

ALGORITMOS Y ESTRUCTURA DE DATOS III



Java: una breve introducción

Cristian Cappelletti (ccappelletti@pol.una.py)

Año 2018 – 1er. semestre

Contenido

- **Parte I** : Introducción al lenguaje.
- **Parte II** : Tipo de datos, operadores, expresiones.
- **Parte III** : Entrada/Salida. Estructuras de control.
- **Parte IV** : Arreglos y cadenas.
- **Parte V** : Introducción a la POO. Clases
- **Parte VI** : Herencia
- **Parte VII** : Manejo de excepciones
- **Parte VIII** : Clases abstractas, interfaces y generic.
- **Parte IX** : Construcciones avanzadas en Java 8
- **Bibliografía**

Parte I

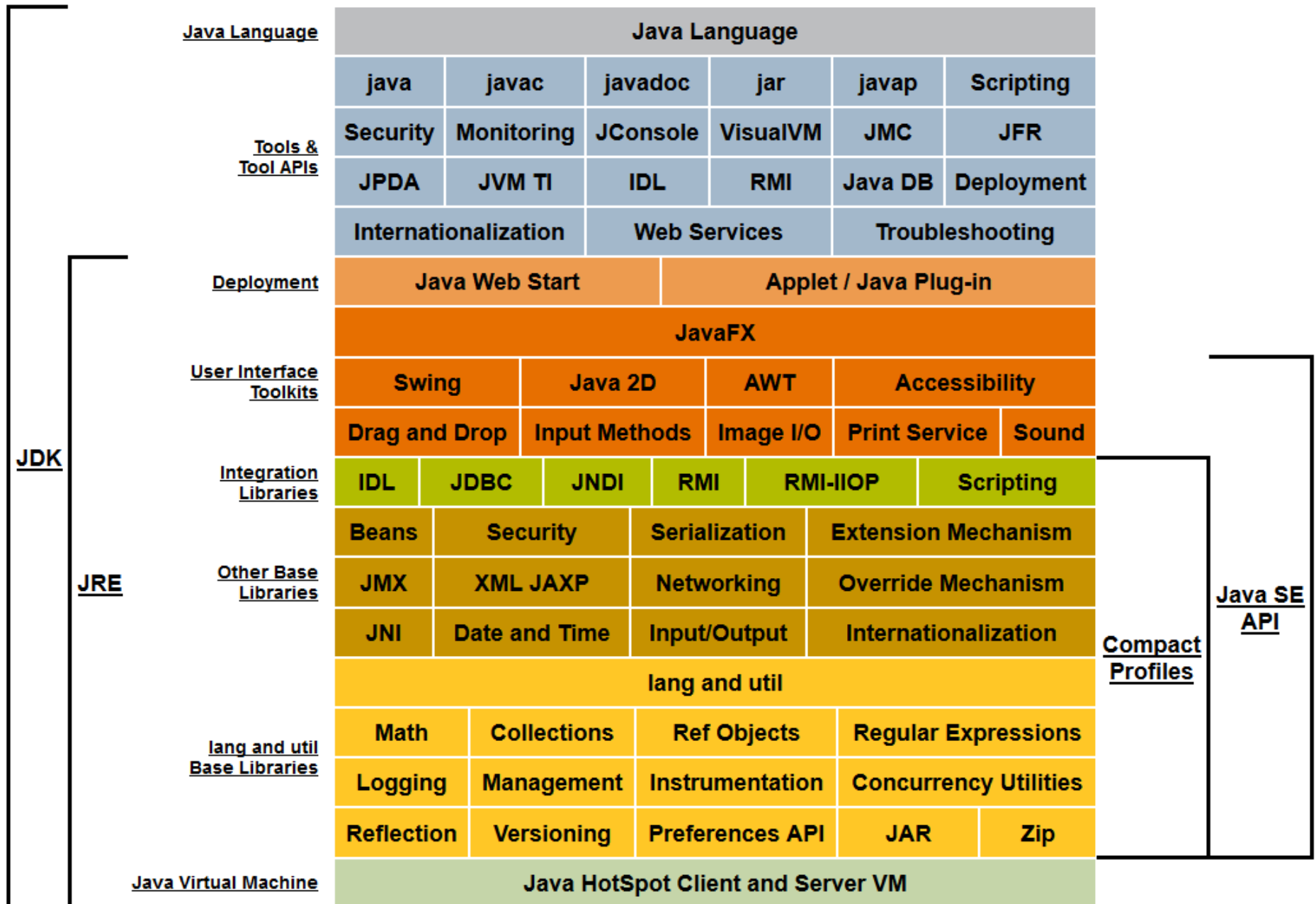
Introducción al lenguaje

Al Contenido

- Motivaciones de Java
- Tecnología Java
- Historia
- Ejemplo

Tecnología Java – un vistazo

JSE JDK 8 (<http://www.oracle.com/technetwork/java/iavase/tech/index.htm>)



Java : Historia

- Java

- Ideado en 1991 y oficializado en 1995 por Sun por James Gosling et al. de Sun Microsystems.
(pasó unos cuatro años antes Sun aceptara liderar el proyecto propuesto por un grupo liderado por Gosling)
- Inicialmente llamado *Oak*, en honor a un “árbol” que Gosling veía desde de su ventana en Sun, este nombre fue cambiado luego a Java debido a que ya existía un lenguaje con el nombre Oak.
- En 2006 Java fue liberado bajo GNU GPL
- En 2009 SUN fue adquirido por Oracle y así también Java

Java : Historia

- Motivación original para Java
 - La necesidad de un lenguaje de plataforma independiente que debía ser “incrustado” en productos electrónicos de consumo como tostadoras, TVs, etc.
- Primeros pasos
 - El primer proyecto fue un entorno para una especie de PDA denominado Star7
 - En ese momento, WWW e Internet estaban adquiriendo gran popularidad. Se propuso entonces que Java podría ser usado para la programación orientada a Internet.

Lema de los creadores de Java

Write once, run anywhere

Java: motivaciones de su diseño

Extraído del paper de James Gosling/Henry McGilton

The Java Language Environment – Mayo 1996

- Simple, orientado a objetos y familiar
- Robusto y seguro
- De arquitectura neutral y portable
- De buen rendimiento
- Interpretado, multihilo y dinámico

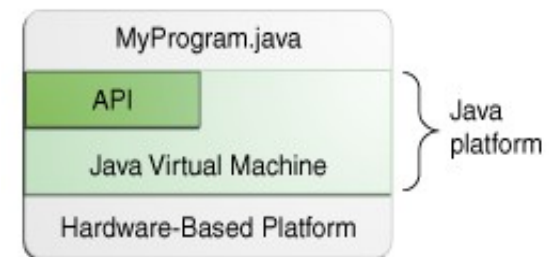
Evolución de las versiones de Java

- Java 1.0 (enero/1996) 8 paquetes, 212 clases
- Java 1.1 (marzo/1997) 23 paquetes, 504 clases
- Java 1.2 (dic/1998) 59 paq., 1520 clases
- Java 1.3 (abr/2000) 77 paq., 1595 clases
- Java 1.4 (2002) 103 paq., 2175 clases
- Java 1.5 (oct/2004) 131 paq., 2656 clases (*major PL changes*)
- Java 6 (2007) update 45 (Java SE 6)
- Java 7 (2011) update 79(abr/15) (Java SE 7)
- Java 8 (2015) update 121(feb/16) (Java SE 8) (enhancement in 8: *lambda expressions* <http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html>)
- Java 9 (2017) actual 9.04 (ene/2018) (main enhancement: *modules*)

Timeline (hasta 2014): <http://oracle.com.edgesuite.net/timeline/java/>

Por qué decimos tecnología JAVA?

- La tecnología JAVA incluye:
 - Un lenguaje de programación
 - Un entorno de desarrollo
 - Un entorno de ejecución
 - Un entorno de distribución
- La plataforma Java incluye:
 - *JVM* (Java Runtime Environment)
 - *Java API* (Java Application Programming Interface)



Tecnología Java:

Lenguaje de programación

Java es un lenguaje de programación de propósito general que puede ser usado para crear cualquier tipo de aplicación, como cualquier otro lenguaje de programación convencional.

Tecnología Java:

Entorno de desarrollo

La tecnología Java incluye una serie de herramientas para crear aplicaciones completas:

- un compilador (javac)
- un intérprete (java)
- un generador de documentación (javadoc)
- un conjunto completo de librerías
- entre otras cosas...

Tecnología Java:

Entorno de ejecución

Las aplicaciones Java son programas que se ejecutan en cualquier máquina donde el entorno de ejecución Java (*JRE – Java Runtime Environment*) se encuentre instalado.

Tecnología Java

Entorno de distribución

Existen dos principales entornos de distribución:

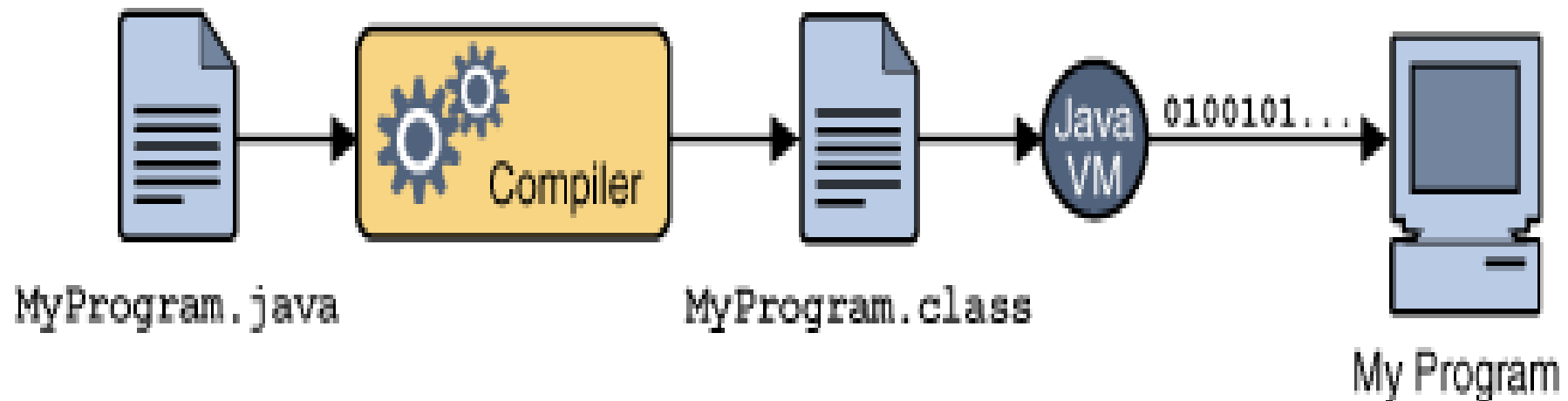
- The JRE que viene con el JDK (Java Development Kit) y contiene una completo conjunto de librerías para aplicaciones básicas, gráficas, web, redes, etc.
- El otro principal entorno de distribución es el navegador web. Casi todos los navegadores contienen el entorno de ejecución Java.

