos los tipos que son **subtipos** de T. T es la **cota superior**.

Un comodín con una cota inferior “? super T” es la familia de todos los tipos que son **supertipos** de T. T es la **cota inferior**.

Los **tipos parametrizados con comodines acotados** son útiles en situaciones en las que es necesario contar con un **conocimiento parcial sobre el tipo de argumento** de los tipos parametrizados.

```
List<? extends Number> l;
```

La familia de todos los tipos de listas cuyos elementos son subtipos de **Number**.

```
Comparable<? super String> s;
```

La familia de todas las instancias de la interface **Comparable** para tipos que son supertipos de **String**.

```
public final class Byte extends Number implements Comparable<Byte>
public final class Double extends Number implements Comparable<Double>
public final class Float extends Number implements Comparable<Float>
public final class Integer extends Number implements Comparable<Integer>
public final class Long extends Number implements Comparable<Long>
public final class Short extends Number implements Comparable<Short>
```

**Subtipos de Number**

```
String, Object, CharSequence, Serializable y Comparable<String>
```

**Supertipos de String**



# Tipos Parametrizados

## Ejemplos

Suma de los números de una lista de números:

```
public static double sum(List<Number> list){  
    double sum = 0;  
    for(Number n : list){  
        sum += n.doubleValue();  
    }  
    return sum;  
}
```

Los **tipos parametrizados** son **invariantes** por lo tanto List<Double> y List<Integer> no están relacionados con List<Number> y el método sum() no se puede usar.

Los **tipos parametrizados con comodines acotados** ofrecen mayor **flexibilidad** que los tipos invariantes.

```
public static double sum(List<? extends Number> list){  
    double sum = 0;  
    for(Number n : list){  
        sum += n.doubleValue();  
    }  
    return sum;  
}
```

```
public static void main(String[] args) {  
    List<Integer> ints = new ArrayList<>();  
    ints.add(3); ints.add(5); ints.add(10);  
    double sum = sum(ints);  
    System.out.println("Suma de ints="+sum);  
}
```

# Tipos Parametrizados con Comodines

## Resumen

Un tipo parametrizado con comodín no es un tipo concreto.

Pueden declararse variables del tipo parametrizado con comodín, pero no pueden crearse objetos con el operador **new**. En ese sentido son similares a las interfaces.

Las variables de tipo parametrizado con comodín hacen referencia a un objeto perteneciente a la familia de tipos que el tipo parametrizado comodín denota.

```
List<?> col= new ArrayList<String>();  
List<? extends Number> lista=new ArrayList<Long>();  
ParOrdenado<String, ?> par=new ParOrdenado<String, String>();
```

```
List<? extends Number> l = new ArrayList<String>(); ERROR!!!!
```

String no es subtipo de Number y consecuentemente **ArrayList<String>** no pertenece a la familia de tipos denotados por **List<? extends Number>**.

**List<?>** denota una lista de elementos de algún tipo, desconocido. Es una lista de “sólo lectura”.

# Reglas de subtipos para genéricos

Los **tipos parametrizados** forman una jerarquía de tipos basada en el tipo base NO en el tipo de los argumentos. Los **tipos parametrizados son invariantes**.

```
public interface List<E> extends Collection<E> {}  
public class ArrayList<E> extends AbstractList<E> implements List<E> {}
```

```
List<Integer> listita = new ArrayList<>();
```

```
List<Integer> li = listita;
```

```
Collection<Integer> c = listita;
```

```
ArrayList<Number> n = listita;
```

```
List<Object> o = listita;
```

```
List li = listita;
```

¿Cuáles asignaciones cumplen las reglas de subtipo?

Un `ArrayList<Integer>` es un `List<Integer>`, un `Collection<Integer>` y un `List`, pero NO es un `ArrayList<Number>` ni un `List<Object>`.

```
List<Integer> li = new ArrayList<>();
```

```
li.add(123);
```

```
List<Number> lo = li;
```

```
Number nro = lo.get(0);
```

```
lo.add(3.14);
```

```
Integer i = li.get(1);
```

¿Es válido este código?

