uc3m | Universidad Carlos III de Madrid

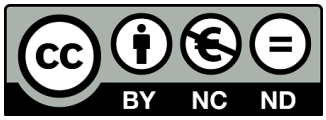
Grado en Ingeniería Informática

2018-2019

Apuntes

Programación

Jorge Rodríguez Fraile[1]



[1]Universidad: 100405951@alumnos.uc3m.es | Personal: jrf1616@gmail.com

ÍNDICE GENERAL

Parte I

Presentación de la Asignatura

Universidad
Carlos III de Madrid

Presentación de la asignatura

Programación

Grado en Ingeniería Informática

2018-2019

Profesores

- Teoría y Prácticas
  - Juan Luis Vicente Carro.
    Despacho: 1.2.B17
    juvicent@inf.uc3m.es

Comunicación

- Tutorías: Horario publicado en Aula Global (AG)
  - Mejor concertar cita previamente (en clase o vía e-mail)
  - Viernes:19 a 20
- Las dudas se atienden en el foro de Aula Global
  - (NO a través de correos privados)
- Web de la asignatura: Aula Global
  - Ejercicios semanales
  - Prácticas
  - Material complementario

Temario

- Tema 1: Introducción a la asignatura
- Tema 2: Diagramas de Flujo
- Tema 3: Introducción a Java
- Tema 4: Control de flujo: condicionales y bucles
- Tema 5: Estructuras de datos sencillas
- Tema 6: Reutilización de código
- Tema 7: Clases útiles
- Tema 8: Introducción a la Programación Orientada a Objetos (POO)
- Tema 9: Algoritmos

Evaluación contínua

- Preguntas en clase [0.7 puntos] (Test de realización sólo durante la asistencia a clase)
- Tests semanales [0.7 puntos](Disponibles en el Aula Global)
- Ejercicios semanales
  - Presentación oral [0.6 puntos] (Individual)
  - Entrega (80%) [0.5 puntos] (Grupos de 2)
- Examen parcial [1 punto] (Finales de octubre)
- Práctica final [1.5 puntos] (grupo de 2) (Entrega a principios de diciembre)
- Examen final [5 puntos] (enero)

Notas

- Nota en Enero:
  - max (evaluación continua, examen final sobre 10 * 0.6)

- Nota en Junio:
  - Se mantiene la nota de la evaluación continua. No se entregan ejercicios ni prácticas adicionales
  - El 0.6 en la fórmula cambia a 1.0

- **Para aprobar la asignatura hay que obtener al menos:**
  - **un 4 en el examen final**
  - **y un 5 en total.**

Exámenes

- Examen parcial:
  - 1’5 horas
  - Fecha: 26 de octubre
  - No se permite material: libros, apuntes, etc.

- Examen final:
  - Fecha: tercera semana de Enero
  - Se permite material: libros, apuntes, etc.
  - No se permiten dispositivos electrónicos

Trabajo semanal

- Es necesario trabajar en casa
  - **¡Las clases no son suficientes!**
- Tests al final de las clases
- Tests en Aula Global
  - Individual
  - Hay que rellenarlos antes del jueves a las 17:00
- Ejercicios de laboratorio
  - Grupos de dos
  - Hay que entregarlos antes del viernes a las 19:00
  - Presentaciones orales individuales en los laboratorios

Clases Prácticas

- Grupos de dos estudiantes

- Tres partes
  - Presentaciones orales de los ejercicios de la semana anterior: estudiantes aleatorios
  - Solución guiada de un ejercicio
  - Conjunto de ejercicios a entregar la semana siguiente
    - Hay que entregar al menos el 80% para obtener nota
    - Hay que terminarlos en casa

Práctica final

- Grupos de dos estudiantes
- Disponible desde el 31 de Octubre
- Entregas (en principio)
  - Entrega parcial: 17 de Noviembre a las 17:00
  - Entrega parcial: 30 de Noviembre a las 17:00
  - Entrega Final: 21 de Diciembre
- **Las copias serán duramente penalizadas**: los grupos implicados, tanto si copiaron como si les copiaron, serán directamente excluidos de la evaluación continua.

Bibliografía Básica

- David Camacho, José Ma Valls, Jesús García, José M. Molina, Enrique Bueno.*Programación, algoritmos y ejercicios resueltos en Java* Pearson/Prentice Hall, 2003.

- Bruce Eckel *Piensa en Java Thinking in Java*. Pearson Educación. Versión española: 2002. 2a Edición. Versión inglesa: 2007. 4a Edición

- Alfonso Jiménez Marín, Francisco Manuel Pérez Montes *Aprende a Programar con Java (Un enfoque práctico partiendo de cero)*. Paraninfo. 2012

Consejos útiles para superar la asignatura

- **No intentes aprender a programar solamente leyendo o viendo un vídeo .** (Nunca aprenderás a resolver el algoritmo y tendrás dificultades para desarrollar las capacidades necesarias)
- **Asegúrate que tu solución propuesta cumple con todo lo pedido y además es la solución más óptima.**(Nunca realices "apaños" o trampas para solucionar tu programa para unos pocos casos de prueba)
- **Resuelve problemas reales similares a las prácticas propuestas con ligeros cambios** (esto te ayudará a pensar y aprender)
- **Programa todos los días, todo requiere un tiempo necesario.**


VAMOOOOS
A PICAR CODIGO

Parte II

Introducción a la Programación

Programación

PLG
Planning and Learning Group

Universidad Carlos III de Madrid

Tema 1: Introducción a la Programación

¿Qué es Programar?

- Según el diccionario de la RAE:
  Inform. Elaborar programas para la resolución de problemas mediante ordenadores
- Otra definición informal pero más explicativa:
  Proporcionar a un ordenador un conjunto de datos y unas instrucciones sobre lo que se debe hacer con esos datos con el objetivo de resolver algún problema

¿Qué es Programar?

- **Objetivo**: Resolver un problema
- **Cómo resolverlo**: Utilizando un conjunto de instrucciones que le indican al ordenador paso a paso cómo resolver el problema

Algoritmo

Lista bien definida, ordenada y finita de operaciones que permite hallar la solución a un problema

Ejemplo de Algoritmo

- Algoritmo para cambiar la rueda de un coche
- Datos: rueda pinchada, y ubicación del gato, de la rueda de repuesto y de la llave inglesa

Ejemplo de Algoritmo

- Algoritmo para cambiar la rueda de un coche
- Datos: rueda pinchada, y ubicación del gato, de la rueda de repuesto y de la llave inglesa
  - PASO 1. Aflojar los tornillos de la rueda pinchada con la llave inglesa
  - PASO 2. Colocar el gato mecánico en su sitio
  - PASO 3. Levantar el gato hasta que la rueda pinchada pueda girar libremente
  - PASO 4. Quitar los tornillos
  - PASO 5. Quitar la rueda pinchada
  - PASO 6. Poner rueda de repuesto
  - PASO 7. Poner los tornillos y apretarlos ligeramente
  - PASO 8. Bajar el gato hasta que se pueda liberar
  - PASO 9. Sacar el gato de su sitio
  - PASO 10. Apretar los tornillos con la llave inglesa

Resolución de Problemas

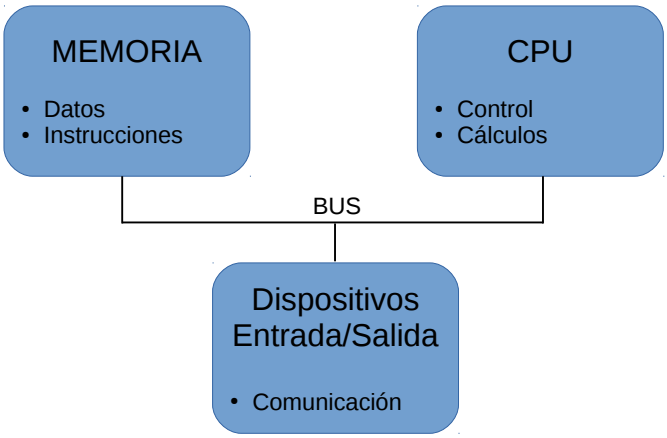
- El objetivo cuando se escribe un programa es resolver un problema
- Pasos generales para resolver problemas
  - Entender el problema
  - Si el problema es grande partirlo en piezas manejables
  - Diseñar una solución (un algoritmo)
  - Implementar la solución
  - Probar la solución y reparar errores

Buen algoritmo

- Preciso
- No ambigüo
- Correcto
- Eficiente
- Mantenible

Arquitectura Básica de un Ordenador

- Hardware vs. Software
- La inmensa mayoría de los ordenadores tiene una arquitectura de Von Neumann (propuesta inicialmente por Eckert y Mauchly).
  - Datos e instrucciones comparten memoria
  - Cada espacio (celda) de la memoria se identifica con un número llamado dirección



Componentes de la Arquitectura de un Ordenador

- Unidad Central de Procesamiento (CPU)
  - Se encarga fundamentalmente de ejecutar las instrucciones y coordinar el resto de elementos
- Memoria
  - Almacena los datos, las instrucciones y los resultados
  - Clasificación: principal/secundaria, permanente/volátil, acceso directo/secuencial
- Dispositivos de Entrada/Salida
  - Para proporcionar los datos e instrucciones y recibir los resultados
- Bus de Datos
  - Para compartir la información entre los componentes anteriores

Tipos de Software

- Software de Sistema
  Proporciona control sobre el hardware y sirve de base a las aplicaciones
- Software de Aplicaciones
  Programas con finalidades específicas, resuelven un problema o familia de problemas determinados
  - Ofimática
  - Contabilidad
  - Diseño
  - Juegos

Lenguaje de Programación

- Programa = datos + instrucciones
- Para comunicarle al ordenador el programa se usa un lenguaje de programación
- Los ordenadores no entienden lenguaje natural
- ¿Cómo decimos al ordenador lo que tiene que hacer?:
  Escribiendo un programa en un lenguaje de programación determinado, para implementar ese algoritmo

Tipos de Lenguajes de Programación

- Lenguaje Binario o Código Máquina
  - Con 0 y 1
- Lenguaje de bajo nivel
  - Instrucciones básicas (mover datos, sumar, ...)
- Lenguajes de alto nivel
  - más cercanos al lenguaje natural
  - ... aunque tampoco demasiado

Código Máquina

- Único lenguaje que entiende el ordenador
- Datos e instrucciones se codifican mediante conjuntos de 0 y 1
- El más rápido: hablamos al ordenador en su propio lenguaje
- Muy propenso a errores, muy complicado

Representación de Datos e Instrucciones

- Información codificada en binario
- La memoria está compuesta por bits que sólo pueden valer 1 o 0
- Los bits se agrupan en bytes (8 bits)
- Cada celda de memoria contiene entre 1 y 8 bytes y almacena un dato, un resultado o una instrucción
- ejemplo:

0	0	1	0	1	1	1	1	1	→	95
1	0	0	0	0	0	1	1	1	→	7
2	1	0	0	1	1	0	1	0	→	sumar
3	0	1	1	0	0	1	1	0	→	102