Algunas características de Java:

- JVM (Java Virtual Machine)
- Colector de basura (Garbage collector)
- Seguridad en el código ejecutado

Fases de la creación y ejecución de un Programa Java

- La siguiente figura muestra el proceso de compilar y ejecutar un programa Java



Fases de un Programa Java

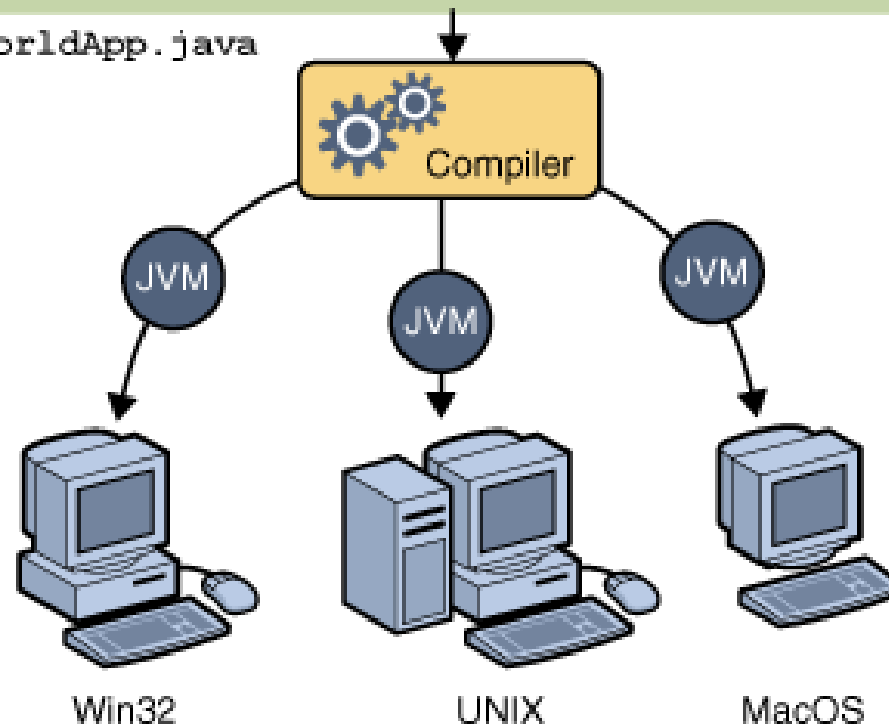
Tarea	Herramienta	Salida
Escribir un programa	Cualquier Editor de Texto	Un archivo con extensión <i>.java</i>
Compilar un programa	Compilador java	Un archivo con extensión <i>.class</i> (java bytecode)
Ejecutar un programa	Intérprete de java	Salida del programa

Java: Un solo código, varias plataformas

Java Program

```
class HelloWorldApp {  
    public static void main(String[] args) {  
        System.out.println("Hello World!");  
    }  
}
```

HelloWorldApp.java



Ejemplo de un código Java

Programa en C que imprime el famoso “Hola Mundo!!”

```
#include <stdio.h>

int main {
    printf("Hola Mundo!!\n");
    return 0;
}
```

Compilamos y ejecutamos

```
$ gcc -o test test.c
$ ./test
Hola Mundo!!
$
```

Ahora en Java. El archivo fuente igual que el nombre de la clase: *Test.java*

```
import java.io.*;

public class Test {

    public static void main ( String [] args ) {
        System.out.println("Hola Mundo!!");
    }

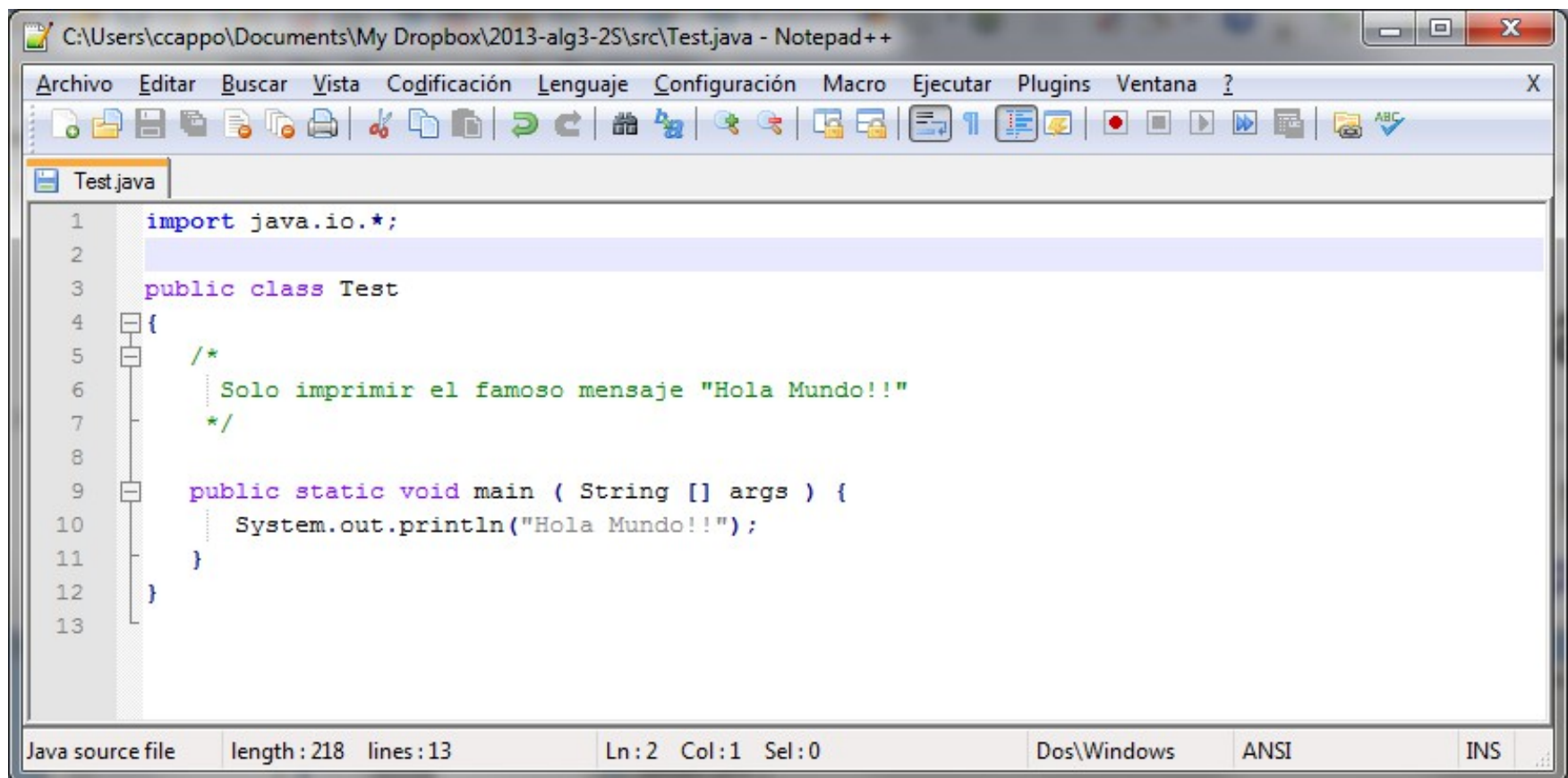
}
```

Compilamos y ejecutamos

```
$ javac Test.java
$ java Test
Hola Mundo!!
$
```

Probemos el compilador

Editar el archivo con cualquier editor de texto, usemos *un editor denominado Notepad++* (<http://notepad-plus-plus.org/>)

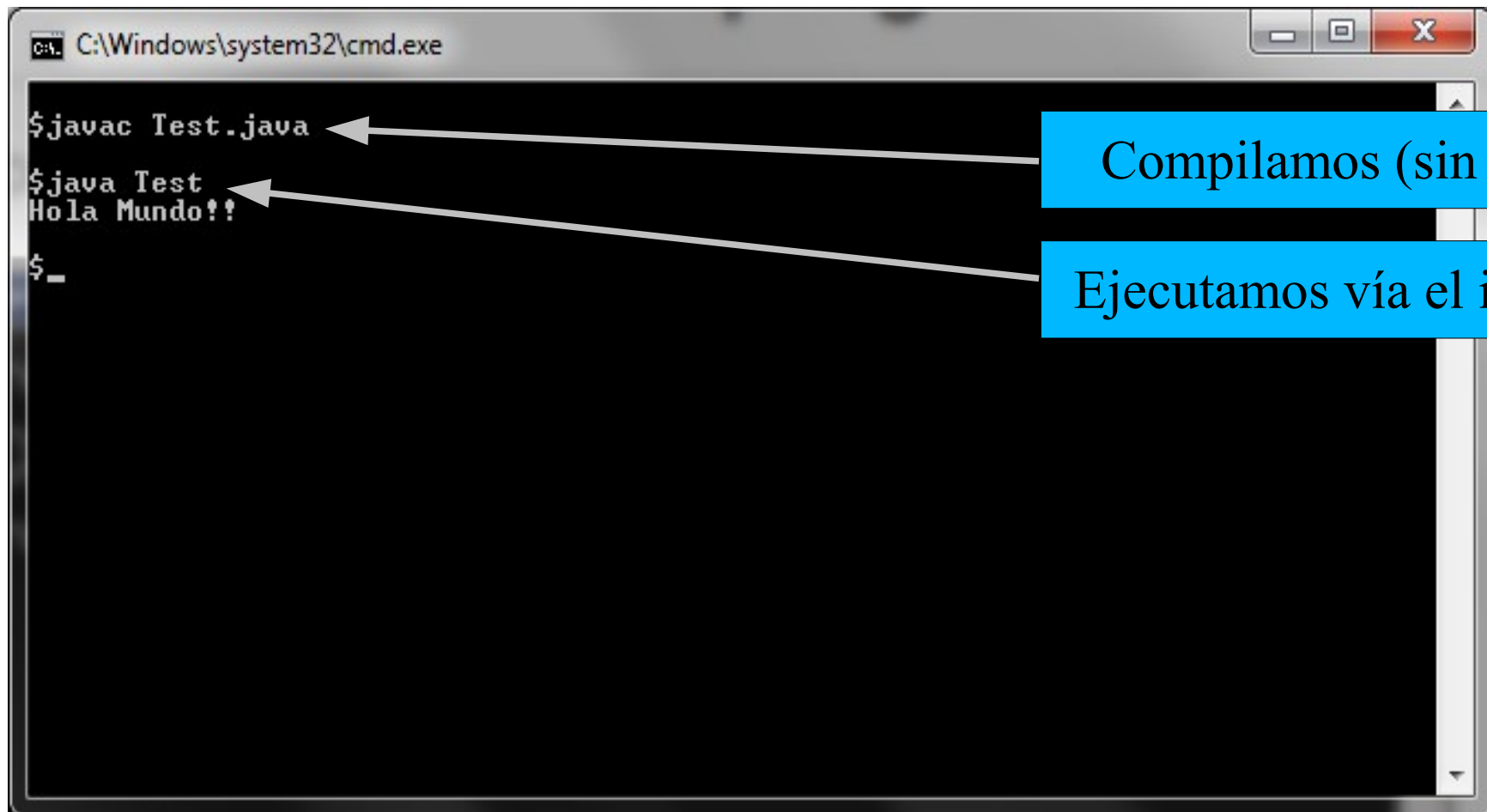


```
1  import java.io.*;
2
3  public class Test
4  {
5      /*
6       * Solo imprimir el famoso mensaje "Hola Mundo!!"
7       */
8
9      public static void main ( String [] args ) {
10         System.out.println("Hola Mundo!!");
11     }
12 }
13
```

Java source file length : 218 lines : 13 Ln : 2 Col : 1 Sel : 0 Dos\Windows ANSI INS

Probemos el programa

El compilador es *javac* y el intérprete *java*



```
C:\Windows\system32\cmd.exe

$javac Test.java
$java Test
Hola Mundo!!
$_
```

The screenshot shows a Windows command prompt window titled "C:\Windows\system32\cmd.exe". The prompt is "\$". The user enters "\$javac Test.java", which is followed by a blue callout box "Compilamos (sin errores)". The user then enters "\$java Test", which is followed by a blue callout box "Ejecutamos vía el intérprete". The output of the program is "Hola Mundo!!". The prompt then changes to "\$_" after the execution.