# Reglas de subtipos para genéricos

Los **tipos parametrizados** forman una jerarquía de tipos basada en el tipo base NO en el tipo de los argumentos. Los **tipos parametrizados son invariantes**.

```
public interface List<E> extends Collection<E> {}
```

```
public class ArrayList<E> extends AbstractList<E> implements List<E> {}
```

```
List<Integer> listita = new ArrayList<>();
```

```
List<Integer> li = listita;
```

```
Collection<Integer> c = listita;
```

```
ArrayList<Number> n = listita;
```

```
List<Object> o = listita;
```

```
List li = listita;
```

Un `ArrayList<Integer>` es un `List<Integer>`, un `Collection<Integer>` y un `List`, pero NO es un `ArrayList<Number>` ni un `List<Object>`.

`List<Integer>` no es un subtipo de `List<Number>`

```
List<Integer> li = new ArrayList<>();
```

```
li.add(123);
```

```
List<Number> lo = li;
```

```
Number nro = lo.get(0);
```

```
lo.add(3.14);
```

```
Integer i = li.get(1);
```

No compila `List<Number> lo=li;` NO ES POSIBLE CONVERTIR DE `List<Integer>` a `List<Number>`

Si asumimos que compila:

- podríamos recuperar elementos de la lista como `Number` en vez de como `Integer`:

```
Number nro = lo.get(0);
```

- podríamos agregar un objeto `Double`: `lo.add(3.14);`

- la línea `li.get(1);` daría error de *casting*, porque no puedo *castear* un `Double` a un `Integer`

# Tipos Parametrizados con Comodines Acotados - Ejemplos

```
interface Collection<E> {  
    public boolean addAll(Collection<? extends E> c);  
}
```

`public interface List<E> extends Collection<E> {}`  
**addAll()** agrega todos los elementos de una colección en otra colección

"**? extends E**": está permitido agregar a una colección de tipos E elementos de otra colección que son subtipo de E.

El parámetro **c** de **addAll()** produce elementos para la colección, por lo tanto los **elementos de c** deben ser subtipos de E.

```
package genericos;  
import java.util.ArrayList;  
import java.util.Arrays;  
import java.util.List;  
public class TestComodinConExtends {  
    public static void main(String[] args) {  
        List<Number> nums = new ArrayList<>();  
        List<Integer> ints = Arrays.asList(1, 2);  
        List<Double> dbls = Arrays.asList(2.78, 3.14);  
        nums.addAll(ints);  
        nums.addAll(dbls);  
    }  
}
```

- **nums** es de tipo **List<Number>** que es subtipo de **Collection<Number>**
- **ints** es de tipo **List<Integer>** que es subtipo de **Collection<? extends Number>**
- **dbls** es de tipo **List<Double>** que es subtipo de **Collection<? extends Number>**.
- **E** toma el valor **Number**.

Si el parámetro de **addAll()** estuviese escrito sin comodines es decir **Collection<E>**: **¿Podría agregarse a *nums* una lista de enteros y de números decimales?**

**NO!!** Solamente estaría permitido agregar listas que estuviesen explícitamente declaradas como listas de **Number**.

# ¿Qué está disponible a través de variables de tipo parametrizado con comodín?

```
public static void printList(PrintStream out, List<?> lista) {  
    // lista.add("hola");  
    for(int i=0, n=lista.size(); i < n; i++) {  
        if (i > 0) out.println(", ");  
        Object o = lista.get(i);  
        out.print(o.toString());  
    }  
}
```

El método **printList()** imprime todos los elementos de la lista pasada como parámetro.

```
interface List<E> {  
    boolean add (E element);  
    E get(int index)  
}
```

- El método **add(E e)** de List<E> acepta un argumento del tipo especificado por el parámetro **E**. En nuestro ejemplo el tipo es “?” (desconocido) por lo tanto el compilador no puede asegurar que el objeto pasado como parámetro al método add() es del tipo esperado por el método. **Por lo tanto, List<?> es de sólo lectura, no es posible invocar a los métodos add( ), set() y addAll( ) de List. Código Seguro.**
- El método **E get(int)** de List<E> devuelve un valor que es del mismo tipo que el parámetro **E**. En nuestro ejemplo el tipo es “?” (desconocido) por lo tanto el método get() puede ser invocado y el resultado puede asignarse a un variable de tipo Object (sabemos que será un objeto).
- **Conclusión:** los tipos parametrizados con comodines son de solo lectura.
- Si en lugar de List<?> usamos List, ¿ocurre lo mismo?

# ¿Qué está disponible a través de variables de tipo parametrizado comodín?

```
List<Integer> li = new ArrayList<>();  
li.add(123);  
List<? extends Number> lo = li;  
Number nro = lo.get(0);  
lo.add(3.14);  
Integer i = li.get(1);
```

→ ¿Se puede hacer?