Lenguajes de Bajo Nivel

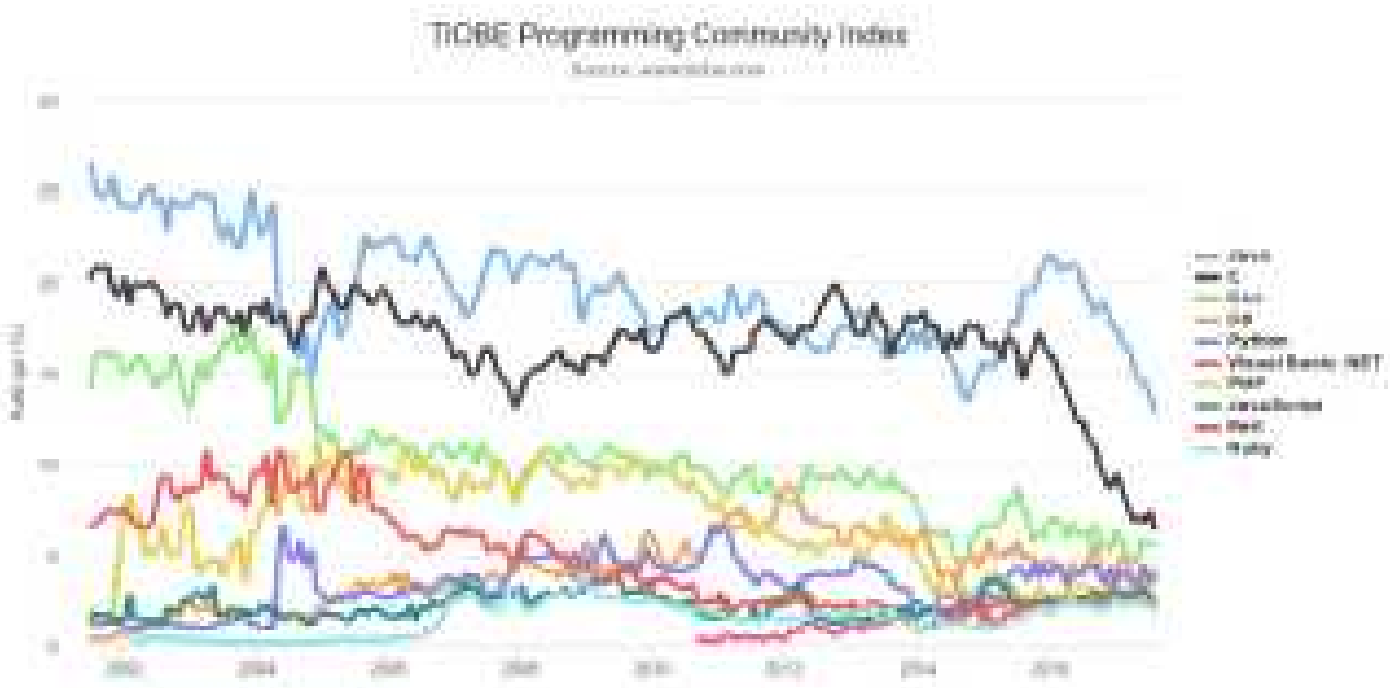
- Se usa un traductor para convertir desde un lenguaje textual a código máquina
- El traductor es un programa que le dice al ordenador cómo realizar la traducción
- Nace el lenguaje ensamblador: cambiamos 0 y 1 por texto, pero sigue siendo poco intuitivo
- Depende totalmente del tipo de procesador

Lenguajes de Alto Nivel

- Intentan acercar el lenguaje de programación al lenguaje humano
- Luego el ordenador se encargará de traducir
- Ideal: poder usar lenguaje natural
- Existen más de 300 (unos 2400 con dialectos)
- Los pioneros incluían conceptos como:
  - Variables: no es necesario gestionar los datos directamente en la memoria
  - Estructuras de datos complejas
  - Nuevas instrucciones, distintas de las que proporciona el ordenador
- Historia:

  http://manuelpereiragonzalez.blogspot.com/2009/09/historia-de-la-informatica-los.html

Uso de Algunos Lenguajes



Compilación e Interpretación

La traducción desde un lenguaje de programación a binario se puede hacer de dos formas:

- Todo a la vez: compilador
  - Ejecución más rápida
- Instrucción a instrucción: intérprete
  - Ejecuta aunque haya errores en el código
  - Permite cambios “en caliente”

Paradigmas de Programación

Tres formas principales de darle instrucciones al ordenador

- Programación Imperativa
  - Se describen los pasos necesarios para solucionar el problema
  - Ejemplos: C, Java, Visual Basic
- Programación Funcional
  - Las instrucciones se dan mediante funciones “matemáticas” que transforman los datos
  - Ejemplos: LISP, Erlang, Haskell
- Programación Lógica
  - Se describe el problema pero no se dan instrucciones: se resuelve mediante inferencia lógica
  - Ejemplo: Prolog

Parte III

Diagramas de Flujo

uc3m | Universidad Carlos III de Madrid

Tema 2

Diagramas de Flujo

Programación

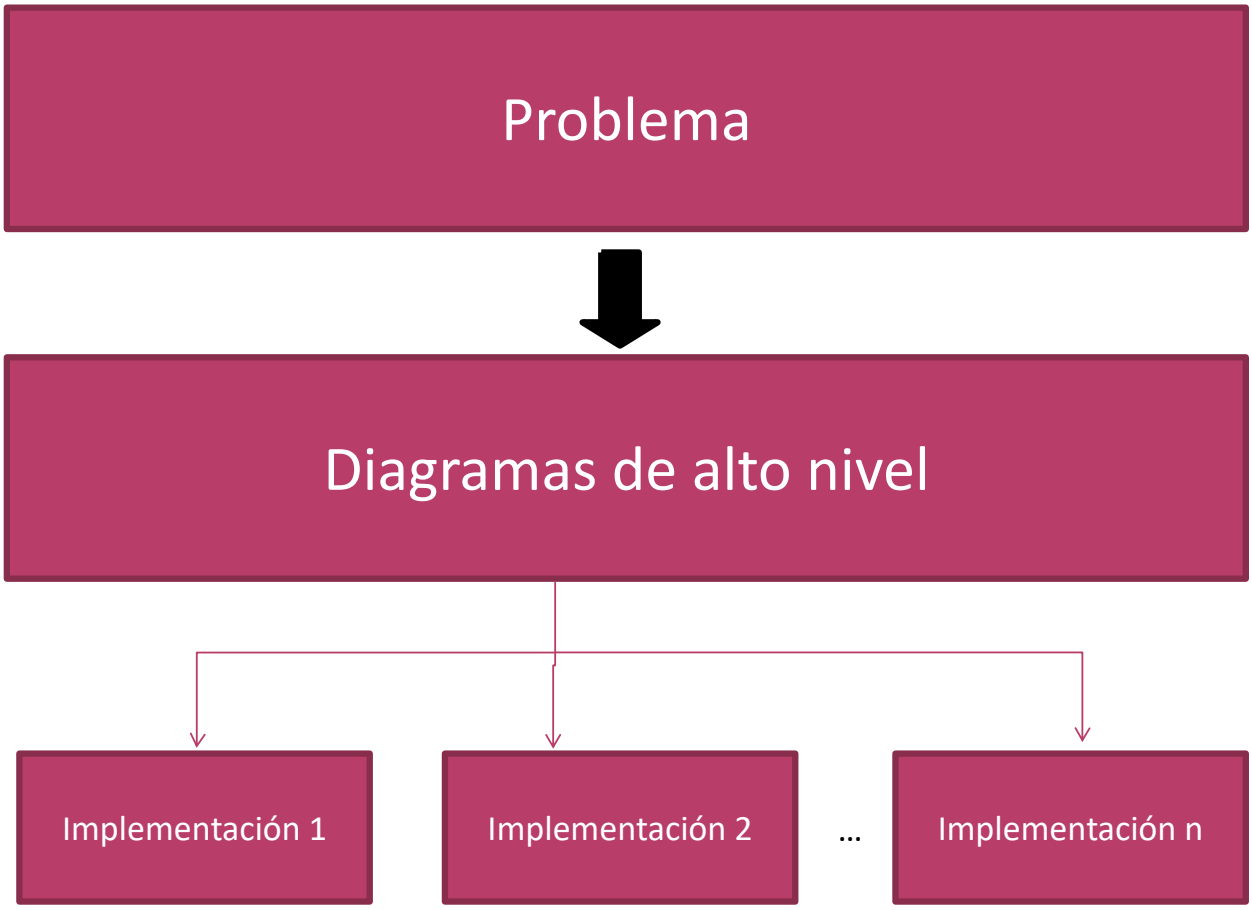
Grado en Ingeniería Informática

2017-2018

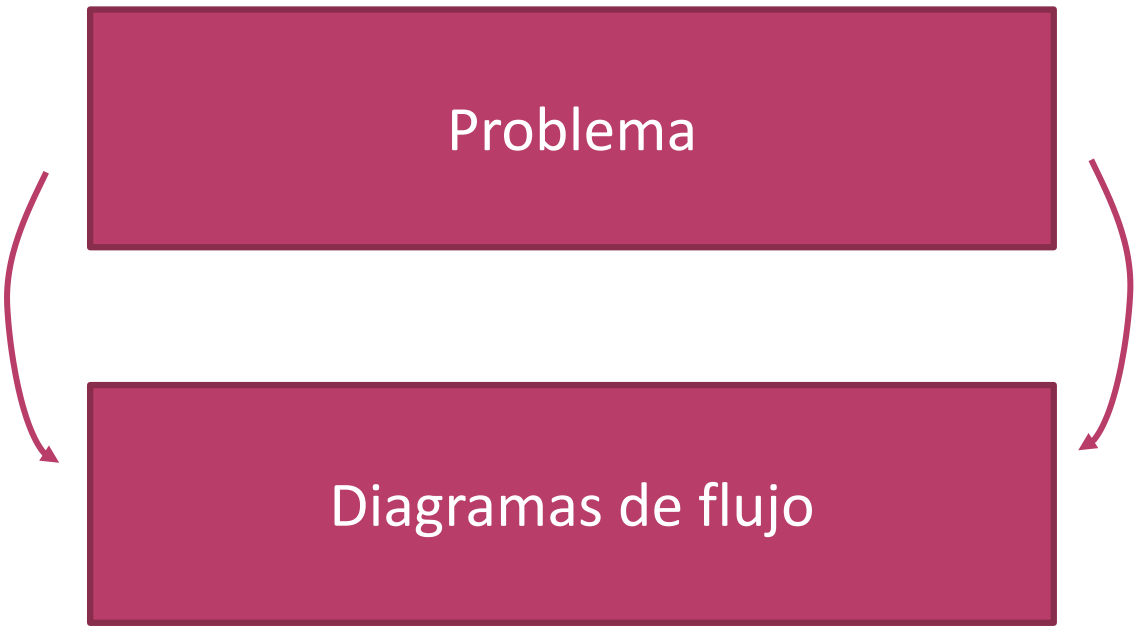
Agenda

- **Análisis de problemas**
- Variables
- Diagramas de flujo
- Bucles
- Resumen y Referencias

Resolución de problemas



Análisis



¿Qué necesito?

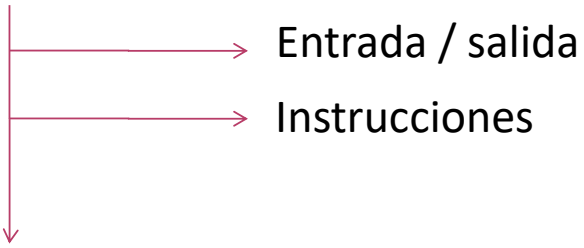
Entrada / salida

¿Qué necesito?