Compilamos (sin errores)

Ejecutamos vía el intérprete

Algunas características

- Sintaxis es muy similar a C y C++, de hecho casi iguales.
- Tipo de datos primitivos se mantienen entre plataforma y plataforma.
- Comentarios al estilo C y C++:

```
// unilínea  
/* multilínea */
```

Algunas diferencias con C

Característica	C	Java
Tipo de lenguaje	Orientado a funciones	Orientado a Objetos
Unidad básica de programación	Función	Clase = TAD
Portabilidad del código fuente	Posible con disciplina	Siempre
Portabilidad del código compilado	NO, dependiente de la arquitectura	SI
Seguridad	Limitada	Parte del lenguaje
Tipo numéricos	Dependiente de la plataforma	Fijos
Tipo booleano	Int y _Bool	boolean
Tipo caracter	ASCII	Unicode
Dirección de memoria	Puntero	Referencia
Arreglos	Estáticos (estáticos de stack)	Dinámicos de Heap con tamaño estático
Métodos de obtener y liberar memoria	<i>malloc()</i> y <i>free()</i>	<i>new</i> y liberación automática
Conversión explícita	Siempre es posible	Se controla
Conversión implícita	Siempre	Debe ser explícita si es con pérdida de precisión
Cadenas	Por convención	Incluido como tipo inmutable
Asignación de memoria para estructura de datos y arreglos	Heap, Stack, Data	Heap

Parte II

Al Contenido

Tipo de datos, operadores, expresiones

- Programa ejemplo
- Literales, identificadores, palabras reservadas
- Comentarios en Java
- Tipo de datos
- Variables: declaración e inicialización
- Sentencias y bloques de sentencias
- Salida de datos
- Operadores: aritméticos, relacionales, lógicos.

Ejemplo, un repaso

```
1  import java.io.*;
2
3  public class Test
4  {
5      /**
6       * Mi Primer programa Java
7       */
8      public static void main( String[] args ){
9          // Imprimo el mensaje "Hola Mundo!!"
10         System.out.println("Hola Mundo!!");
11         // Puedo hacerlo tambien como en C ;)
12         System.out.printf("%s", "Hola Mundo!!");
13
14     }
15 }
```

Ejemplo...

```
1  import java.io.*;  
2  
3  public class Test{  
4
```

- Una declaración ***import*** permite que un nombre de tipo pueda ser referido por un nombre simple (sin la especificación completa).
- En este ejemplo no es estrictamente necesario el uso de import ya que no tenemos ningún tipo que comience con *java.io.** (aquí mostramos solo para explicar el sentido de ***import***)

Ejemplo...

```
1 import java.io.*;  
2  
3 public class Test{  
4
```

- Indica que el nombre de la clase será **Test**
- En Java, todo el código debe ser puesto dentro de la declaración de una clase.
- La clase utiliza un modificador de acceso “**public**”, que indica que la clase es accesible desde otras clases del mismo paquete (*paquete=es una colección de clases agrupadas en un directorio*).

Ejemplo..

```
1  import java.io.*;
2
3  public class Test
4  {
5      /**
6       * Mi Primer programa Java
7       */
8      public static void main( String[] args ){
```

- Indica que el nombre de un método en **Test** que se llama **main**.
- El método **main** es el punto de entrada de un programa Java.
- Asegurarse de que el “*signature*” o “*firma*” del método tenga esta forma.

Ejemplo..

```
1  import java.io.*;
2
3  public class Test
4  {
5      /**
6       * Mi Primer programa Java
7       */
8      public static void main( String[] args ){
9
10         // Imprimo el mensaje "Hola Mundo!!"
11         System.out.println("Hola Mundo!!");
12
13     }
14 }
```

System.out.println(), imprime el texto encerrado entre comillas dobles en la salida estándar con un enter al final

Ejemplo..

```
1  import java.io.*;
2
3  public class Test
4  {
5      /**
6       * Mi Primer programa Java
7       */
8      public static void main( String[] args ){
9
10         // Imprimo el mensaje "Hola Mundo!!"
11         System.out.println("Hola Mundo!!");
12         // Puedo hacerlo tambien como en C ;)
13         System.out.printf("%s", "Hola Mundo!!");
```

System.out.printf(), igual que *printf* de C

Repasando..

- Recordar que:
 - La extensión de los archivos fuentes es **.java**
 - El nombre del archivo siempre debe coincidir con el nombre de la clase.
 - Conviene siempre que nuestro código tenga comentarios útiles

Java: *sintaxis y otras cosas*

- Recordar que mayúsculas y minúsculas son diferentes, así *Hola* es diferente a *HOLA*, *hola*, etc.
- Las sentencias siempre terminan en ; (punto y coma) igual que C y C++.
- Un bloque es un conjunto de sentencias que se inicia con un { y termina con un }.
- Java es de formato libre, es decir se puede “jugar” con los espacios, aunque es recomendable mantener siempre un estilo de programación (*ver recomendaciones del documento Java Code Conventions*)

Java: *identificadores*

- Se utiliza para nombrar variables, métodos, nombre de clases, etc.
- Deben empezar siempre con una letra, el signo _ o el signo \$.
- No pueden utilizarse palabras reservadas (keywords)
- Se recomienda:
 - Que los nombres de *clase* empiecen con Mayúscula.
 - Que los nombres de *métodos* y *variables* empiecen con minúsculas. Si es un nombre de método tiene dos palabras, el inicio de la siguiente palabra es en mayúscula. Por ejemplo: *esPrimo*, *agregarArista*, *etc.*

Java: *palabras reservadas*

Palabras clave en Java

abstract	assert	boolean	break	byte
case	catch	char	class	continue
default	do	double	else	enum
extends	final	finally	float	for
if	implements	import	instanceof	int
interface	long	native	new	package
private	protected	public	return	short
static	strictfp	super	switch	synchronized
this	throw	throws	transient	try
void	volatile	while		

Palabras clave que no se utilizan actualmente

const	goto
-------	------

Java : *literales*

- Del tipo enteros, pueden ser en diferentes “bases”.
 - Decimal : normal: **12121**, **12812**. Es del tipo *long* con el sufijo L, así **10L** es long no int.
 - Hexadecimal : precedido por 0x ó 0X . **0xAB**
 - Octal, precedido por 0 (cero): **014**
- Del tipo punto flotante
 - **583.45** (estandar) ó **5.8345e2** (científico)
 - **0.0d** (double) ó **0.0f** (float)
- Del tipo lógico
 - **true** o **false** (en minúsculas)

Java: literales (cont.)

- Del tipo caracter
 - Encerrados entre comillas simples 'a', 'b' '\n', '\b' (con secuencia de escape como C y C++)
 - *Obs: son del tipo UNICODE (16 bits) no ASCII*
- Cadenas (En Java no es primitivo pero tiene un tratamiento especial, el tipo de dato es *String*)
 - Secuencia de caracteres encerrados entre comillas dobles. Por ejemplo “Algoritmos y ED III”.

Java : tipo de datos primitivos – dominio

Tipo	Tamaño en bits	Valores	Estándar
boolean		true o false	
[Nota: una representación boolean es específica para la Máquina virtual de Java en cada plataforma].			
char	16	'\u0000 ' a '\uFFFF ' (0 a 65535)	(ISO, conjunto de caracteres Unicode)
byte	8	-128 a +127 (-2^7 a $2^7 - 1$)	
short	16	-32,768 a +32,767 (-2^{15} a $2^{15} - 1$)	
int	32	-2,147,483,648 a +2,147,483,647 (-2^{31} a $2^{31} - 1$)	
long	64	-9,223,372,036,854,775,808 a +9,223,372,036,854,775,807 (-2^{63} a $2^{63} - 1$)	
float	32	<i>Rango negativo:</i> -3.4028234663852886E+38 a -1.40129846432481707e-45 <i>Rango positivo:</i> 1.40129846432481707e-45 a 3.4028234663852886E+38	(IEEE 754, punto flotante)
double	64	<i>Rango negativo:</i> -1.7976931348623157E+308 a -4.94065645841246544e-324 <i>Rango positivo:</i> 4.94065645841246544e-324 a 1.7976931348623157E+308	(IEEE 754, punto flotante)

Java: *variables*

- Una variable es un ítem de dato asociado a un objeto en memoria.
- Tiene siempre un tipo de dato y un identificador o nombre.
- Declarar una variable

`<tipo_dato> <identificador> [= <valor_inicial>]`

Java: *tipos de variables*

- En Java existen dos tipos de variables
 - Primitivas
 - Referencias
- **Primitivas:** asociadas a tipos primitivos
 - Ejemplo: `int a, float b, boolean esPar;`
- **Referencia:** es un puntero que en realidad esta asociado a un dato en el HEAP (área de datos de la memoria). Asociado a objetos (de clases) y arreglos.
 - `String cad = "Hola" //cad es del tipo referencia.`

Java: *operadores*

- Tipos de operadores disponibles:
 - Aritméticos:
 - Binarios: *, /, +, -, %
 - Unarios: ++, --, -
 - Relacionales: >, <, <=, >=, ==, !=, instanceof
 - Bits: <<, >>, >>>, &, |, ^, !
 - Lógicos: && (and en sc), & (and), || (or en sc), |, ! (not)
 - Condicionales: ?:
 - Asignación: =, +=, -=, *=, /=, %=, ..

Operadores ordenados por precedencia

Operador	Descripción	Asociatividad
++ --	unario de postincremento unario de postdecremento	de derecha a izquierda
++ -- + - ! ~ (<i>tipo</i>)	unario de preincremento unario de predecremento unario de suma unario de resta unario de negación lógica unario de complemento a nivel de bits unario de conversión	de derecha a izquierda
* / %	multiplicación división residuo	de izquierda a derecha
+ -	suma o concatenación de cadenas resta	de izquierda a derecha
<< >> >>> < <= > >= instanceof	desplazamiento a la izquierda desplazamiento a la derecha con signo desplazamiento a la derecha sin signo menor que menor o igual que mayor que mayor o igual que comparación de tipos	de izquierda a derecha

Operador	Descripción	Asociatividad
==	es igual a	de izquierda a derecha
!=	no es igual a	
&	AND a nivel de bits AND lógico booleano	de izquierda a derecha
^	OR excluyente a nivel de bits OR excluyente lógico booleano	de izquierda a derecha
	OR incluyente a nivel de bits OR incluyente lógico booleano	de izquierda a derecha
&&	AND condicional	de izquierda a derecha
	OR condicional	de izquierda a derecha
?:	condicional	de derecha a izquierda
=	asignación	de derecha a izquierda
+=	asignación, suma	
-=	asignación, resta	
*=	asignación, multiplicación	
/=	asignación, división	
%=	asignación, residuo	
&=	asignación, AND a nivel de bits	
^=	asignación, OR excluyente a nivel de bits	
=	asignación, OR incluyente a nivel de bits	
<<=	asignación, desplazamiento a la izquierda a nivel de bits	
>>=	asignación, desplazamiento a la derecha a nivel de bits con signo	
>>>=	asignación, desplazamiento a la derecha a nivel de bits sin signo	

Parte III

Al Contenido

Entrada/Salida y Estructuras de Control

- Argumentos de la línea de comandos
- Lectura de datos desde la consola
- Estructuras de control

Java: conversiones

- **Conversión explícita:** a través de métodos específicos (`Math.round()`, `Integer.parseInt()`, etc)
- **Conversión automática:** para tipos primitivos y solo cuando es “legal” (de un tipo de rango menor a otro de rango mayor)
- **Casting** (conversión explícita): requerida por el programador y que implica pérdida de precisión
- Conversión automática para String: un tipo puede ser convertido a String automáticamente vía +

```
System.out.println("Hola" + 10.54f + !true);
```

Argumentos de la linea de comandos

```
1  import java.io.*;
2
3  public class FiboJava2
4  {
5      /**
6       * Lectura de param. desde la linea de comandos.
7       */
8      public static void main( String[] args ){
9
10         if ( args.length < 2 ) {
11             System.out.println("Debe ingresar dos valores..");
12             System.exit(1);
13         }
14
15         A = Integer.parseInt(arg[0]);
16         B = Integer.parseInt(arg[1]);
17         . . . .
```

Lectura desde la Consola (*forma complicada*)

```
private static int leer_int() {  
    BufferedReader stdin = new BufferedReader(  
        new InputStreamReader(System.in));  
  
    String linea = "";  
    int n        = 0;  
  
    while ( true ) {  
        try {  
            System.out.print("Introduzca un valor entero:");  
            linea = stdin.readLine();  
            n     = Integer.parseInt(linea);  
            break;  
        }  
        catch (Exception e) {  
            System.out.println("Error...intente de nuevo");  
        }  
    }  
    return n;  
}
```

Lectura desde la consola (forma más sencilla)

```
import java.util.*;
```

```
...
```

```
.. class ..{
```

```
    private static int leer_int() {  
        int n;  
        Scanner s = new Scanner(System.in);  
        n = s.nextInt();  
        return n;  
    }
```

```
}
```

Obtener siguiente entero.