- Se puede asignar `li` a `lo` porque `li` es de tipo `List<Integer>` y es subtipo de `List<? extends Number>`.
- No se puede agregar un `Double` a `lo` (`List<? extends Number>`) dado que podría ser una lista de algún otro subtipo de `Number`.

En general: si una variable referencia a una colección que declara contener elementos de tipo `<? extends E>` es posible recuperar elementos de la estructura pero **NO** es posible agregar elementos a la estructura. **Es una variable de solo lectura.**

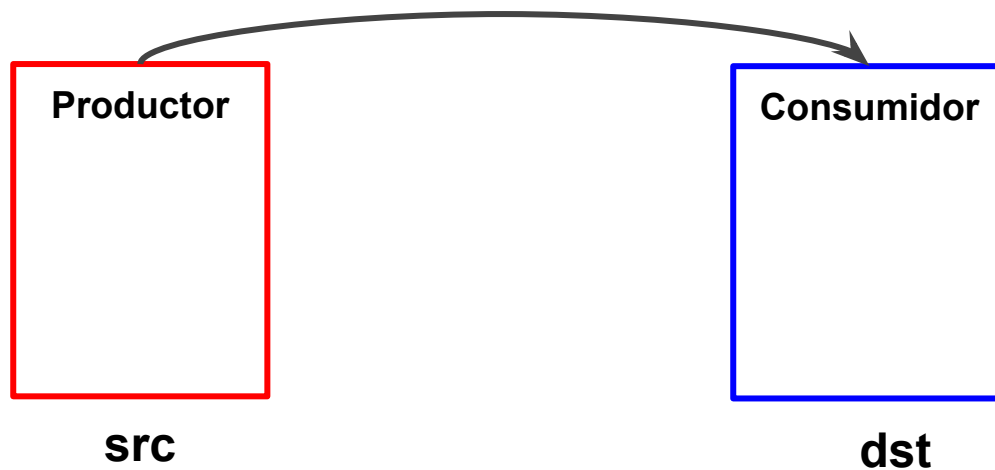
# Tipos Parametrizados con Comodines

## Acotados - Ejemplos

```
public class Collections{  
    public static <T> void copy( List<? super T> dst , List<? extends T> src) {  
        for (int i = 0; i < src.size(); i++) { dst.set(i, src.get(i)); }  
    }  
}
```

El método **copy()** de la clase **Collections** copia elementos desde una **lista fuente** a una **destino**

La **lista destino** tiene que ser capaz de **guardar los elementos de la lista fuente**.



Paradigma productor-consumidor:

productor-extends, consumidor-super

- src **produce** para dst y dst **consume** de src.
- Los elementos de src deben ser subtipos de los de dst.
- Los elementos de dst deben ser supertipos de los de src.

**List<? extends T>** (solo lectura)

**List<? super T>**

**copy:** es un método genérico, acepta argumentos de tipo List<? super T> y List<? extends T>, devuelve void y se aplica a cualquier tipo T.

**List<? super T> dst:** la lista destino puede contener elementos de cualquier tipo que sea supertipo de T.

**List<? extends T> src:** la lista fuente puede contener elementos de cualquier tipo que sea subtipo de T.



# Tipos Parametrizados con Comodines Acotados - Ejemplos

```
public class Collections{  
    public static <T> void copy( List<? super T> dst , List<? extends T> src) {  
        for (int i = 0; i < src.size(); i++) { dst.set(i, src.get(i)); }  
    }  
}
```

Object



T



Integer

```
package genericos;  
import java.util.*;  
public class TestMetodosGenericos {  
    public static void main(String[] args) {  
        List<Object> objs = Arrays.asList(2, 3.14, "hola");  
        List<Integer> ints = Arrays.asList(5, 6);  
        Collections.copy(objs, ints);  
    }  
    Collections.<Number>copy(objs, ints);  
}
```

Integer es subtipo de T y Object es supertipo de T. El compilador elige dentro de los posibles, por ejemplo **Number**

```
List<String> strs = Arrays.asList("2", "hola", "3.14" );  
List<Integer> ints2 = Arrays.asList(5, 6);  
Collections.copy(strs, ints2); NO ES APLICABLE
```

[5, 6, hola]