Entrada / salida

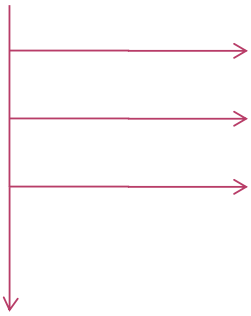
Instrucciones

¿Qué necesito?



Variables

¿Qué necesito?



Entrada / salida

Instrucciones

Variables

Condicionales

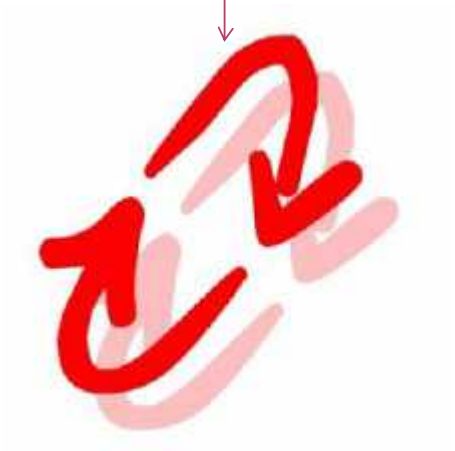
¿Qué necesito?

Entrada / salida

Instrucciones

Variables

Condicionales



Bucles

Agenda

- Análisis de problemas
- **Variables**
- Diagramas de flujo
- Bucles
- Resumen y Referencias

Variable

- Es un objeto cuyo valor puede ser modificado a lo largo de la ejecución de un programa.
- Una variable utiliza una porción de memoria que permite guardar valores.
- Una variable se caracteriza por:
  - Su nombre: es la forma de referirse a ellas y diferenciarlas de las demás.
  - Su tipo: es el tipo de datos que puede almacenar

Variable

- Podemos suponer que una variable es una caja donde se guarda el **valor que puede cambiar** en el tiempo.
- El **nombre** es la etiqueta en la caja.
- El **tipo** se identifica con la forma y el tamaño de la caja.

Asignación

- El operador de asignación es un operador binario que asigna (o coloca) el valor del operando de la derecha en el operando de la izquierda.

- El operando de la izquierda es obligatoriamente una variable.
- El operando de la derecha puede ser una variable o una expresión.

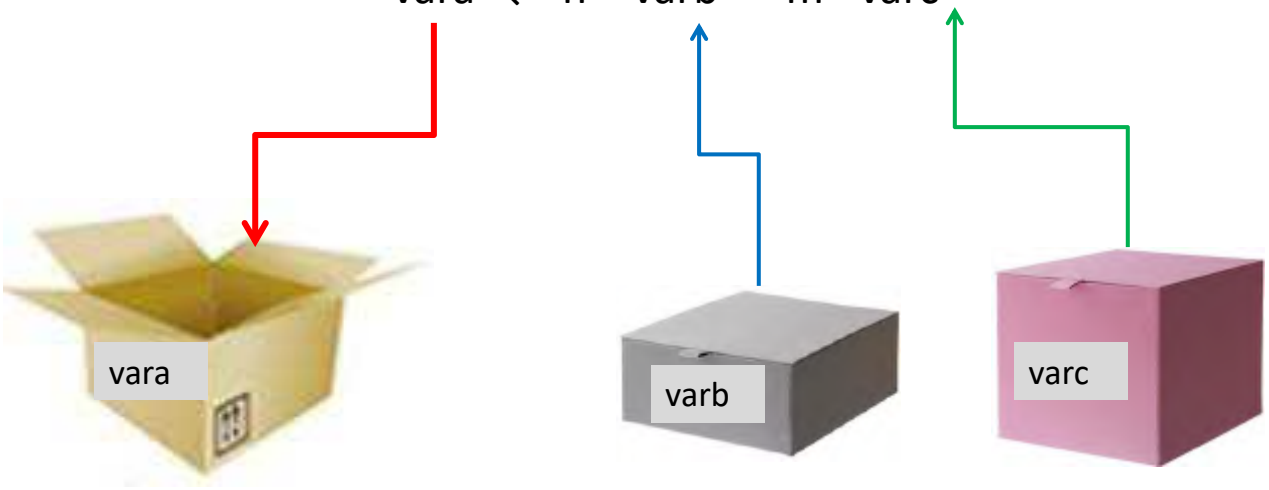
- En los diagramas de flujo se representa por una flecha de derecha a izquierda. “ ← “