Otros: nextLong, nextFloat,
nextDouble, nextBoolean, etc

Estructuras de control

- Permiten el cambio del flujo de ejecución de un programa.
- Dos posibles tipos:
 - **Condicionales o de decisión** : permiten seleccionar un trozo de código (if, if-else, switch)
 - **Iteración**: permiten repetir una porción de código (while, for, do-while).

Estructuras de control

Condicionales

```
if ( <expresion_booleana> )  
    <sentencia>;
```

```
if ( <expresion_booleana> ) {  
    <sentencia>;  
    <sentencia>;  
    . . .  
}
```

Bloque

Estructuras de control

Condicionales

```
if ( <expresion_booleana> )  
    <sentencia> | <bloque_sentencias>  
else  
    <sentencia> | <bloque_sentencias>
```

Estructuras de control

Condicionales

```
if ( <expresion_booleana> )  
    <sentencia> | <bloque_sentencias>  
else if ( <expresion_booleana> )  
    <sentencia> | <bloque_sentencias>  
else if ( <expresion_booleana> )  
    <sentencia> | <bloque_sentencias>  
..  
else  
    <sentencia> | <bloque_sentencias>
```

Estructuras de control

Condicionales - switch

- Permite la selección de varios bloques según el contenido de una expresión “entera” o una cadena (desde la versión 7).

```
switch( <expresion_entera_cadena> ){  
    case <constante_entera_cadena_1>:  
        statement1;//  
        statement2;//bloque 1  
        break;  
    case <constante_entera_cadena_2>:  
        statement1;//  
        statement2;//bloque 2  
        break;  
    default:  
        statement1;//  
        statement2;//bloque 3  
}
```

Estructuras de control

Condicionales - switch

- Java evalúa primero la expresión del switch y salta a la secuencia de sentencias del case que hace match. Fijarse que la secuencia no tiene “llaves”..
- Se ejecuta las sentencias hasta que se encuentre un “break” (OJO: el break es opcional!!!).
- Si ningún *case* coincide entonces se ejecuta la secuencia de sentencias asociadas con el default. La clausula *default* es opcional

Estructuras de control

Condicionales – switch – Ejercicio (completar...)

Se lee un número 1, -1 o cero y se debe imprimir la expresión “Cero”, “Negativo” o “Positivo” según el valor.

```
switch( n ){  
    case 0:  
        System.out.println("Cero");  
        break;  
    case 1:  
        System.out.println("Positivo");  
        break;  
    case -1:  
        System.out.println("Negativo");  
  
}
```

Estructuras de control

Repetición - while

```
while ( <expresion_booleana> )  
    <sentencia> | <bloque_sentencia>
```

- Características:
 - Ciclo se ejecuta 0-N veces.
 - Si la expresión booleana es true entonces se ejecuta la sentencia o el bloque de sentencias

Estructuras de control

Repetición - do-while

```
do {  
    <lista_sentencias>  
} while ( <expresion_booleana> );
```

- Características
 - La lista de sentencias se ejecuta 1-N veces
 - Mientras la expresión es true se mantiene en el ciclo

Estructuras de control

Repetición – do-while (Ejercicio)

Imprimir la cantidad de dígitos que tiene una variable entera n.

Aquí una posible solución en Java.

```
c = 0; //En c dejamos la cantidad de dígitos
do {
    n = n / 10; //n es el número
    c++;
} while ( n > 0 );
// Imprimir c
```

Estructuras de control

Repetición – for

```
for ( <expr_1>; <expr_2>; <expr_3> )  
    <sentencia> | <bloque_sentencia>
```

La semántica del for es:

```
<expr_1>  
while ( <expr_2> ){  
    <sentencia> | <bloque_sentencia>  
    <expr_3>;  
}
```

Parte IV

Arreglos y cadenas

Al Contenido

- Arreglos
- Cadenas

Cadenas..

- Es un objeto.. es decir no es un tipo primitivo. Son dinámicos.
- Es tratado por el compilador de una manera especial en relación a los otros “objetos” de otras clases. Por ejemplo el tratamiento de la concatenación (+) y otras cosas (es inmutable.. *que es esto?*)
- Los caracteres componentes son Unicode no ASCII.
- Su tipo es **String**

```
String s = new String("Hola");
```

```
String s = "Hola"; // Preferible para constantes.
```

```
String s1 = "Hola"; // El compilador optimiza el espacio.
```

Cadenas (métodos usuales)

SOME STRING CLASS METHODS

Method

`char charAt(int index)`

`String concat(String str)`

`Boolean endsWith(String str)`

`boolean equals(String str)`

`boolean equalsIgnoreCase(String str)`

`int indexOf(char c)`

`int indexOf(char c, int n)`

`int indexOf(String str)`

`int indexOf(String str, int n)`

`int length()`

`String replace(char c1, char c2)`

`String substring(int n1, int n2)`

`String toLowerCase()`

`String toUpperCase()`

`String valueOf()`

Description

Returns the character at the specified index.

Concatenates this string with another and returns the result.

Tests whether this string ends with the suffix *str*.

Compares this string to another and returns true if they're equal.

Tests whether this string equals *str*, ignoring case.

Returns the integer index of char *c* at its first occurrence.

Returns the first index of *c* starting at index *n*.

Returns the index of the first occurrence of *str*.

Returns the index of the first occurrence of *str* starting at index *n*.

Returns the length (number of characters) of this string.

Replaces all occurrences of *c1* with *c2* and returns the result.

Returns the substring of this string between index *n1* and *n2*.

Converts all letters in this string to lowercase.

Converts all letters in this string to uppercase.

Converts the argument to a string. There are versions of this method that accept boolean, char, double, float, int, and long data types.

Cadenas

Ejemplo: dada una cadena imprimir un “-” (guión) entre cada caracter de la misma.

Aquí una posible solución en Java.

```
String s = "HOLA";  
for ( int k=0; k < s.length(); k++ )  
    System.out.print(s.charAt(k) + "-");
```

¿Qué pasa cuando hago `<= s.length()`?

Arreglos

- ¿Qué es un arreglo? Dato estructurado cuyos componentes son del mismo “tipo”. Existen en todo los lenguajes y es una facilidad que se me brinda para solucionar más fácilmente algunas cuestiones.

- Ya lo vimos con el método main

```
public static void main (String [] arg)
```

- Declaración:

```
<tipo_dato> [] <identificador>
```

```
int [] A ; // Arreglo de ints.
```

```
String [] Palabras; // Arreglo de Cadenas.
```

Arreglos

- Un arreglo es en realidad un “objeto” para Java. Es un tipo de dato “referencia” (*un puntero*)
- Cuando yo declaro un arreglo fijarse que no le digo la cantidad de elementos que contendrá (como en C o C++).
- Instanciación (en Java quiere decir “creación”)

```
<var_Arreglo> = new <tipo_dato_arreglo> [<tamaño>]
```

```
int [] A;
```

```
A = new int [100] ; // A ahora tiene 100 ints.
```
- La variable de tipo arreglo declarada pero no instanciada contiene tiene por defecto el valor “*null*”, que es como decir “*no apunta a nada por ahora*”.

Arreglos

- Acceso a componentes:

```
<var_Arreglo>[<indice>]
```

```
int [] A = new int[100] ; //podemos inicializar.
```

```
A [0] = 100 ; //fijarse que el 1er. elemento esta en 0.
```

- Instanciación (Inicialización estática)

```
int [] A = {10,20,30,40,50};
```

```
String [] Palabras = {"Hola", "que", "tal"};
```

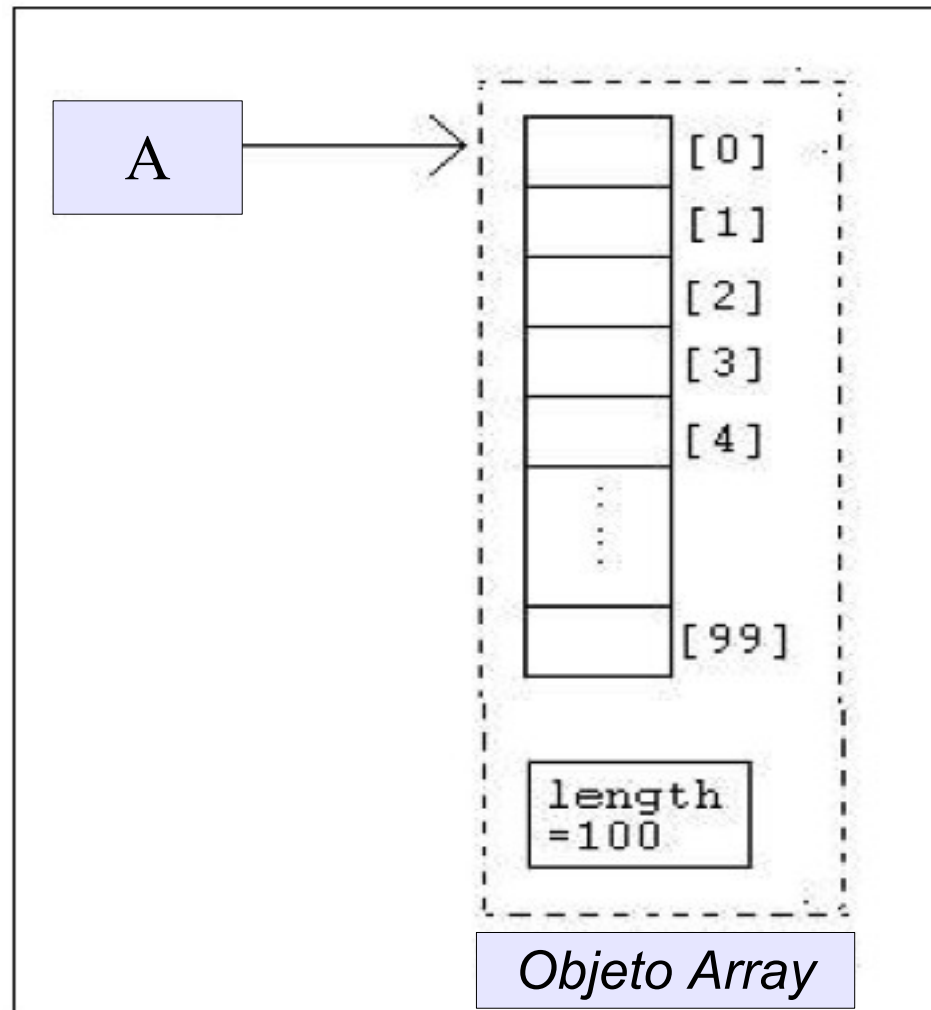
```
boolean [] Resultados = {true, false, true, true};
```

- Cantidad de elementos:

```
<var_Arreglo>.length //Fijarse que no tiene ()
```

Arreglos

- Ejemplo: `int [] A = new int[100];`



Arreglos multidimensionales

```
int [] [] A = { { 10,20,30,40,50} ,  
                { 10,20,30,40,50}  
              };
```

```
System.out.println(A[0][0]); // Imprimiría 10
```

```
String [] [] Palabras = { {"Hola"}, {"que"}, {"tal"} };  
// imprimir todo el contenido de Palabras.  
for (int f=0; f < Palabras.length; f++ )  
    for ( int c=0; c < Palabras[f].length; c++ )  
        System.out.println(Palabras[f][c]);
```

Parte V

Introducción a POO. Clases

Al Contenido

- Conceptos básicos sobre Programación Orientada a Objetos.
- Clases. Definición. Uso. Características en Java.
- Ejercicios sencillos de definición de clases y de uso.