Productor  
(src)

ints

Consumidor  
(dst)

objs

```
public static <E> Set<E> union(Set<? extends E> s1, Set<? extends E> s2)
```

S1 y s2 son ambos productores, sus elementos de guardarán en el conjunto receptor del método union()

# Métodos Genéricos

De la misma manera que las clases e interfaces los métodos pueden ser **genéricos**, es decir pueden ser parametrizados por uno o más tipos de datos.

```
public static <T> int countOccurrences(T[] list, T itemToCount) {  
    int count = 0;  
    if (itemToCount == null) {  
        for (T listItem : list)  
            if (listItem == null)  
                count++;  
    } else {  
        for (T listItem : list)  
            if (itemToCount.equals(listItem))  
                count++;  
    }  
    return count;  
}
```

**countOccurrences()** cuenta la cantidad de ocurrencias del elemento **itemToCount** en el arreglo genérico **list** y se aplica a **cualquier tipo T**

Declarar métodos genéricos es similar a declarar tipos genéricos pero el alcance del tipo como parámetro está limitado al método.

Los métodos genéricos expresan dependencias entre los tipos de los argumentos y/o el tipo de retorno del método.

```
class Collections {  
    public static <T> void copy(List<T> dest, List<? extends T> src) {}  
}
```

Cuando se usa un método genérico no hay una mención explícita al tipo que sustituirá al tipo del parámetro formal. El compilador infiere el tipo a partir de los parámetros reales. **Inferencia de tipos.**

**En nuestro caso como list es un arreglo de Number, entonces T es un Number.**

**countOccurrences (arrNumber,10);**

```
Number arrNumber[]=new Number[5];  
arrNumber[0]=10.9; arrNumber[1]=5L;  
arrNumber[2]=15; arrNumber[3]=10;  
arrNumber[4]=10;
```

# Métodos Genéricos

```
public static <T extends Comparable <? superT>> T max(Collection<? extends T> coll)
```

**max** devuelve el mayor valor de una colección de elementos de un tipo desconocido T, de acuerdo al orden natural de sus elementos.

Aquí el parámetro **coll produce** números que cumplen relaciones de orden.

Los **Comparables** y **Comparator** siempre son **consumidores**.

Los métodos genéricos se invocan de la forma usual, los argumentos que representan a los tipos concretos no necesitan ser explicitados, son **inferidos automáticamente**.

```
import java.util.*;
public class TestMetodos {
    public static void main(String[] args) {
        List<String> strList=new ArrayList<>();
        strList.add("hola");
        strList.add("chau");
        strList.add("hi");
        strList.add("bye");
        System.out.println(Collections.max(strList));
    }
}
```

<T extends Comparable <? superT>> T

Cualquier tipo T que puedan compararse, es decir que son objetos mutuamente comparables.

**El compilador invoca automáticamente al método max() usando como argumento el tipo String.**

**El compilador infiere automáticamente el tipo del argumento.**

# Tipos Parametrizados con Comodines

## Algunos tips

Si se usan correctamente los **tipos comodines** son **invisibles** para los usuarios de una clase.

Los **métodos aceptan los parámetros** que deben aceptar y **rechazan** los que deberían rechazar.

Si el usuario de una clase tiene que pensar en tipos de comodines, probablemente haya algo mal con la API.

# Ejemplos

```
public class TestListaParametrizada {  
    public static void main(String[] args) {  
        List<String> listaPalabras = new ArrayList<>();  
        //listaPalabras.add(args); // ERROR DE COMPILACIÓN!!!  
        for(String arg : args)  
            listaPalabras.add(arg);  
        String unaPalabra = listaPalabras.get(0); // CASTING AUTOMÁTICO!!!  
    }  
}
```

SIN TIPOS GENÉRICOS un error accidental en la inserción causaría una falla de EJECUCIÓN

Map es un tipo genérico del framework de colecciones con 2 parámetros que representan tipos de datos: uno representa el tipo de las claves y el otro el tipo del valor de cada clave.