- En el código se representa por un signo de igualdad. “ = “

Asignación

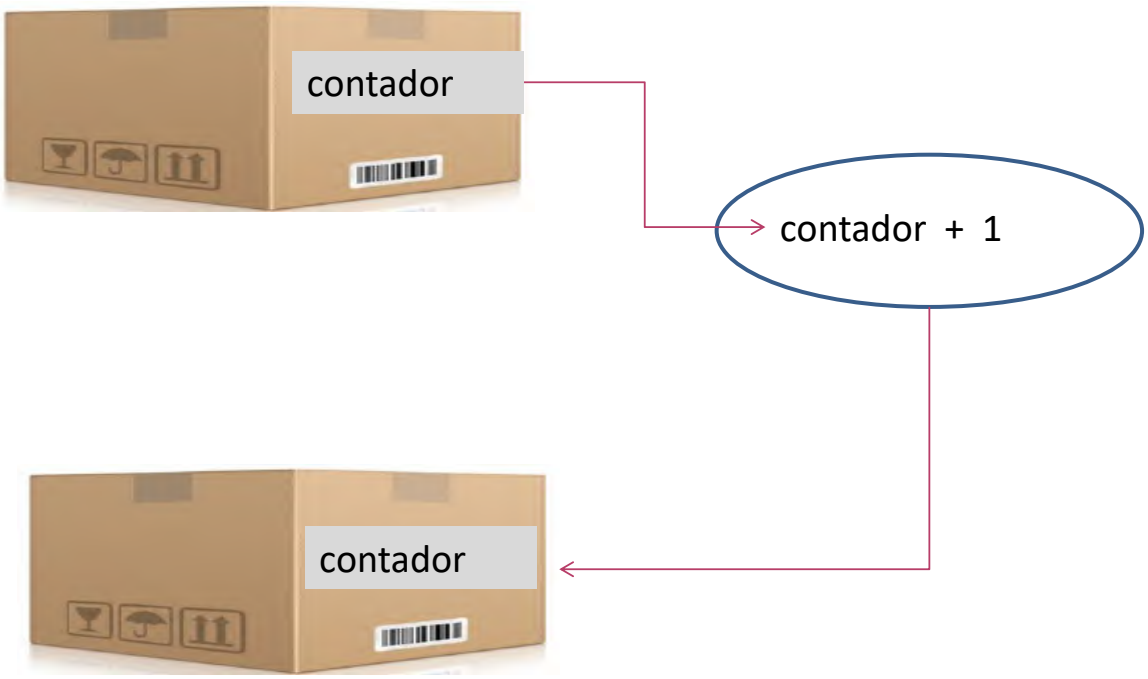
Variable ← expresión

vara ← n * varb + m* varc



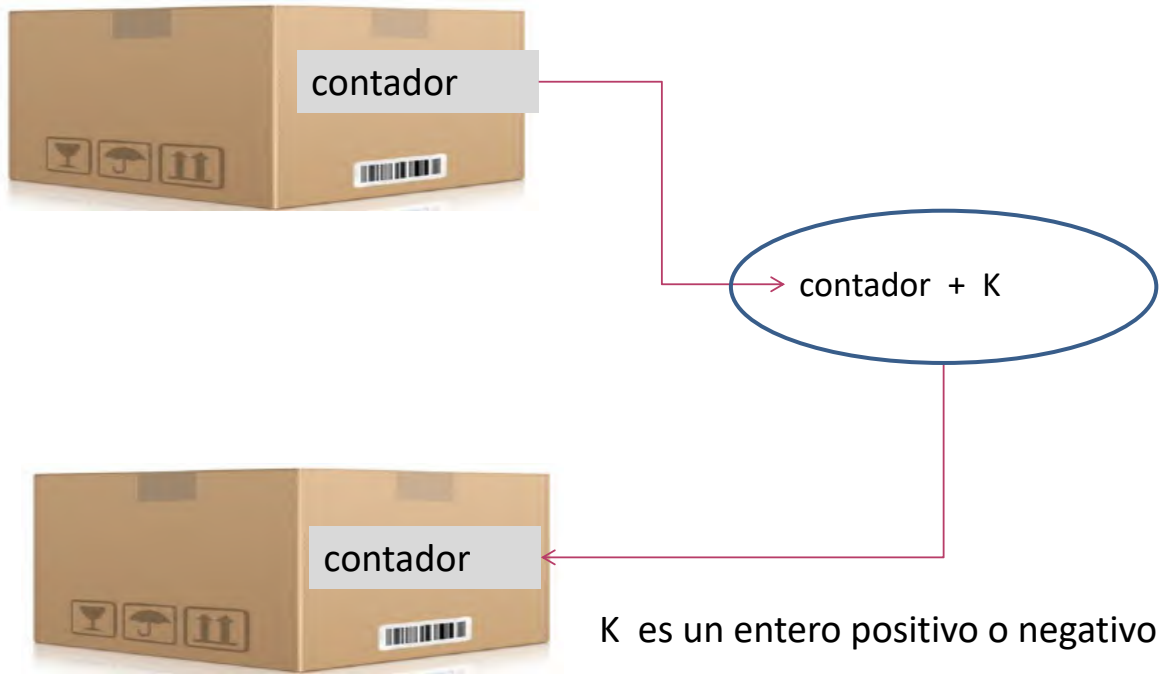
Variable contador

contador ← contador + 1



Variable contador general

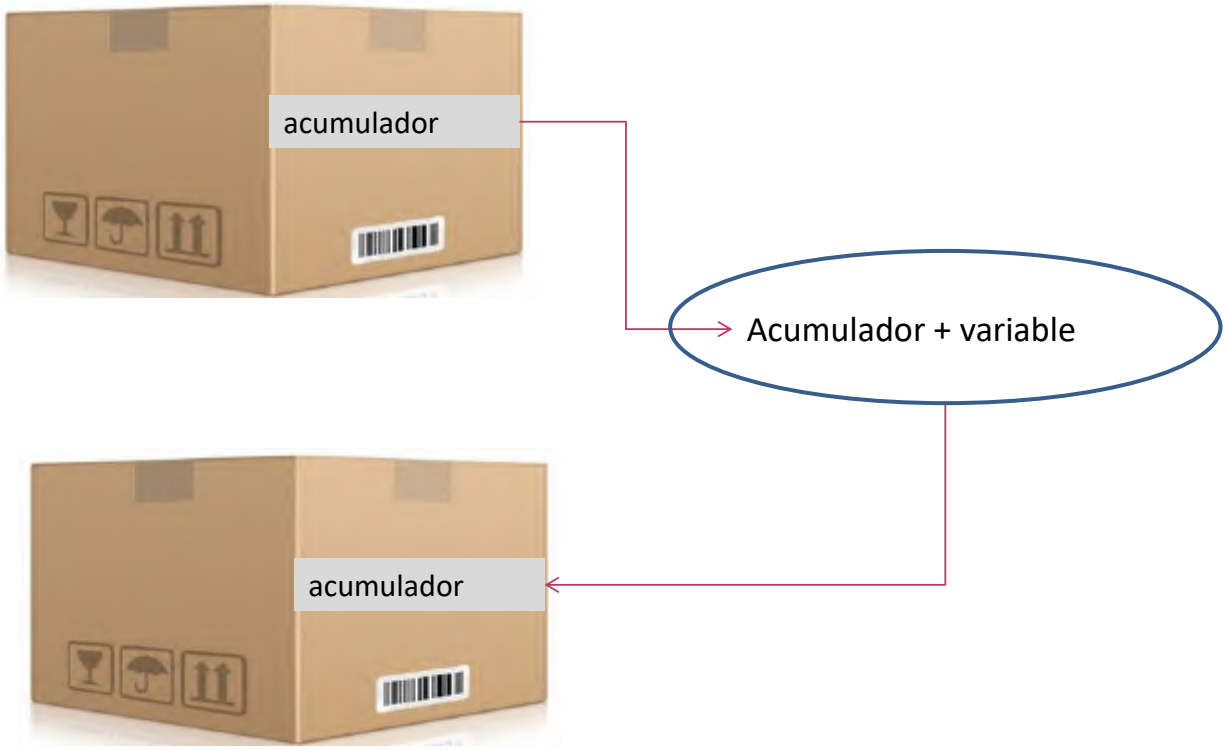
contador ← contador + K



K es un entero positivo o negativo

Variable acumulador

acumulador ← acumulador + variable

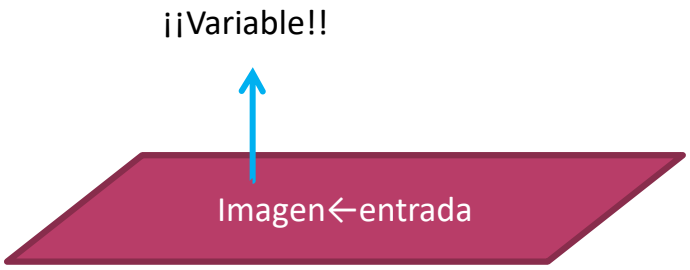


Agenda

- Análisis de problemas
- Variables
- **Diagramas de flujo**
- Bucles
- Resumen y Referencias

Diagramas de flujo

Entrada / Salida



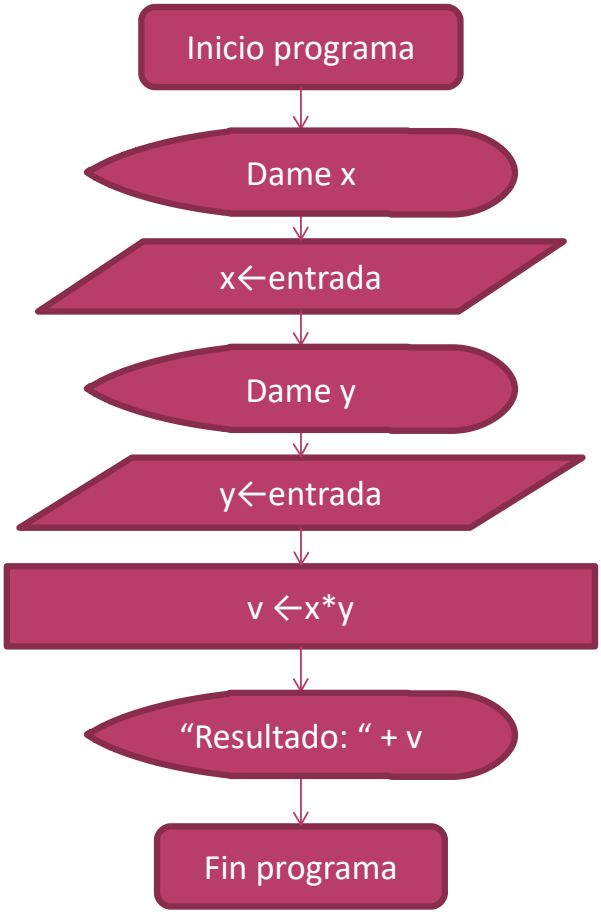
Instrucciones

3+5

¿¿¿???