Repaso

- Cadenas. Recordar que es un tipo de dato que en Java es inmutable.
 - Recordar la diferencia entre utilizar **new** y asignación directa con el operador **=**.
- Arreglos.
 - Es un tipo de dato estructurado de tamaño dinámico.
 - Es mutable. Para Java es un tipo de *Objeto* particular.
 - El primer elemento siempre empieza en la posición cero.
 - Se “instancia” (recordar que es esto) con el operador **new**

Conceptos básicos sobre POO

- Recordar que la OO es un técnica cuyo objetivo es de alguna forma “modelar” el mundo real de una forma más natural.
- Detrás de esto existe una serie de ventajas:
 - Escribir menos, mejorar el mantenimiento del código, aprovechar el código ya escrito entre otras cosas.
- Existen tres características básicas para que un lenguaje sea considerado orientado a objetos:
 - *Encapsulamiento de datos* (asociar dato y proceso aplicables a los datos). *Ocultamiento de información*.
 - *Herencia* .
 - *Enlace dinámico* (polimorfismo).

Conceptos básicos

- **Clase:** se puede decir que es la definición de las características de cierto tipo de objeto del mundo real. Por ejemplo *Persona* es una clase. *Auto* es una clase.
- **Objeto:** es una instancia “viva” de cierta clase.
 - *Persona* es la clase y *Juan Perez* es una instancia de la clase *Persona*.
- Las clases tienen dos componentes fundamentales:
 - **Atributos** (*características del objeto*) o datos miembros
 - En *Persona* sería: `apellido`, `nombre`, `estatura`, `peso`, etc.
 - **Métodos** (*qué puedo hacer o que puede hacer el objeto*).
 - En *Persona* sería: `cambiarPeso()`, `cambiarApellido()`, etc.

Sintaxis de creación de clases

- Para definir un clase en Java

```
<modificador> class <identificador_clase> {  
    <atributos>*  
    <constructores>*  
    <métodos>*  
}
```

donde

- <modificador> indica es el modificador de acceso a la clase (public, private, protected, package)

Ejemplo: *Persona*

- Qué características comunes tienen las Personas (atributos)
- Qué acciones pueden hacer las personas o pueden aplicarse a las Personas. (métodos aplicables).
 - Persona:
 - apellido, nombre, cedula, fecha de nacimiento, etc.
 - Qué podemos hacer
 - imprimir sus datos, asignar su nuevo nombre, cambiar su cédula, corregir su nombre, etc, etc.

Persona. Definición Java

```
public class Persona {
    private String nombre ;
    private String apellido ;
    private String cedula;

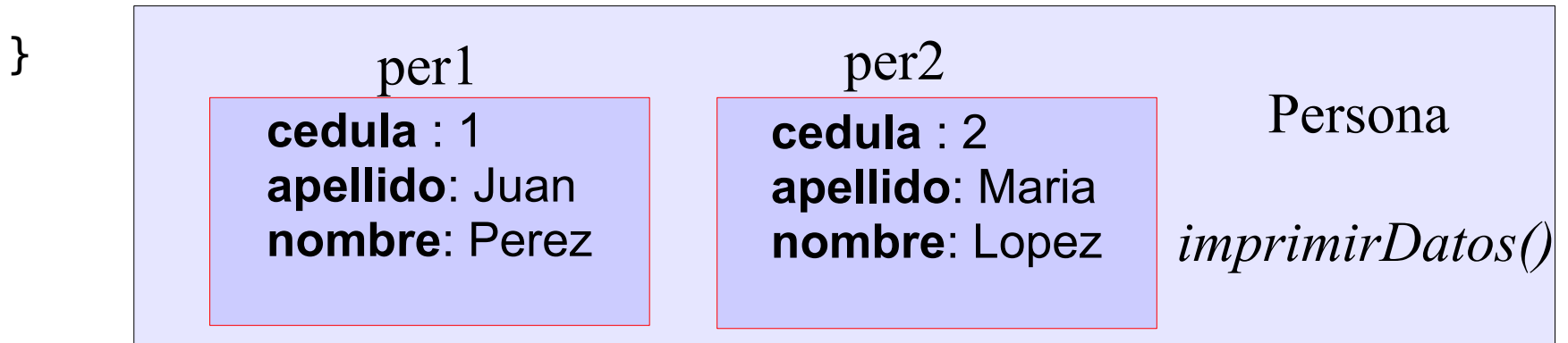
    /* Constructor de persona */
    Persona ( String nom, String ap, String ced ) {
        this.nombre     = nom;
        this.apellido    = ap;
        this.cedula      = ced;
    }

    public void imprimirDatos(){
        System.out.println("Cedula   : " + this.cedula   +
                           "Apellido: " + this.apellido +
                           "Nombre   : " + this.nombre );
    }
    .....
}
```

Usando la clase Persona

```
import Persona; // No es necesario si estoy en el mismo
                // directorio

public class  UsoPersona {
    public static void main (String [] args ) {
        Persona per1 = new Persona ("1", "Juan", "Perez");
        Persona per2 = new Persona ("2", "Maria", "Lopez");
        per1.imprimirDatos();
        per2.imprimirDatos();
    }
}
```



Variables de instancia vs. Variables de clase

Clase Auto		Objeto Auto A	Objeto Auto B
Variables de instancia	Número de placa	ABC 111	XYZ 123
	Color	Azul	Rojo
	Fabricante	Nissan	Toyota
	Velocidad máxima	120 km/h	180 km/h
Variable de clase	Cantidad = 2		
Métodos de instancia	Método Acelerar		
	Método Encender		
	Método Frenar		

Variables de instancia

- No tienen el modificador “*static*” por delante.

```
public class Auto {
```

```
    private int color;  
    private int cantPuertas;  
    private String Marca;  
    private String Modelo;
```

```
    . . . . .
```

- Estas variables existen por cada instancia creada de la clase (por cada **new** que hago)

Variables de clase

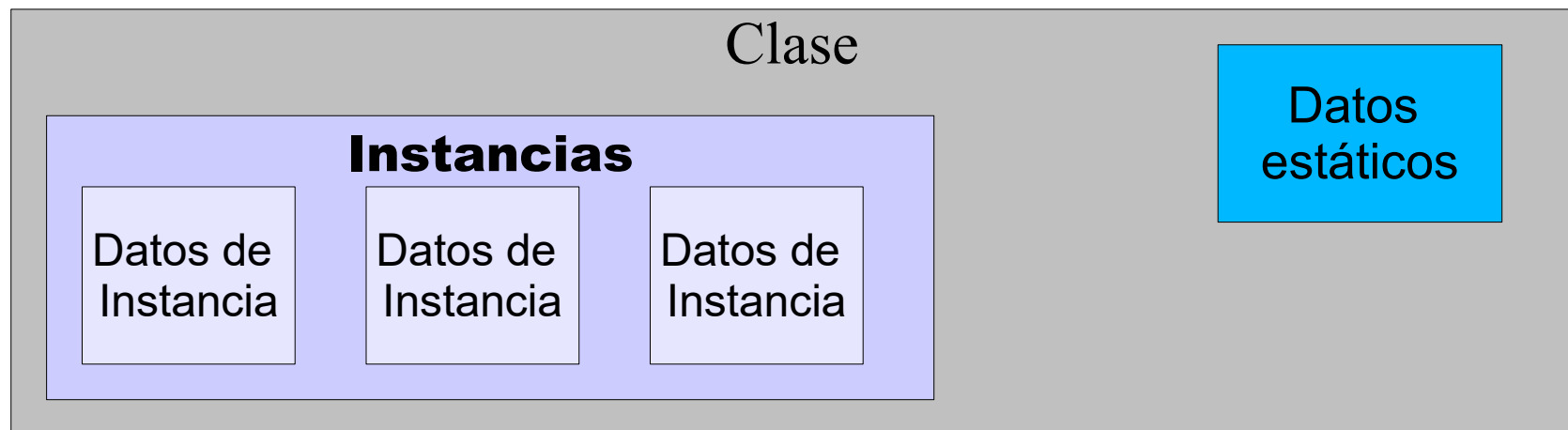
- Tienen el modificador “static” por delante.

```
public class Auto {
```

```
    private static int cantidad_Autos_Definidos;
```

```
    ....
```

```
// Es una sola instancia para toda la clase.
```



Métodos de instancia vs. Métodos estáticos

- **Métodos de instancia:** actúan sobre variables de instancia o variables de clase. No tienen “static” como modificador.
- **Métodos de clase:** actúan solo sobre variables de clase y pueden ser invocados sin necesidad de instanciar ningún objeto.
 - Ejemplos:
 - `System.out.println(...)` // Mirar en la documentación la clase `System` y el dato miembro `out`.
 - `Integer.parseInt(..)` // Mirar la documentación.

Accesos a variables y métodos

- **public**
 - Cualquier clase puede acceder.
- **package** o friendly (*cuando no se define modificador este es el acceso otorgado*)
 - Pueden acceder clases del mismo paquete (y la misma clase)
- **protected**
 - Pueden acceder solo clases “hijas” (y la misma clase)
- **private**
 - Solo la propia clase puede acceder

Constructores

- Es un método especial que tiene el mismo nombre de la clase.
- Su objetivo es ejecutar código de inicialización sobre el objeto que se está creando.
- Se le llama justo en el momento de instanciar al objeto.
- Si no colocamos ningún constructor Java coloca uno por defecto.
- Pueden existir muchos constructores que se diferencian por la cantidad y tipo de los parámetros que recibe.

Constructores - ejemplo

definición

```
public class Persona {
```

...

```
Persona ( ) {
```

```
    apellido = "";
```

```
    nombre   = "";
```

```
    cedula   = "";
```

```
}
```

```
Persona (String a, String n, String c) {
```

```
    apellido = a;
```

```
    nombre   = n;
```

```
    cedula   = c;
```

```
}
```

....

