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, espacios de nombres y módulos

- 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.
- A partir de Java 9, 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**.

Al ser único el nombre del dominio de Internet, 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 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;
```

Importación de tipos por demanda
(los tipos son importados a medida que se necesitan)

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

Es posible usar el nombre simple de las clases por ej. 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;
public class ConflictoDeNombres {
   //TODO
}
```

¿Se puede hacer?

```
package pruebas;
import java.util.*;
import java.awt.*;
public class ConflictoDeNombres {
    //TODO
}
```

¿Se puede hacer?

```
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");
  }
}
```

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**.

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

```
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 diferente firma que los definidos en la clase **Arrays**.

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;
import java.awt.*;
    public class Rectangulo extends Graphics implementes Draggable {
}
```

ar/edu/unlp/linti/graficos/Rectangulo.class

Cuando el "intérprete" JAVA ejecuta un programa y necesita localizar dinámicamente un archivo **.class,** por ej. cuando se crea un objeto o se accede a 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 "\" 0 "/" (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 entono **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 **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
```

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 en JAVA** son:

public	protected	package (no tiene palabra clave)	private

Más libre — Más restrictivo

El **control de acceso** permite **ocultar la implementación**. Separa la **interface públca** de la **implementación**, favoreciendo 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 qué 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");
}
}
```

```
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;
}
}
```

¿Qué observan en el código?

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 puden usarse en la subclase, decimos NO SE HEREDAN.