Resultado ← 3+5

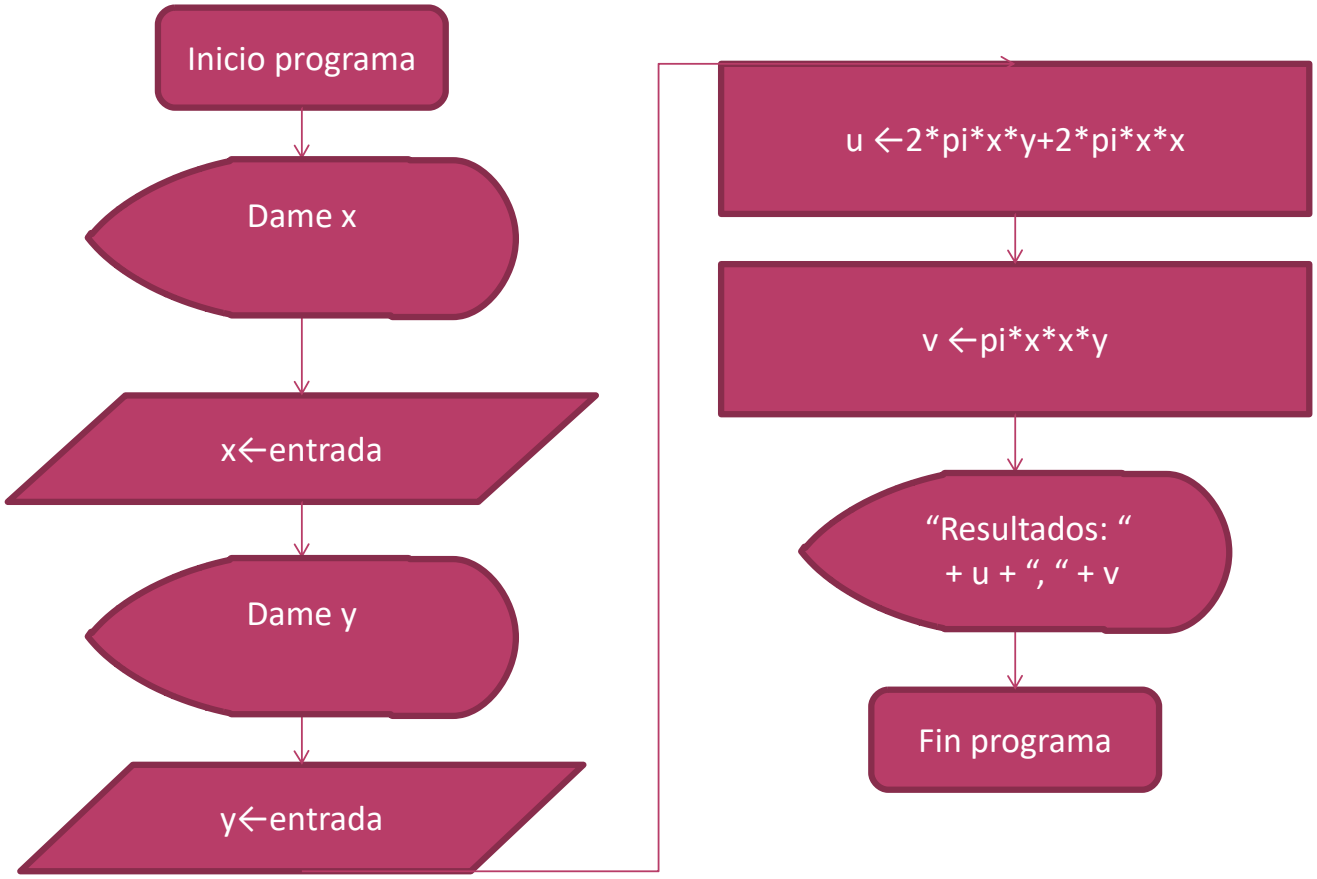
Estructura secuencial



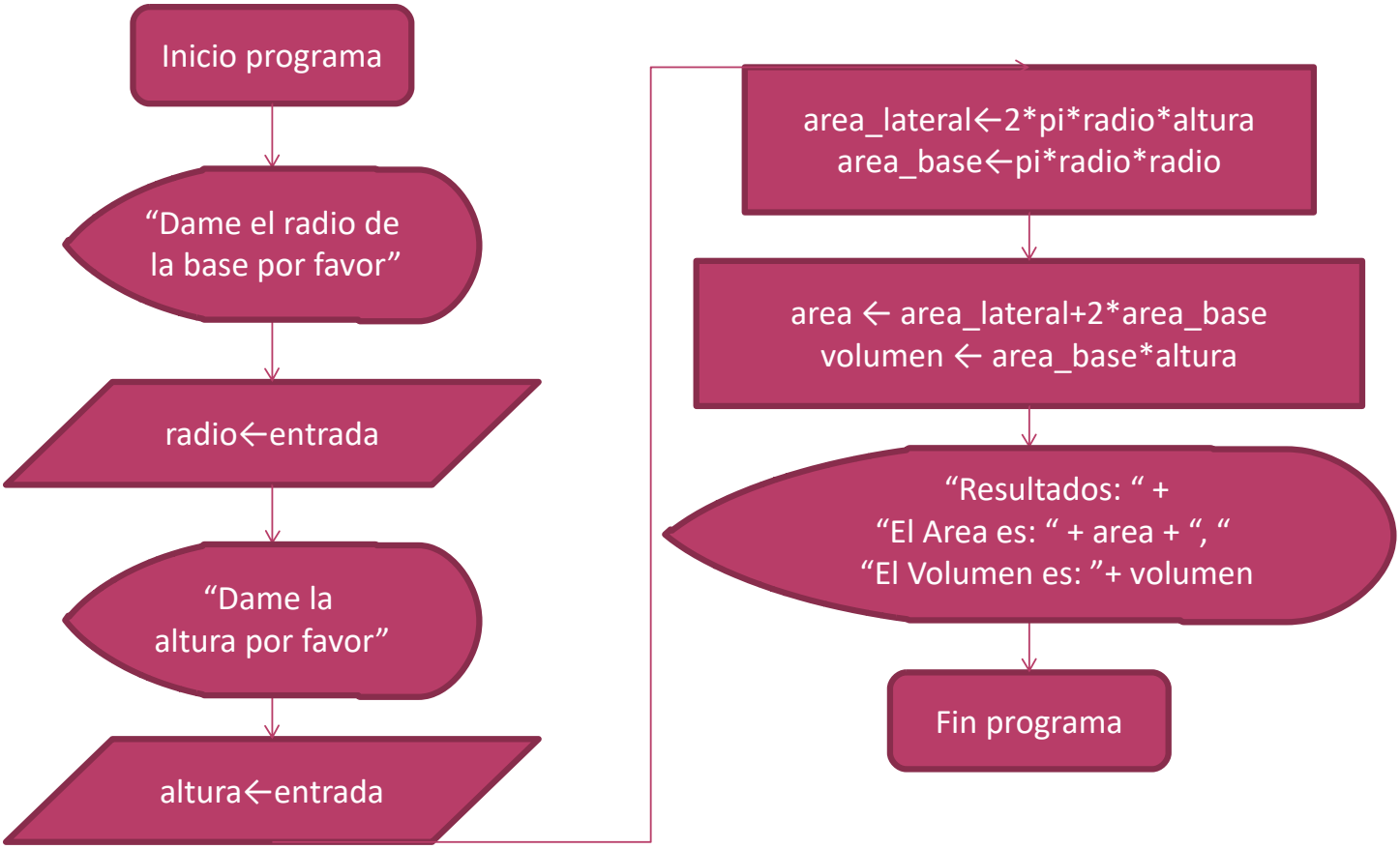
Variables

x,y, v

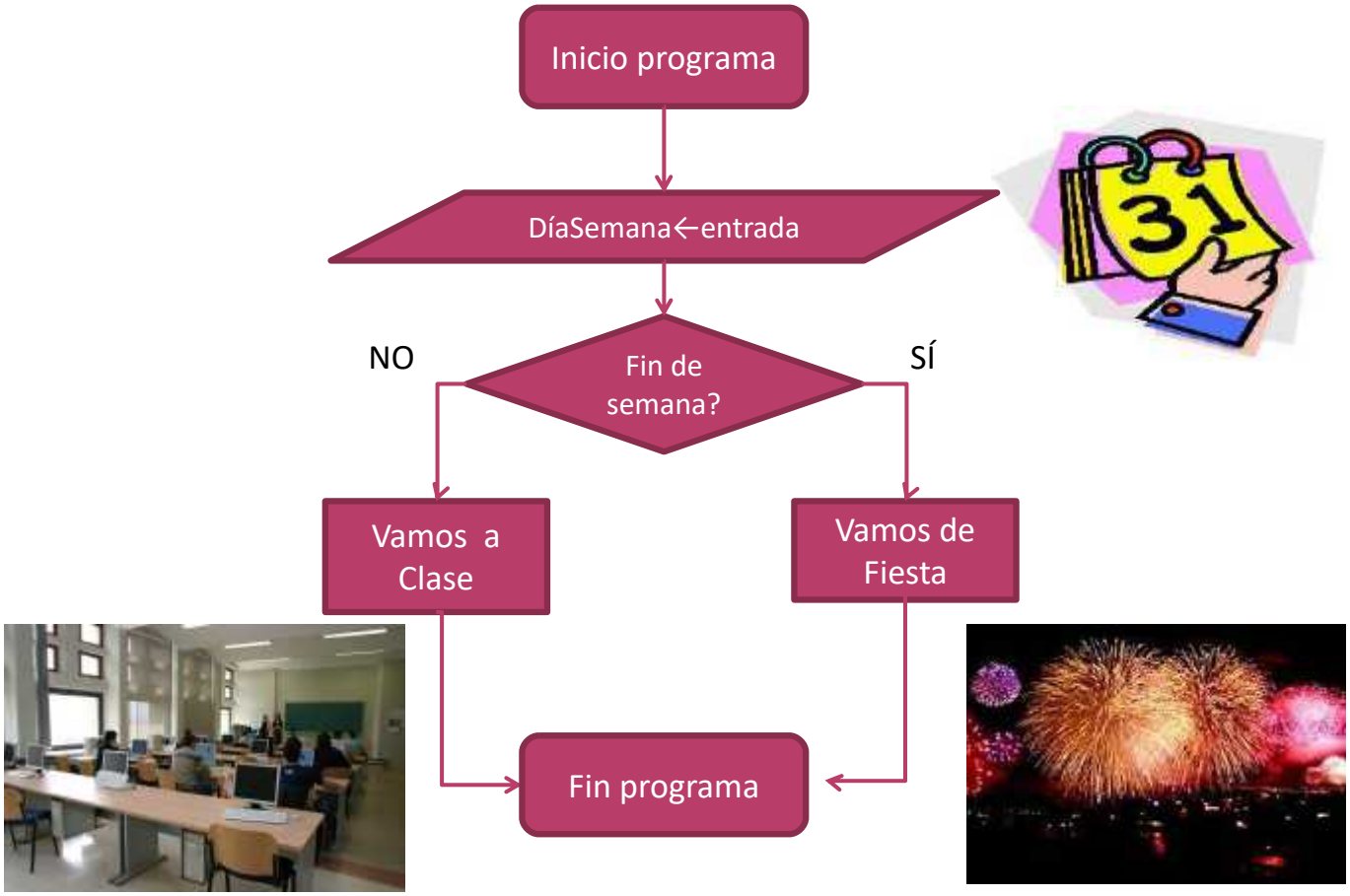
Variables


Inicio programa
Dame x
x←entrada
Dame y
y←entrada
u ←2*pi*x*y+2*pi*x*x
v ←pi*x*x*y
“Resultados: “
+ u + “, “ + v
Fin programa

Variables (II)



Condicionales



Agenda

- Análisis de problemas
- Variables
- Diagramas de flujo
- **Bucles**
- Resumen y Referencias

Bucles

Poner ladrillos para hacer una pared es un proceso repetitivo

Bucles

Antes de empezar…

¿Sé cuántos ladrillos necesito?

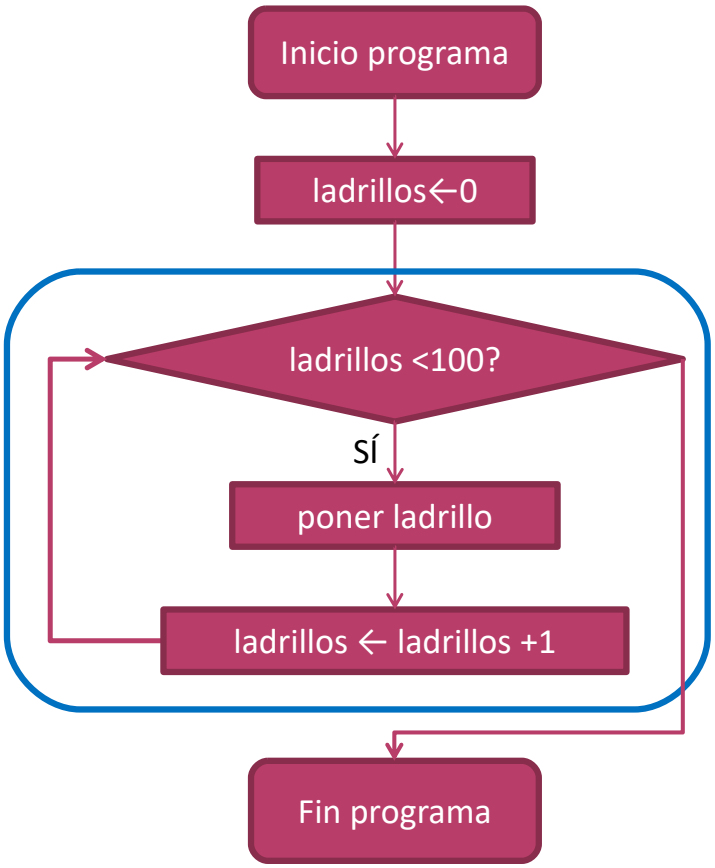
¿Voy a poner siempre al menos un ladrillo?

¿Tengo más ladrillos?