Constructores - ejemplo

llamada

```
import Persona;

public class UsoPersona {

    .....

    public static void main (String [] args ) {

        Persona per1 = new Persona();
        Persona per2 = new Persona ("1","Juan","Perez");

        .....
    }
}
```

Ejercicio

- Defina la clase **ListaEnlazadaSimple** con las siguientes operaciones:
 - Agregar (por el frente de la lista)
 - Remove (del frente de la lista)
 - Iterar sobre la lista (métodos: *hasNext*, *next*) *next()* retorna el dato actual y avanza. *hasNext()* retorna true si hay un dato siguiente



cabeza

Ejemplo de uso de *ListaEnlazadaSimple*

- Ejemplo de agregar a un lista varios nombres.

```
ListaEnlazadaSimple amigos = new ListaEnlazadaSimple();  
amigos.agregar("Juan");  
amigos.agregar("Maria");  
amigos.agregar("Pedro");  
// Ahora listamos el contenido  
for ( ; amigos.hasNext(); )  
    System.out.println((String)amigos.next() );
```

Parte VI

Herencia

Al Contenido

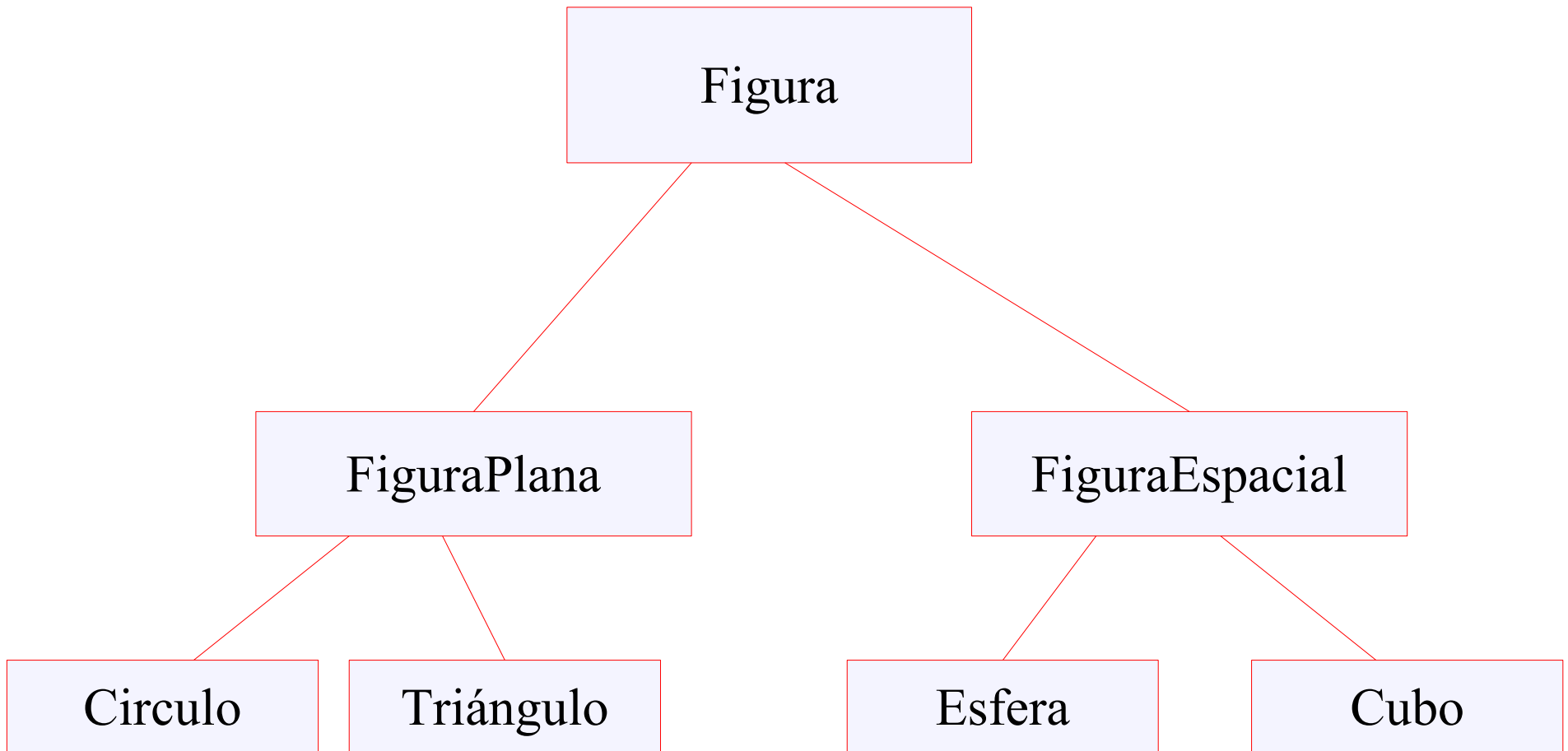
- Herencia. Conceptos básicos y forma de aplicarlo en Java.
- Ejercicios sencillos de definición de clases y sub-clases.

Herencia *(usaremos poco en el curso)*

- **Objetivo principal:** reutilización de código
 - Uno define características generales en una clase y se heredan en las clases hijas
 - Las clases hijas heredan todas las propiedades y métodos, pueden definir sus propios atributos y métodos o cambiar funcionalidades ya existentes.
- **Definiciones**
 - superclases
 - subclases o clases hijas

Ejemplo

fijarse que se forma una jerarquía



Herencia en Java

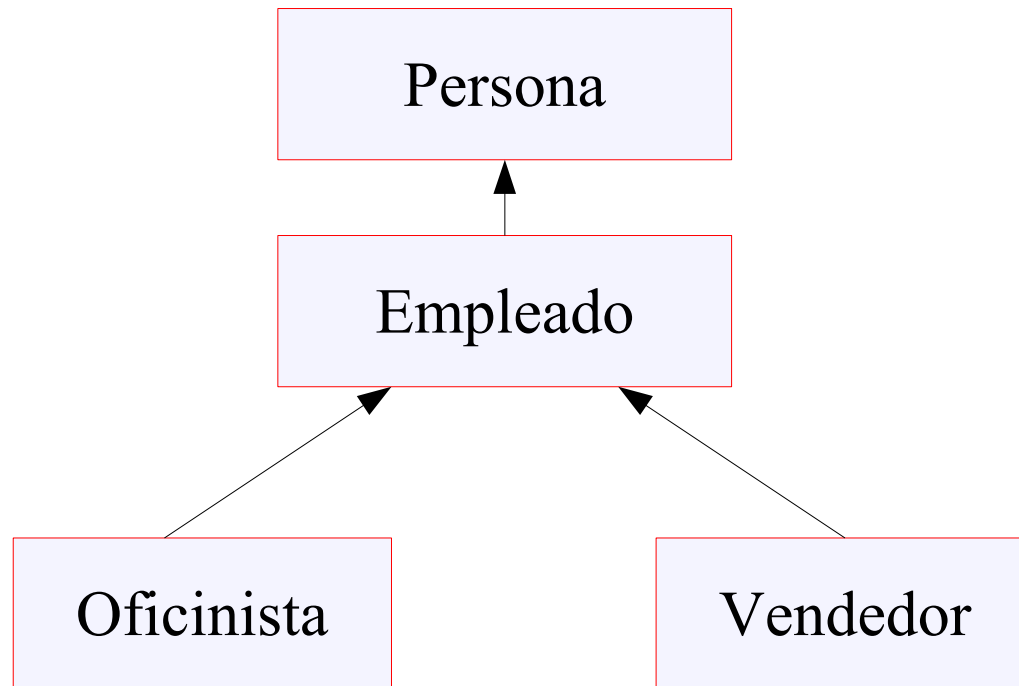
```
public class Figura {  
    private String nombre;  
    public float dibujar ();  
}  
  
public class FiguraPlana extends Figura{  
    private int x; // Coordenada  
    private int y; //  
  
    public float superficie();  
}  
  
public class Circulo extends FiguraPlana {  
    private float diametro;  
    public float superficie () { float radio = diametro/2;  
        return Math.PI*radio*radio;  
    } ...  
}
```

Algunas cuestiones de Java

- Todos los métodos son virtuales (modificables en las clases hijas) excepto cuando se coloca el modificador “*final*”.
- Solo existe Herencia simple (en C++ hay herencia múltiple). (*Una clase puede tener un solo “padre”*)
- Cualquier clase en Java siempre es hija de la clase **Object** (esta clase tiene algunas funcionalidades ya definidas).
- Los constructores no se heredan.

Ejemplo de Herencia

- **Extender la clase Persona:** de forma que ahora tengamos tres clases derivadas. Empleado y este tenga dos subclases que sea Oficinista y Vendedor.



Parte VII

Manejo de Excepciones

Al Contenido

- Excepciones. Concepto. Definición y uso de excepciones.
- Agregar excepciones a la jerarquía de clases implementada.

Manejo de Excepciones

- ¿Qué es una excepción?
 - Evento inusual que ocurre en tiempo de ejecución y que causa una interrupción en el flujo normal de ejecución
 - Ejemplos:
 - División por cero
 - Acceso fuera de los límites de un arreglo
 - Entrada inválida
 - Problemas en el disco
 - Archivo no existe
 - No hay más memoria
 - No hay conexión de red
 - Etc, etc

Ejemplo de una excepción

```
1 class DivisionPorCero {  
2     public static void main(String args[]) {  
3         System.out.println(3/0);  
4         System.out.println("Luego de la division.");  
5     }  
6 }  
7 // Como reacciona este programa?
```

Excepción – Salida del ejemplo

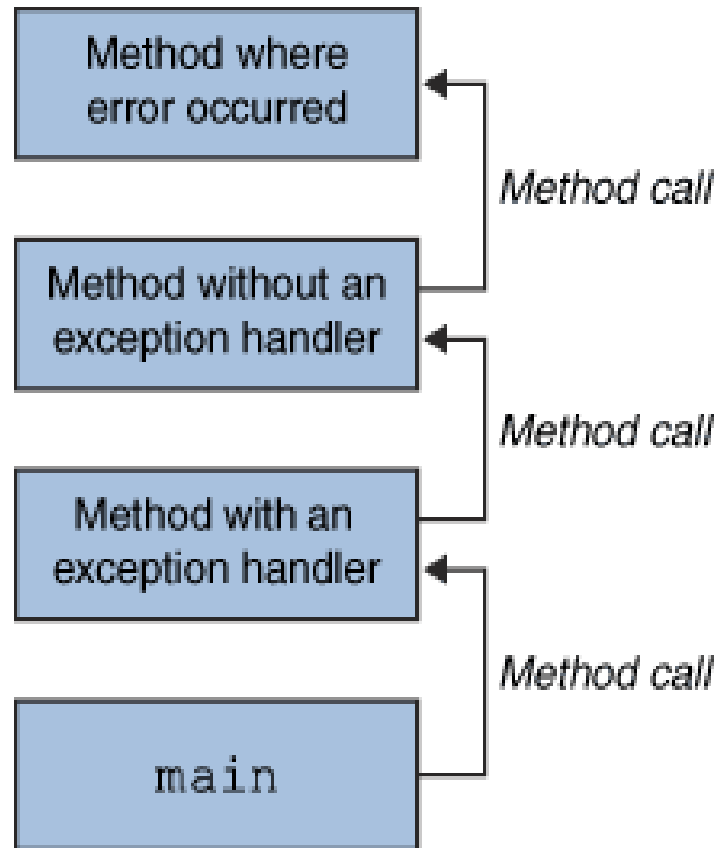
```
Exception in thread "main"  
    java.lang.ArithmeticException: / by zero  
        at DivisionPorZero.main(DivisionPorCero.java:3)
```

- En Java el manejador de excepciones por defecto:
 - Es proveído por el runtime
 - Imprime la descripción de la excepción.
 - Imprime la llamada de los métodos que causó la excepción
 - Termina el programa

Excepciones

- ¿Qué ocurre cuando existe una excepción en un método?
 - Se crea un “objeto” de Excepción que es pasado al intérprete. Esto se llama “lanzamiento de una excepción” (*throw*).
 - El objeto de excepción contiene información acerca del error que ocurrió.
 - El Runtime busca en la secuencia de llamadas el método que manejaría la excepción, finalmente si no lo encuentra el programa termina.