**public interface Map<K,V> {}**

```
public class TestMapParametrizado{  
    public static void main(String[] args) {  
        Map<String,Integer> tabla = new HashMap<>();  
        for(int i=0; i < args.length; i++)  
            tabla.put(args[i], i);  
        int posicion =tabla.get("hola"); // CASTING AUTOMÁTICO!!!  
    }  
}
```

Operaciones de boxing y unboxing permiten convertir automáticamente de tipos primitivos a clases wrapper

java TestMapParametrizado chau hola adiós

# Ventajas de usar Genéricos

## Detección temprana de errores

El compilador puede realizar más chequeos de tipos. Los errores son detectados tempranamente y reportados por el compilador en forma de mensajes de error en lugar de ser detectados en ejecución mediante excepciones.

```
package genericos;
import java.util.Date;
import java.util.LinkedList;
public class TestErrores {
    public static void main(String[] args) {
        List<String> list= new LinkedList<String>();
        list.add("hola");
        list.add(new Date());
    }
}
```

Es una lista homogénea de strings

El compilador chequea que la lista sólo contenga strings. En otro caso, el compilador lo rechaza

Con **tipos no-parametrizados** (LinkedList) es posible agregar diferentes tipos de elementos a la colección. La compilación será exitosa.

```
List list= new LinkedList ();
list.add("hola");
list.add(new Date());
```

# Ventajas de usar Genéricos

## Detección temprana de errores (Continuación)

```
package genericos;
import java.util.*;
public class TestErrores {
    public static void main(String[] args) {
        List<String> list= new LinkedList<String>();
        list.add("hola");
        String str=list.get(0);
    }
}
```

Los elementos se recuperan sin realizar *casting*. Está garantizado que la lista contiene strings.

Con **tipos no-parametrizados** (LinkedList) no hay conocimiento ni garantías respecto del tipo de los elementos que se recuperan. Todos los métodos retornan referencias a **Object** que deben ser *downcasteadas* al tipo real del elemento a recuperar.

```
List list= new LinkedList();
list.add("hola");
list.add(new Date());
String str=(String)list.get(0);
```

El *casting* podría causar errores en ejecución, **ClassCastException**, en caso que el elemento recuperado no sea un string

# Ventajas de usar Genéricos

## Seguridad de Tipos

En JAVA se considera que un programa es **seguro respecto al tipado** si compila sin errores ni advertencias y en ejecución NO dispara ningún **ClassCastException**.

Un programa bien formado permite que el compilador realice suficientes **chequeos de tipos basados en información estática** y que no ocurran errores inesperados de tipos en ejecución. **ClassCastException** sería un error inesperado de tipos que se produce en ejecución sin ninguna expresión de casting visible en el código fuente.



# Interoperabilidad con código *legacy*

## ¿Cómo se implementan los tipos genéricos?

Mediante la **técnica llamada "borrado" o *erasure***: el compilador usa la información de los tipos genéricos y tipos parametrizados para compilar y luego elimina la información del TIPO.

**List<Integer>, List<String> y List<String>** son traducidas a **List**. El bytecode es el mismo que el de **List**.

**Después de la traducción por "borrado" desaparece toda la información del TIPO.**

- Simplicidad: hay una única implementación de List, no una versión de cada tipo.
- Compatibilidad: la misma librería puede ser accedida tanto por genéricos como por no-genéricos.
- Evolución Fácil: no es necesario cambiar todo el código a genéricos, es posible evolucionar el código a genéricos actualizando de a un paquete, clase o método.

Podremos mantener versiones del código fuente de nuestras librerías que usen tipos genéricos y otras que usen raw types es decir sin tipos genéricos. Es posible invocar a un método diseñado con tipos parametrizados con tipos sin parametrizar y viceversa.

**Compatibilidad de Migración.**

# Compatibilidad entre tipos parametrizados y tipos en bruto (*raw types*)

Es compatible la asignación entre un tipo en bruto y todas las instanciaciones de un tipo genérico.

La asignación de un tipo parametrizado a un tipo en bruto está totalmente permitida; la asignación de un tipo en bruto a uno parametrizado produce advertencias en compilación **“conversiones no chequeadas/seguridad de tipos”**.

Es importante tener en cuenta que se rompe la seguridad de tipos genéricos.

```
List<Integer> li = new ArrayList<Integer>(); // tipo parametrizado
```

```
List lo=new ArrayList (); // raw type
```

```
li.add(123);
```

```
lo=li;
```

```
Object nro = lo.get(0);
```

```
lo.add("hola"); // inserción de un tipo no permitido no se detecta en compilación
```

```
Integer i = li.get(1);
```

La advertencia indica que el compilador desconoce si el ArrayList que se está asignando contiene o no Integer

Este código compila, pero en ejecución dispara un error de casting: **ClassCastException**