Sé cuántos voy a poner

Si tenemos 100 ladrillos

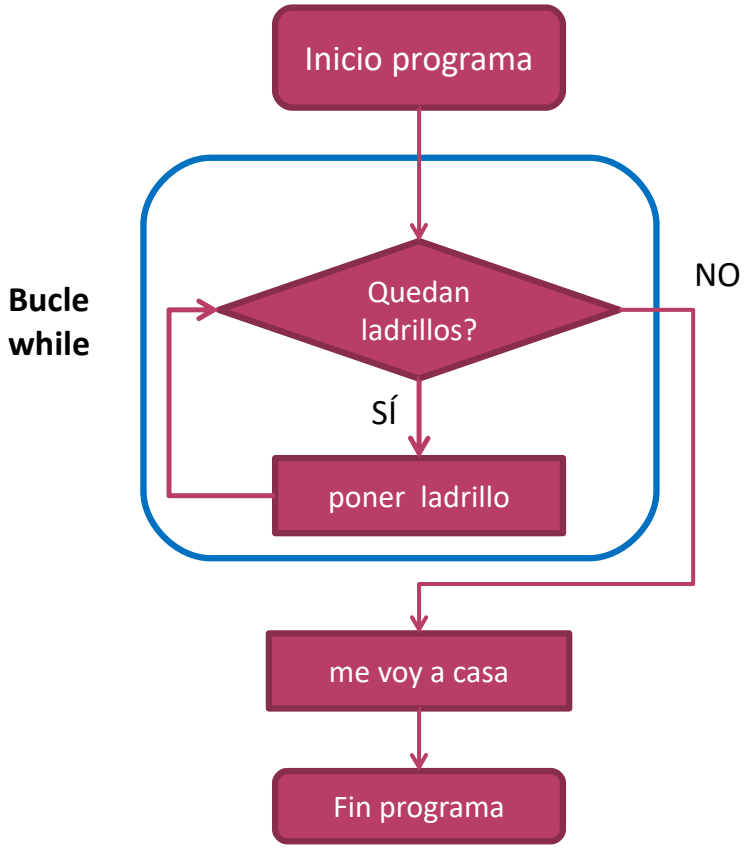
Bucle for



Pondré al menos uno

Bucle
do … while

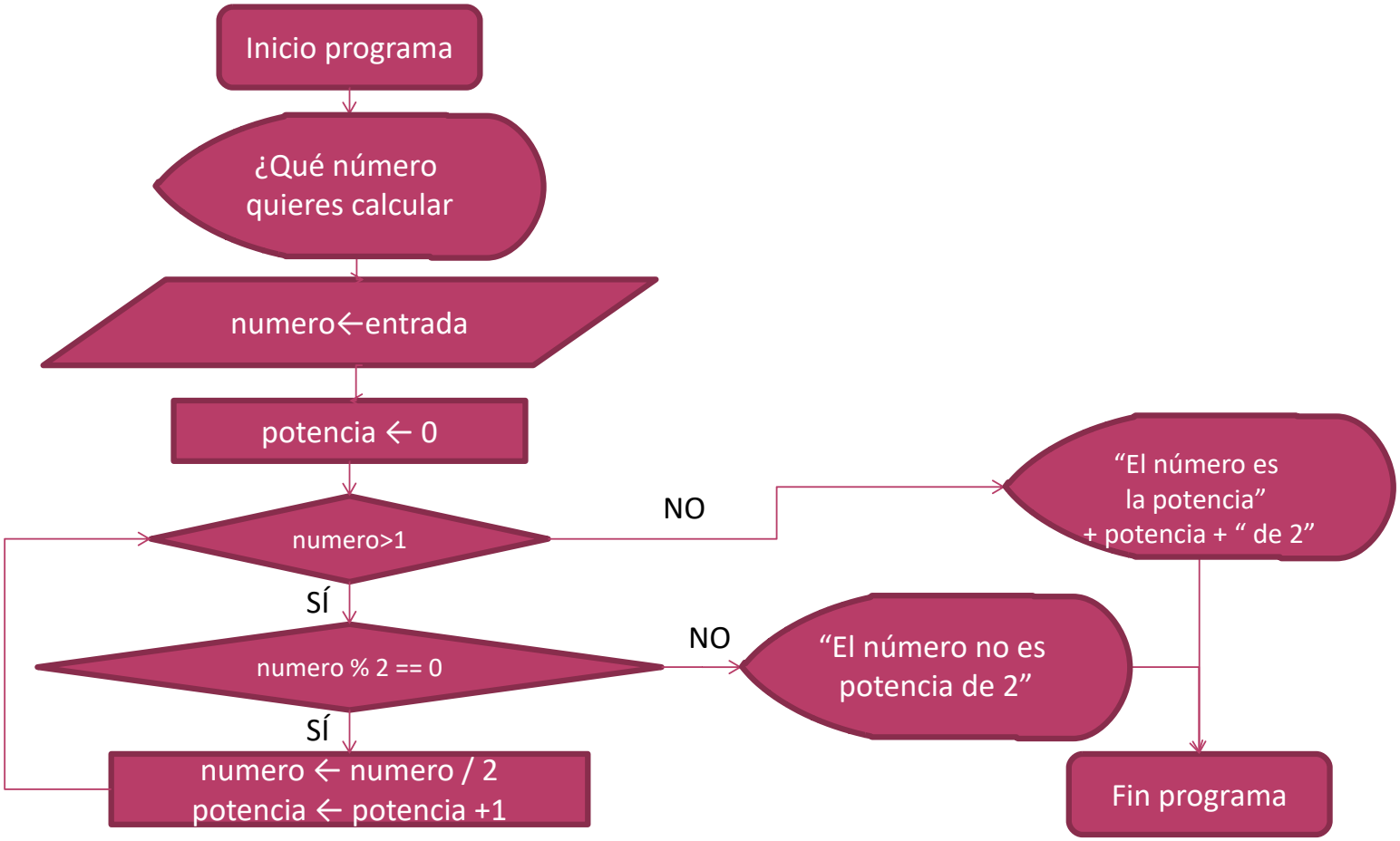
Puede que ya haya terminado…



Ejercicio

¿Qué potencia de dos es el número x?

Solución


Inicio programa
¿Qué número quieres calcular
numero←entrada
potencia ← 0
numero>1
NO
"El número es la potencia" + potencia + " de 2"
SÍ
numero % 2 == 0
NO
"El número no es potencia de 2"
SÍ
numero ← numero / 2
potencia ← potencia +1
Fin programa

Agenda

- Análisis de problemas
- Variables
- Diagramas de flujo
- Bucles
- **Resumen y Referencias**

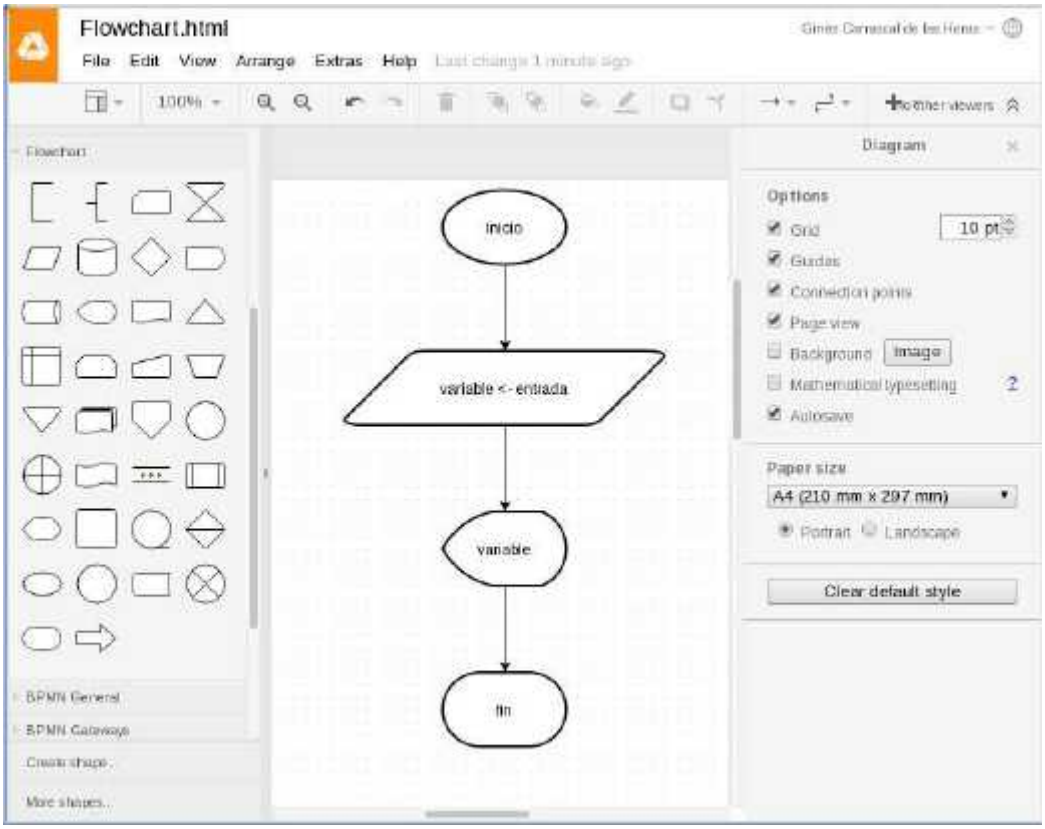
Resumen

- Análisis de problemas
  - Entrada / Salida
  - Instrucciones
  - Variables
  - Condiciones
  - Bucles

- Variables
  - Nombre
  - Tipo
  - Asignacion
  - Contadores y acumuladores
- Diagramas de flujo
  - Tipos de representación
  - Secuencia: inicio-fin
  - Condiciones
- Bucles
  - for
  - do...while
  - while

Herramienta para realizar diagramas

- https://www.draw.io
- Guardar y compartir diagramas en Drive
- Paleta “Flowchart”



Bibliografía y referencias web

- Diagramas de flujo:
  - https://es.wikipedia.org/wiki/Diagrama_de_flujo
- Algoritmos y diagramas:
  - http://www.aprenderaprogramar.com/index.php?option=com_content&view=article&id=213:conceptos-de-algoritmos-pseudocodigo-y-diagramas-de-flujo-una-introduccion-cu00123a&catid=28:curso-bases-programacion-nivel-i&Itemid=59
  -
- Usad vuestra imaginación:
  - http://www.google.com/search?q=flowcharting
  -

Parte IV

Introducción a Java

Programación

PLG
Planning and Learning Group

Universidad Carlos III de Madrid

Tema 3: Introdución a Java

En este tema

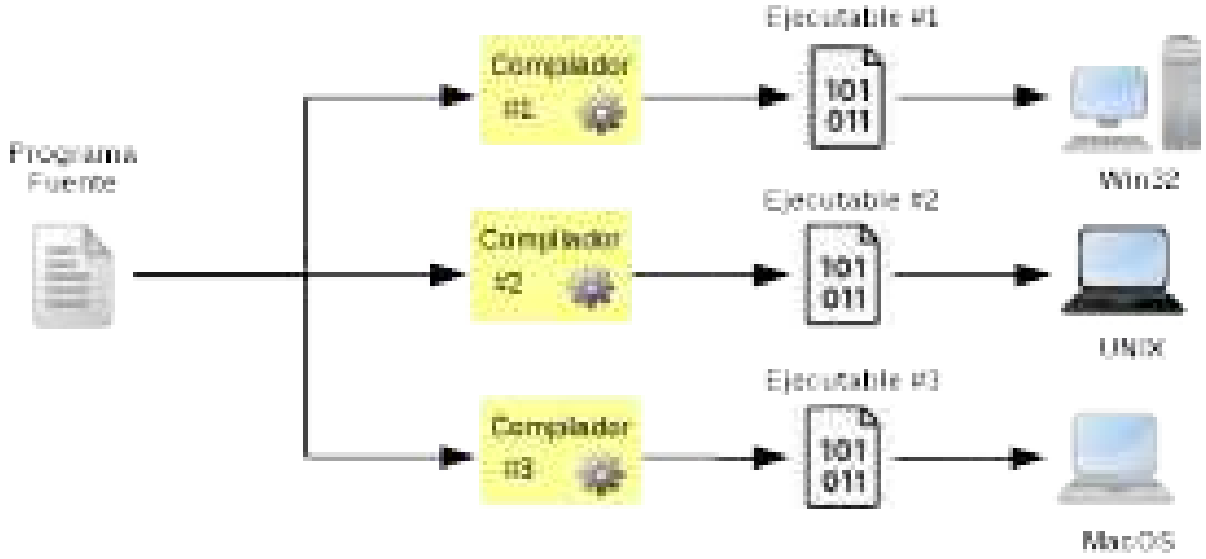
Tema 3: Introdución a Java

- La JVM (*Java Virtual Machine*)
- Tipos básicos o tipos primitivos
- Creación de programas en Java
- Variables y Constantes
- Operadores
- Otras cuestiones: datos, comentarios y errores

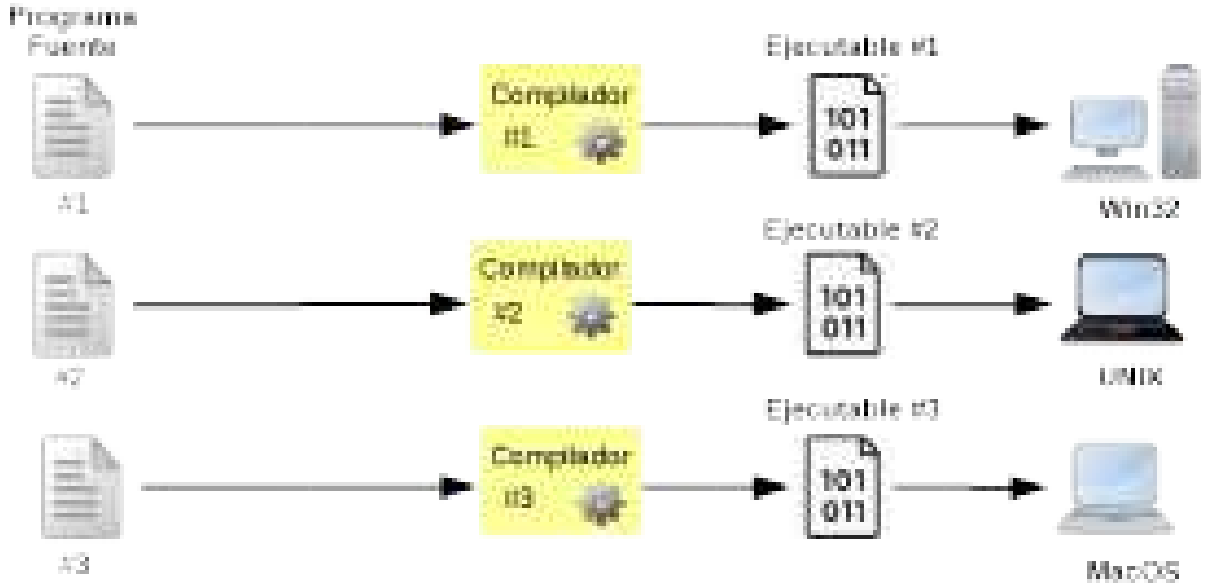
¿Qué es Java?

- Lenguaje de programación de alto nivel orientado a objetos
- Es también una plataforma de desarrollo
- 1991: Sun Microsystems diseña un lenguaje para sistemas embebidos, (set-top-boxes), electrodomésticos
  - Lenguaje sencillo, pequeño, neutro
  - Necesidad de un nuevo lenguaje: orientado a objetos, multiplataforma
  - Incialmente ninguna empresa muestra interés por el lenguaje
- Java: tipo de café?