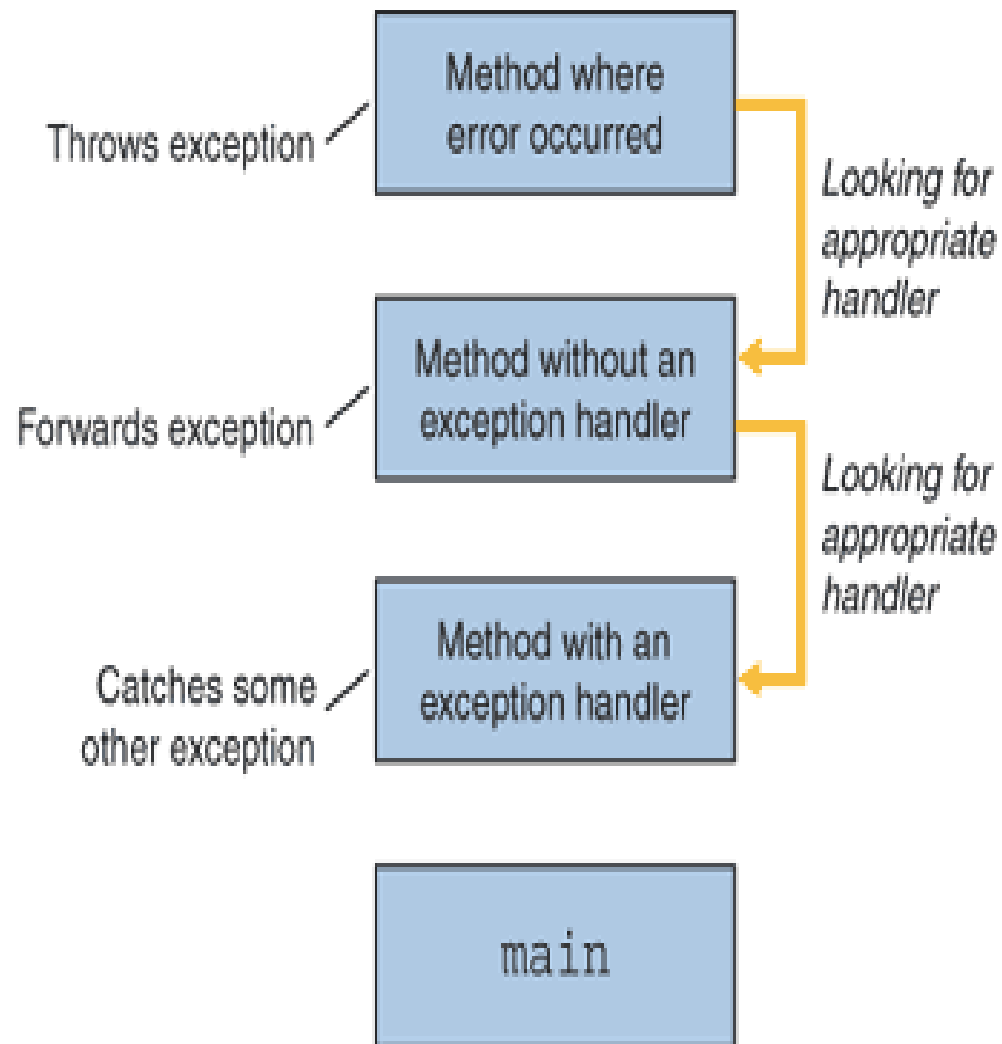
Excepciones



Excepciones

- Si existe un método que pueda manejar la excepción
 - El Runtime pasa el objeto de Excepción a este método.
 - El método en este caso define las excepciones que se atraparán. (*catch*)

Excepciones



Excepciones - Beneficios

- No mezclamos el manejo de errores con el código “normal”.
- Los errores pueden propagarse
- Podemos agrupar o categorizar los tipos errores (ya que las excepciones son objetos de la clase *Exception*)

Excepciones

- Atrapando excepciones

```
try {  
    <codigo_protegido>  
} catch (<TipoExcepcion> <Variable>) {  
    <Manejador de la excepción>  
}  
...  
} catch (<TipoExcepcion> <Variable>) {  
    <Manejador de la excepción>  
}
```

Excepciones – try-catch (ejemplo)

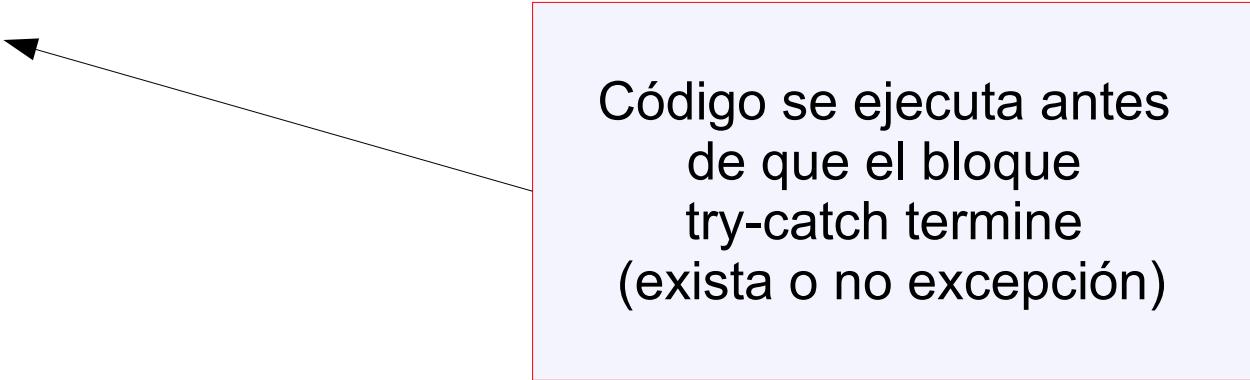
```
1 public class DivisionByCero {
2     public static void main(String args[]) {
3         try {
4             System.out.println(3/0);
5             System.out.println("Luego de la division.");
6         } catch (ArithmeticException exc) {
7             //Division por zero es ArithmeticException
8             System.out.println(exc);
9         }
10        System.out.println("Luego de la excepción.");
11    }
12 }
```

Excepciones – múltiples catch

```
1 public class MultipleCatch {  
1     public static void main(String args[]) {  
2         try {  
3             int den = Integer.parseInt(args[0]);  
4             System.out.println(3/den);  
5         } catch (ArithmeticException exc) {  
6             System.out.println("Divisor fue 0.");  
7         } catch (ArrayIndexOutOfBoundsException exc2) {  
8             System.out.println("Faltan argumentos");  
9         }  
10        System.out.println("Luego de la excepción.");  
11    }  
12 }
```

Excepciones - finally

```
try {  
    <codigo_protegido>  
} catch (<TipoExcepcion> <Identificador>) {  
    <Manejador_de_la_excepcion>  
} ...  
} finally {  
    <finalización>  
}
```



Código se ejecuta antes
de que el bloque
try-catch termine
(exista o no excepción)

Lanzar excepciones

- Un método siempre debe atrapar las excepciones o listar las que va a dejar pasar. Excepto para las clases Error o RuntimeException o sus derivados.
- Si un método puede causar una excepción y no la atrapa entonces debe mencionar en su definición que tipo de excepciones dejará pasar (usando la cláusula throws)

```
<tipo> <Nombre_Metodo> (<lista_parametros>) throws  
    <Lista_Excepciones_no_atrapadas> {  
    <cuerpo_método>  
}
```

Ejemplo de throws..

```
public static int leer_int ( )    {
    BufferedReader stdin = new BufferedReader(
        new InputStreamReader (System.in));

    String linea = "";
    int n = 0;

    while ( true ) {
        try {
            linea = stdin.readLine();
            n = Integer.parseInt(linea);
            break ;
        }
        catch ( Exception e ) {
            //No hacemos nada
        }
    }
    return n;
}
```

Ejemplo de throws

```
public static int leer_int ( ) throws Exception {  
    BufferedReader stdIn = new BufferedReader(  
        new InputStreamReader (System.in));  
  
    String linea = "";  
    int n = 0;  
  
    linea = stdIn.readLine();  
    n = Integer.parseInt(linea);  
    return n;  
}  
  
public static void main(String[] args) throws Exception {  
    int n = leer_int();  
  
    System.out.println("El valor leido fue " + n);  
}
```

Creando nuestro propio tipo de excepción

- Por ejemplo en el caso de *Fecha* creamos una excepción en caso de que los parámetros de año, mes y día son incorrectos.
- Asumir que el año debe estar entre 900 y 2099, el mes debe estar en 1 y 12 y el día debe estar entre 1 y 31.
 - La cantidad máxima para cada mes (de enero a diciembre) es:
31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31.
 - Para Febrero es 29 el tope si el año es bisiesto. Un año es bisiesto si es múltiplo de 4 pero no de 100 (a no ser que sea de 400).

Ejemplo de definición

```
class FechaInvalida extends Exception {  
    FechaInvalida ( String msg ) {  
        super(msg);  
    }  
}  
  
public class Fecha {  
    public Fecha ( int a, m , d ) throws FechaInvalida {  
        si ( fecha es invalida .. ) {  
            throw new FechaInvalida("La fecha es incorrecta");  
        }  
    }  
    ....  
}
```

La jerarquía de Excepciones en Java

- Throwable
 - Es la raíz de todas las excepciones
 - Dos subclases
 - Error : generadas por el propio runtime,
 - Exception : las que pueden ser “controladas” por el programador.

Exception Class Hierarchy			
Throwable	Error	LinkageError, ...	
		VirtualMachineError, ...	
	Exception	ClassNotFoundException,	
		CloneNotSupportedException,	
		IllegalAccessException,	
		InstantiationException,	
		InterruptedException,	
		IOException,	EOFException,
			FileNotFoundException,
			...
		RuntimeException,	ArithmeticException,
			ArrayStoreException,
			ClassCastException,
			IllegalArgumentException,
			(IllegalThreadStateException and NumberFormatException as subclasses)
			IllegalMonitorStateException,
			IndexOutOfBoundsException,
			NegativeArraySizeException,
			NullPointerException,
			SecurityException
		...	

A tener en cuenta

- Existen dos tipos de excepciones:
 - Checked (*comprobadas*):
 - El compilador detecta que sean atrapadas o dejadas pasar.
 - Derivan de *Exception* excepto *RuntimeException*
 - Unchecked (*no comprobadas*)
 - Se refieren a excepciones que no se detectan en tiempo de compilación
 - *Error*, *RuntimeException* (por ejemplo division por cero).

Parte VIII

Al Contenido

Clases abstractas, interfaces y generic

- **Clases Abstractas.** Uso y definición en Java.
- **Interface.** Concepto. Uso y definición en Java. Ejemplo sencillo de uso.
- **Generics.** Concepto. Utilización. Sintaxis y ejemplos sencillos.

Clases abstractas

- Una clase abstracta es aquella que tiene uno o más métodos abstractos,
 - Un método es abstracto si no tiene implementación o cuerpo
- **public abstract float getSueldo();**
- Una clase abstracta también puede ser definida como:
public abstract class Empleado {...};
- Estas clases no pueden ser instanciadas.
- Sus descendientes tienen que implementar los métodos abstractos.
- Permiten “definir” funcionalidades básicas que deben tener las clases hijas, sin decir todavía “como”.

Clases abstractas

- Un ejemplo de esto puede aplicarse a la clase *Empleado*.
- En la empresa no existe un Empleado que no sea Oficinista, Vendedor u otro. Pero necesitamos definir las funcionalidades generales de Empleado para tener métodos que actúen sobre cualquier tipo de Empleado.
- Ejemplos:
 - Cambiar en la clase Empleado los métodos de *getSueldo* y *getTipoEmpleado* añadiéndoles el modificador *abstract*
 - Qué pasaría si intentamos instanciar la clase Empleado?
`Empleado e = new Empleado("1", "Torres", "Juan");`

Interfaces

- Es una forma de definir el comportamiento de las clases, es lo que se denomina “*software por contrato*”.
- Es como una clase abstracta pero sin ninguna implementación.
- Se pueden definir en una interfaz métodos y constantes.
- Todos los métodos son abstractos, es decir, sin implementación
- Una clase concreta puede “implementar” una interface, lo que le obliga concretar los métodos definidos en la interfaz.
- Una clase puede implementar más de una interface.

Interface

Ejemplo de definición

```
public interface Serial {  
    int siguiente();  
    void reset();  
    void reiniciar(int x);  
}
```

```
public class De-A-Dos implements Serial {  
    private int actual;  
    private int inicio;  
    public PorDos(){  
        reset();  
    }  
}
```

```
public int siguiente(){  
    actual += 2;  
    return actual;  
}
```

```
public void reset(){  
    actual = 0;  
    inicio = 0;  
}
```

```
public void reiniciar(int x){  
    actual = x;  
    inicio = x;  
} } /* Fin de PorDos */
```

Interface

Ejemplo 2

```
public interface EmiteRuido {  
    public void hacerRuido();  
}  
  
public class Humano implements EmiteRuido {  
    private String nombre;  
  
    public Humano (String s) { this.nombre = s);  
  
    public void hacerRuido() {  
        System.out.println("HOLA, yo puedo hablar y soy " + nombre);  
    }  
}  
  
public class Auto implements EmiteRuido {  
    public void hacerRuido() {  
        System.out.println("RRRRRRRRMMMMMM");  
    }  
}
```

Interface

```
public class Perro implements EmiteRuido {  
    public void hacerRuido() {  
        System.out.println("GAU-GAU");  
    }  
}  
  
public class Gato implements EmiteRuido {  
    public void hacerRuido() {  
        System.out.println("MIAUUUU");  
    }  
}  
  
public class Sirena implements EmiteRuido {  
    public void hacerRuido() {  
        System.out.println("UUUUUUHHHHH");  
    }  
}
```

Interface

```
public class Ciudad {  
    private ArrayList<EmiteRuido> cosas;  
  
    public Ciudad () {  
        cosas = new ArrayList<EmiteRuido>();  
    }  
  
    public void agregar(EmiteRuido e) {  
        cosas.add(e);  
    }  
  
    public void escucharRuidos() {  
        for ( EmiteRuido e : cosas )  
            e.hacerRuido();  
    }  
}
```


Interface

```
public class UsoCiudad {  
  
    public static void main( String [] args ) {  
        Ciudad c = new Ciudad();  
        EmiteRuido r = new Gato(); //Puedo usar como "tipo".  
        c.agregar(r);  
        c.agregar(new Perro());  
        c.agregar(new Humano("Maria"));  
        c.agregar(new Humano("Luis"));  
        c.agregar(new Sirena());  
        c.agregar(new Auto());  
  
        c.escucharRuidos();  
    }  
  
}
```

Interface -beneficios

- Muy interesante cuando se hace trabajo en equipo.
- Revelar los requerimientos no la implementación
Concepto de encapsulamiento
- La implementación puede cambiar sin afectar a los usuarios
- El que usa la *interface* no necesita tener la implementación durante el desarrollo.
- Podemos relacionar clases “no *conectadas*” (como el ejemplo).
- De alguna manera suple la herencia múltiple.

Interface

- Para definir una interface

```
public interface <identificador> {  
    // métodos sin implementación  
}
```

- Una interfaz puede extender otra

```
public interface EmiteRuidoMasFuerte extends EmiteRuido {  
    void hacerRuidoFuerte();  
}
```

- Las interfaces no son parte de la jerarquía de clases

Generics

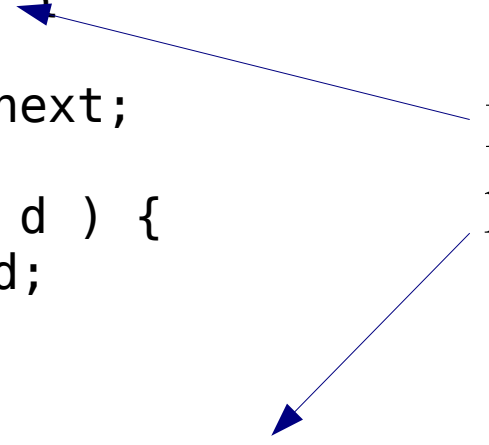
- Proveen abstracción sobre tipos
 - Son clases, interfaces o métodos que son parametrizados por tipos
- Me permiten lograr implementaciones independiente del tipo de dato “base” a utilizarse
- Muy usado en Colecciones de objetos.

Generics

Ejemplo -Lista simplemente enlazada : definimos una lista de tipo genérico

```
public class Nodo<E> {  
    public E dato;  
    public Nodo<E> next;  
  
    public Nodo ( E d ) {  
        this.dato = d;  
    }  
}
```

Recibe como
parámetro un tipo



```
public class ListaEnlazada <E>  
    private Nodo<E> header = null;  
    public E agregar ( E dato ) {  
        Nodo<E> tmp = header;  
        header      = new Nodo<E>(dato);  
        header.next = tmp;  
    }  
}
```

Generics

```
/* Ejemplo de uso de nuestra ListaEnlazada* /  
public static void main ( String [] args){  
  
    ListaEnlazada<String> LS = new ListaEnlazada<String>();  
  
    LS.agregar("Hola");  
    LS.agregar("Que tal");  
    LS.agregar("Como estan");  
}
```

Generics

Métodos genéricos

Ejemplo de un método genérico que encuentra el máximo en un arreglo de tipo genérico pero con cierto tipo de restricciones.

```
public static <T extends Comparable <? super T>> T findMax( T[]a )
{
    int maxIndex = 0;
    for( int i = 1; i < a.length; i++ )
        if( a[ i ].compareTo( a[ maxIndex ] ) > 0 )
            maxIndex = i;
    return a[ maxIndex ];
}
```

Generics

```
public static void main( String [ ] args )
{
    String  [ ] sL = { "Joe", "Bob", "Bill", "Zeke" };
    Integer [ ] iL = { 10, 20, 30, 50};
    Double  [ ] fL = {10.20, 10.303, 23.22};

    System.out.println( findMax(sL) );
    System.out.println( findMax(iL) );
    System.out.println( findMax(fL) );
}
```


Ejemplos utilizando Generics

ArrayList

```
public class GenericSimpleArrayList<T>
{
    private static final int CAPACIDAD_POR_DEFECTO = 10;

    private int theSize;
    private T [ ] theItems;

    /**
     * Constructor por defecto
     */
    public GenericSimpleArrayList( )
    {
        clear( );
    }
    ...
}
```

Ejemplos utilizando Generics

ArrayList

```
/**
 * Retorna en el número de items en la colección
 */
public int size( )
{
    return theSize;
}

/**
 * Retorna el item en la posición idx
 */
public T get( int idx )
{
    if( idx < 0 || idx >= size( ) )
        throw new ArrayIndexOutOfBoundsException("Índice " +
idx + "; size " + size( ) );
    return theItems[ idx ];
}
```

Ejemplos utilizando Generics

ArrayList

```
..
/**
 * Cambiar el item de la posición idx. Retorna el valor
 * anterior
 */
public T set( int idx, T newVal )
{
    if ( idx < 0 || idx >= size() )
        throw new ArrayIndexOutOfBoundsException( "Indice "
+ idx + "; size " + size( ) );

    T old = theItems[ idx ];
    theItems[ idx ] = newVal;
    return old;
}
..
```

Ejemplos utilizando Generics

ArrayList

```
..  
/**  
 * Agregar un item a la colección, al final.  
 */  
public boolean add( T x )  
{  
    if (theItems.length == size() )  
    {  
        T [ ] old = theItems;  
        theItems = (T []) new Object[theItems.length * 2 + 1];  
        for( int i = 0; i < size(); i++ )  
            theItems[i] = old[ i ];  
    }  
  
    theItems[ theSize++ ] = x;  
  
    return true;  
}  
..
```

Ejemplos utilizando Generics

ArrayList

```
..  
/**  
 * Remueve el item de la colección, se desplazan  
 * los componentes.  
 */  
public T remove( int idx )  
{  
    T removedItem = theItems[ idx ];  
  
    for( int i = idx; i < size( ) - 1; i++ )  
        theItems[ i ] = theItems[ i + 1 ];  
  
    theSize--;  
  
    return removedItem;  
}  
..
```

Ejemplos utilizando Generics

ArrayList

```
..  
/**  
 * Inicializa la colección  
 */  
public void clear( )  
{  
    theSize = 0;  
    theItems = (T []) new Object[ CAPACIDAD_POR_DEFECTO];  
}  
  
} /* Fin de la clase
```

Ejemplos utilizando Generics

ArrayList

```
public class UsoGenericList {  
  
    public static void main ( String [] args ) {  
        GenericSimpleArrayList<String> sa = new  
            GenericSimpleArrayList<String>();  
        sa.add("Hola");  
        sa.add("Que tal");  
  
        for ( int i = 0 ; i <= sa.size() ; i++)  
            System.out.println(sa.get(i));  
    }  
}
```

Parte IX

Construcciones avanzadas en Java 8

Expresiones lambda

- Funciones que no están asociadas a un determinado nombre y pueden ser argumento de otras funciones.
- En vez de pasar datos ahora podemos pasar también código o cómputo.

Expresiones lambda (ejemplo)

```
public class Calculator {
    interface IntegerMath {
        int operation(int a, int b);
    }

    public int operateBinary(int a, int b, IntegerMath op) {
        return op.operation(a, b);
    }

    public static void main(String... args) {

        Calculator myApp          = new Calculator();

        IntegerMath addition      = (a, b) -> a + b;
        IntegerMath subtraction   = (a, b) -> a - b;

        System.out.println("40 + 2 = " +
                           myApp.operateBinary(40, 2, addition));

        System.out.println("20 - 10 = " +
                           myApp.operateBinary(20, 10, subtraction));
    }
}
```

```
public class Persona {  
    public enum Sexo {  
        MASC, FEM  
    }  
  
    private String nombre;  
    private LocalDate fecha_nac;  
    private Sexo genero;  
    private String emailAddress;  
  
    public int getEdad() { ... }  
    public Sex getSexo() { ... }  
    public String getCorreo() { ... }  
  
    ...  
}  
  
List<Persona> listaPersona = new List<>();
```

Operaciones agregadas

Tengo una lista de *Persona* en la variable `listaPersona`. *Persona* tiene atributos “Sexo”, “Fec. De Nac”, etc. De esta lista necesito imprimir las direcciones de mail de las personas masculinas entre 18 y 25 años.

```
listaPersona.stream()
    .filter(
        p -> p.getSexo() == Persona.Sexo.MASC
            && p.getEdad() >= 18
            && p.getEdad() <= 25)
    .map(p -> p.getCorreo())
    .forEach(email -> System.out.println(email));
```

Bibliografía

Existen muchos libros de Java, algunos en español:

- ♦ Flanagan, David. ***Java in a NutShell, 5th edition.*** O'Reilly. 2005. (*En biblioteca edición antigua en español*).
- ♦ Deitel H. y Deitel P. ***Como programar en Java.*** Pearson Education. 2008 (*en Biblioteca*).
- ♦ Dean J y Dean R. ***Introducción a la programación con JAVA.*** 2009 (*en Biblioteca*)

Una buena fuente de inicio de uso del
lenguaje

The Java™ Tutorials

<http://download.oracle.com/javase/tutorial/index.html>

Especificación del lenguaje (versión 9):

<http://docs.oracle.com/javase/specs/jls/se9/html/index.html>