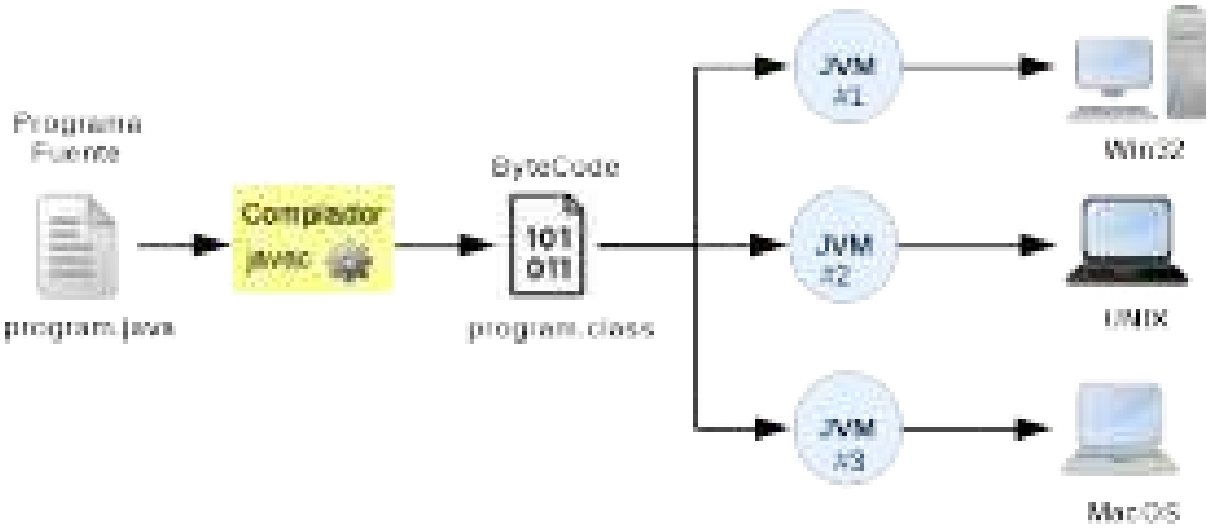
Historia de Java: otros lenguajes



Historia de Java: otros lenguajes



Historia de Java: Java



Historia de Java

- 1995: Java se presenta como lenguaje para Internet
- Netscape 2.0 introduce la primera JVM en un navegador web
- Filosofía Java: *Write once, run everywhere*
- 1997: Aparece Java 1.1. Muchas mejoras respecto a 1.0
- 1998: Java 1.2 (Java 2). Plataforma muy madura
- Apoyado por grandes empresas: IBM, Oracle, Inprise, Hewlett-Packard, Netscape, Sun
- 1999: Java Enterprise Edition. Revoluciona la programación en el lado servidor

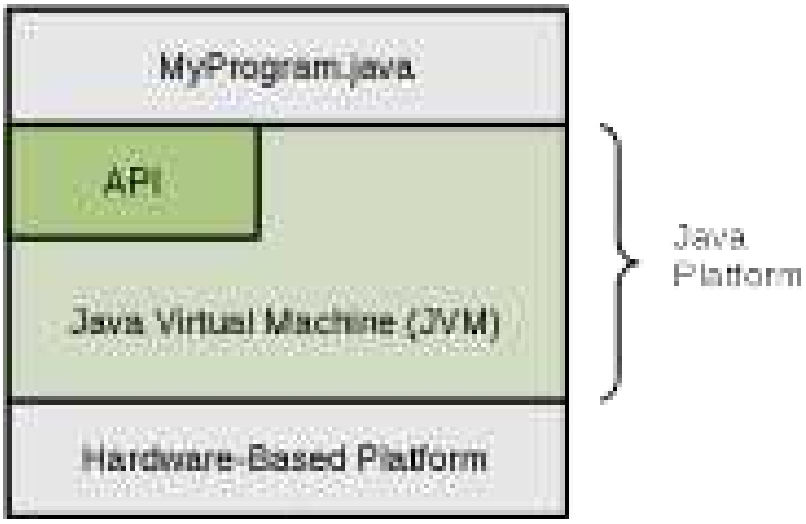
Características principales de Java

- Orientado a Objetos
- Totalmente Portable
- Lenguaje Interpretado (compilado a código intermedio, no a código máquina)
  - Java Virtual Machine (JVM)
  - ByteCode: Independiente de la máquina
- Gestión Automática de Memoria Dinámica. Recolector de basura (Garbage Collector)
- Sensible a Mayúsculas / Minúsculas
- Distribuido
- ¿Seguro?
- ¿Lento?

Versiones

- 1.0 (1996) – 1.1 (1997)- 1.2 (Java2) (1998) – 1.3 (2000) -1.4 (2002) – 1.5 (Java5.0) (2004) – Java 6 (2006) – Java 7 (2011) – Java 8 (Marzo-2014)
- Múltiples especificaciones
  - J8ME (Java 8 Micro Edition)
  - J8SE (Java 8 Standard Edition)
  - J8EE (Java 8 Enterprise Edition)

Plataforma Java



JDK (Java Development Kit)

- Compilador: javac
- Intérprete: java
- Plataforma de ejecución: JRE (Java Runtime Environment). Incluye JVM
- Plataforma de desarrollo: Java JDK (Java Software Development Kit):
  - Incluye Compilador, etc.
  - Incluye JRE

Entornos RAD (Rapid Application Development) o IDE (Integrated Development Environment)

- Productividad
- Modelado visual
- Depuración
- Rapidez de desarrollo
- Eclipse, Netbeans, Jbuilder, Symantec Café, Oracle Jdeveloper, Sun Java Workshop, IBM VisualAge, ...
- Prácticas:
  - J8SE (Java8 Standard Edition). Gratuito: `http://www.java.com/download`
  - Eclipse
    - Gratuito: `http://www.eclipse.org`
    - Versiones para Windows, Linux, etc.

En este tema

Tema 3: Introdución a Java

- La JVM (*Java Virtual Machine*)
- Tipos básicos o tipos primitivos
- Creación de programas en Java
- Variables y Constantes
- Operadores
- Otras cuestiones: datos, comentarios y errores

Programa

Programa = datos + instrucciones

- Datos: `3.5`, `a`, `María`, ...
- Instrucciones: operan con los datos
  - Operadores: sumar, restar, etc.
  - Control de flujo: condicionales, bucles, etc.
  - Entrada/Salida (E/S): leer, escribir, imprimir datos

Variables

- Una variable es un espacio de memoria que se utiliza para guardar un dato
- Los datos son de un tipo
  - *3* es un número entero y *3.5* es un número real
  - *verdadero* es un valor lógico
  - *'a'* es un caracter
  - *María* es una palabra
- Una variable se define por un nombre y por el tipo del dato que almacena

Tipos básicos o primitivos en Java

- Números
  - Enteros: dependiendo del rango hay 4 subtipos
  - Reales: 2 subtipos
- Lógicos: 1 tipo
- Letras: 1 tipo

Java es un lenguaje fuertemente tipado: al declarar una variable hay que decir el tipo

Números enteros

	Tipo	Tamaño	Rango
Enteros	byte	8 bits	-128 a 127
	short	16 bits	-32.768 a 32.767
	int	32 bits	-2.147.483.648 a 2.147.483.647 ($\sim 2 * 10^9$)
	long	64 bits	-9.223.372.036.854.775.808L a 9.223.372.036.854.775.807L ($\sim 9 * 10^{18}$)

- Siempre con signo
- Rango independiente de la plataforma
- Por defecto son de tipo int
- Los *long* se escriben con una “L” al final: 989493849859L, -284829848L

Números Reales (coma flotante)

	Tipo	Tamaño	Rango
Reales	float	32 bits (1 signo, 8 exponente, 23 mantisa)	± 1.4E-45F ± 3,40282347E+38F
	double	64 bits (1 signo, 11 exponente, 52 mantisa)	±4.9E-324 ±1,79769313486231570E+308

- Por defecto son de tipo double, pero *float* es más rápido y ocupa menos memoria
- Los *float* se escriben con una “F” al final
  - 3.45E+21F
  - -284829848F
- Rango mayor que enteros. *double*: 15-16 cifras, *float*: 8-9 cifras. Más allá se trunca.
- Para notación científica se puede poner 22e-5 o 22E-5 (e$< x >$=$10^{<x>}$ por lo que 1e2=100)

Tipos lógicos

- Tipo: boolean
  - *true*
  - *false*

Caracteres

- Tipo: char (16 bits – 65536 caracteres)
  - Entre comillas simples: 'a'
  - En memoria se codifican numéricamente: *Unicode* ('a' = 97)
  - Caracteres especiales (secuencias de escape)

Caracteres especiales

Secuencia	Descripción
\b	Retroceso *(backspace)*
\t	Tabulador *(tab)*
\r	Retorno de carro *(return)*
\n	Nueva Línea *(line feed)*
\’	Comilla simple *(single quote)*
\”	Comilla doble (la del 2) *(double quote)*
\\	Barra invertida *(backslash)*

Cadenas de caracteres

- Son secuencias de caracteres o palabras
- NO son un tipo básico, se utiliza la clase String
- Pero se pueden tratar como datos de tipo básico
- Se denotan entre comillas: “hola”

En este tema

Tema 3: Introdución a Java

- La JVM (*Java Virtual Machine*)
- Tipos básicos o tipos primitivos
- Creación de programas en Java
- Variables y Constantes
- Operadores
- Otras cuestiones: datos, comentarios y errores

Nuestro Primer Programa en Java

- Programas en Java
  - ficheros de texto con extensión .java (ej. `Hola.java`)
  - Hay que crear una clase y guardarla en el fichero. Una clase es un programa, y un programa es una clase.

Nuestro Primer Programa en Java

```
public class Hola
{
    public static void main(String[] args)
    {
        System.out.println("Mi primer programa en Java");
    }
}
```

Nuestro Primer Programa en Java

- El código va dentro del método main
- Primero los datos, luego las instrucciones.
- Las instrucciones terminan en “;”

En este tema

Tema 3: Introdución a Java

- La JVM (*Java Virtual Machine*)
- Tipos básicos o tipos primitivos
- Creación de programas en Java
- Variables y Constantes
- Operadores
- Otras cuestiones: datos, comentarios y errores

Nombres e identificadores válidos

- Un identificador sirve para nombrar ficheros, variables, constantes, etc.
  - Un identificador empieza por _, $ o una letra.
  - y luego continua con _, o una letra o un número.
  - Ejemplos: `$var1, _var2, Variable`
- Sensible a las diferencias de mayúsculas y minúsculas
- No se pueden utilizar palabras reservadas
  class, public, static, int, float, true, . . .

Convenciones para identificadores

- Conjunto de recomendaciones para facilitar la compresión del código.
- CamelCase convención que escribe un conjunto de palabras unidas sin espacio y con la letra inicial de cada palabra en mayúscula.
  - Upper CamelCase: la primera letra en mayúscula
    `Num1, MiVariableContador`
  - Lower CamelCase: la primera letra en minúsculas
    `num1, miVariableContador`
- Los nombres de clases deben escribirse en Upper CamelCase
- Los nombres de variables deben escribirse en lower CamelCase

Uso de variables (Almacenamiento en Memoria)

- Declarar la variable
  - Reservar memoria, declarando el tipo
  - Dar un nombre a la dicha posición de memoria (identificador de la variable)

```
int numero1;
```

Uso de variables (Almacenamiento en Memoria)

- Declarar la variable
  - Reservar memoria, declarando el tipo
  - Dar un nombre a la dicha posición de memoria (identificador de la variable)

```
int numero1;
```

- Inicializar la variable
  - Dar el valor inicial del dato
  - Java no da valores por defecto a las variables

```
numero1 = 3;
```

Uso de variables (Almacenamiento en Memoria)

- Declarar la variable
  - Reservar memoria, declarando el tipo
  - Dar un nombre a la dicha posición de memoria (identificador de la variable)

```
int numero1;
```

- Inicializar la variable
  - Dar el valor inicial del dato
  - Java no da valores por defecto a las variables

```
numero1 = 3;
```

- Ambos pasos a la vez

```
int numero1 = 3;
```

Uso de variables (Almacenamiento en Memoria)

- Declaración de varias variables a la vez
  - separando por comas
  - algunas pueden estar inicializadas

```
int numero1 = 3, numero2 = 1, i, j;
float x, y, real1 = 3.5f;
```

Uso de constantes

- Las constantes son variables especiales: una vez les hemos asignado el valor, éste no puede cambiarse
- Se declaran igual que las variables pero con la palabra `final` delante
- Por convención se escriben en mayúsculas. Si son varias palabras se separan con _

```
final float PI = 3.141593f;
final int ALTURA_MAXIMA = 100;
```

Salida por pantalla

- Para mostrar en la pantalla el contenido de variables

```
int n = 5, m = 6;
System.out.println(n);
System.out.print(m);
System.out.println(n + " ," + m);
```

- También se puede imprimir un dato directamente

```
System.out.println(10.5);
```

- O concatenar varias variables o datos

```
int n = 5, m = 6;
System.out.println("Valores: " + n + " ," + m);
```

En este tema

Tema 3: Introdución a Java

- La JVM (*Java Virtual Machine*)
- Tipos básicos o tipos primitivos
- Creación de programas en Java
- Variables y Constantes
- Operadores
- Otras cuestiones: datos, comentarios y errores

Instrucciones en Java

- Instrucciones
  - Operaciones
  - Control de flujo (más adelante)
  - Entrada/Salida
- Tipos de Operadores
  - de asignación
  - aritméticos
  - lógicos

Operador de Asignación

- El operador = lo utilizamos para copiar en una variable un dato, el contenido de otra variable o el resultado de una operación
- Visto en la inicialización de variables
- Nos permite cambiar los valores de las variables

```
int a = 4, b,c;
b = a;
a = 6;
```

Operadores Aritméticos

Operador	Descripción	int a=2, b=3, c	Valor en c
+	Suma	c = a + b	5
-	Resta	c = a - b	-1
*	Multiplicación	c = a * b	6
/	División	c = a / b	0
%	Módulo	c = a % b	3

Operadores Aritméticos

- El resultado de las operaciones con byte y short pasa a int automáticamente.
- Para el resto de tipos, el resultado de las operaciones es del mismo tipo

```
byte b1 = 5, b2 = 6;
int i1 = 3, i2 = 4, suma, producto;
double d1 = 3.0, d2 = 2, resultado;

suma = b1 + b2;
producto= i1 * i2;
resultado = d1/d2;

System.out.println(suma);
System.out.println(producto);
System.out.println(resultado);
```

Operadores de Autoincremento y Autodecremento

- Forma abreviada de sumar o restar 1 a una variable
  - ++ : suma uno a la variable
  - -- : resta uno a la variable
- Prefijos: operador antes de la variable
  - se hace el incremento (o decremento) y se genera el valor
- Postfijos: operador después de la variable
  - se genera el valor y luego se hace el incremento (o decremento)

```
int i=0, j=0, x, y;
i++;
++j;
x = i++;
y = ++j;
```

Operadores de Autoincremento y Autodecremento

- Forma abreviada de sumar o restar 1 a una variable
  - ++ : suma uno a la variable
  - -- : resta uno a la variable
- Prefijos: operador antes de la variable
  - se hace el incremento (o decremento) y se genera el valor
- Postfijos: operador después de la variable
  - se genera el valor y luego se hace el incremento (o decremento)

`int i=0, j=0, x, y;`	declaraciones
`i++;`	i vale 1
`++j;`	j vale 1
`x = i++;`	x vale 1, i vale 2
`y = ++j;`	y vale 2, j vale 2

Operaciones Aritméticas (char)

- Las operaciones artiméticas entre variables char pasan a int automáticamente.
- Los auto-incrementos o auto-decrementos con char mantienen el mismo tipo

```
char letra = 'a';
int letraNum;

letraNum = letra + 1;
letra++;
```

Operaciones Fuera de Rango

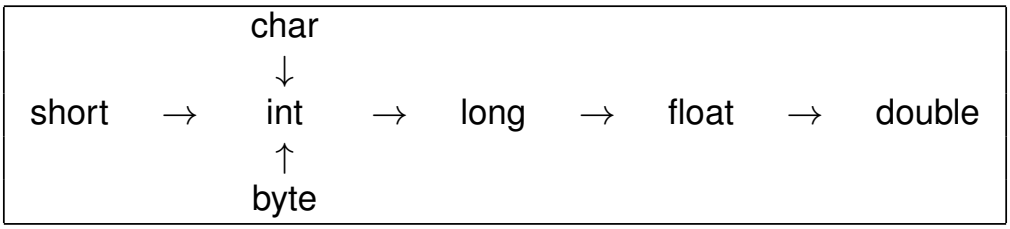
- ¿Qué ocurre si al operar nos salimos del rango?

Operaciones Fuera de Rango

- ¿Qué ocurre si al operar nos salimos del rango?
- Entre enteros
  - Resultado no coherente.
  - Desborda bits más significativos y error de signo
- Entre reales
  - Toma el mayor valor posible (`Infinity` o `-Infinity`)
- Casos Especiales
  - `Infinity`: Infinito positivo. Ejemplo `1/0`
  - `NaN`: Not a Number. Ejemplo Infinity-Infinity, Math.sqrt(-1).
  - No se puede asignar directamente `Infinity` o `NaN` a una variable

Mezcla de Tipos en Operaciones Aritméticas

- Reglas de compatibilidad
  - Todos los tipos numéricos son compatibles entre si. char también, pasa a int
  - boolean no es compatible con ningún otro tipo
  - String no es compatible con char ni con ningún otro tipo,
- Al realizar operaciones con tipos combinados, el resultado es del tipo de mayor capacidad.



Asignación de datos entre distintos tipos de variable

- Automático cuando:
  - Los tipos son compatibles, y
  - El tipo destino es mayor que el origen (no se pierde información)

Asignación de datos entre distintos tipos de variable

- Automático cuando:
  - Los tipos son compatibles, y
  - El tipo destino es mayor que el origen (no se pierde información)

```
int a = 3;
double d;
d = a;
```

Casting: conversión explícita entre tipos

- Cuando no se cumple lo anterior hay que forzar la conversión: Casting

```
double d = 3.5;
int a;
a = (int) d;
```

Casting: conversión explícita entre tipos

- Cuando no se cumple lo anterior hay que forzar la conversión: Casting

```
double d = 3.5;
int a;
a = (int) d;
```

- De real a entero: la parte decimal se trunca
- Si se sale de rango en `byte` y `short` resultado no coherente
- Se guarda el máximo valor posible

Operador de concatenación de *String*

- Operador + para concatenar `String`

```
String s1 = "Hola ";
String s2 = " mundo";

String s3 = s1 + s2 ;
String s4 = "Hola otra vez " + s2;
```

Operador de concatenación de *String*

- Operador + para concatenar `String`

```
String s1 = "Hola ";
String s2 = " mundo";

String s3 = s1 + s2 ;
String s4 = "Hola otra vez " + s2;
```

- Conversión automática si se opera con otro tipo

```
String s5 = "Hola otra vez " + 3;
```

Operadores relacionales

Operador	Descripción	
==	Igual	`3==5, 3.0 == 3`
!=	Distinto	`3!=5`
>	Mayor	`'a' > 'c'`
<	Menor	`'a' < 'c'`
>=	Mayor ó igual	`'a' >= 60`
<=	Menor ó igual	`'a' >= 60`

Operadores relacionales

Operador	Descripción	
==	Igual	`3==5, 3.0 == 3`
!=	Distinto	`3!=5`
>	Mayor	`'a' > 'c'`
<	Menor	`'a' < 'c'`
>=	Mayor ó igual	`'a' >= 60`
<=	Menor ó igual	`'a' >= 60`

- Para tipos básicos (`boolean`, sólo `==` y `!=`)
- Resultado de tipo boolean
- Mezcla de tipos
- ¡Cuidado con la igualdad y los números reales!
- Reales especiales: `Infinity` == `Infinity`, pero `NaN` != `Nan`

Operadores lógicos

Operador	Descripción
&	AND
|	OR
^	XOR
!	NOT
&&	AND (evaluación perezosa – cortocircuito)
||	OR (evaluación perezosa – cortocircuito)

Operadores abreviados: operación + asignación

- El único operador de asignación es =
- Pero hay abreviaturas que permiten operar y asignar en una sola instrucción.

Operador	Descripción
+=	`a += <expresión>` significa `a = a + <expresión>`

Operadores abreviados: operación + asignación

- El único operador de asignación es =
- Pero hay abreviaturas que permiten operar y asignar en una sola instrucción.

Operador	Descripción
+=	`a += <expresión>` significa `a = a + <expresión>`

- Igual con otros operadores aritméticos: -=, *=, /=
- Y también lógicos: &=, |=, ^=

Precedencia de operadores

Se evalúan de izqda a dcha, excepto en *Asignación* que es de dcha a izqda

1. Paréntesis: ()
2. Unarios: !, $\sim$, ++, $--$, (*cast*)
3. Multiplicativos: $*$, /, %
4. Aditivos: +, $-$
5. Rotación: $>>$, $<<$
6. Relacional: $>$, $>=$, $<$, $<=$
7. Igualdad: ==, ! =
8. Lógico: &
9. Lógico: ˆ
10. Lógico: |
11. Lógico: &&
12. Lógico: ||
13. Condicional: ? :
14. Asignación: =, + = (*etc.*), & = (*etc.*)
15. Coma: ,

Precedencia de operadores

Se evalúan de izqda a dcha, excepto en *Asignación* que es de dcha a izqda

1. Paréntesis: ()
2. Unarios: !, $\sim$, ++, $--$, (*cast*)
3. Multiplicativos: $*$, /, %
4. Aditivos: +, $-$
5. Rotación: $>>$, $<<$
6. Relacional: $>$, $>=$, $<$, $<=$
7. Igualdad: ==, ! =
8. Lógico: &
9. Lógico: ˆ
10. Lógico: |
11. Lógico: &&
12. Lógico: ||
13. Condicional: ? :
14. Asignación: =, + = (*etc.*), & = (*etc.*)
15. Coma: ,

¡Usad paréntesis en caso de duda!

En este tema

Tema 3: Introdución a Java

- La JVM (*Java Virtual Machine*)
- Tipos básicos o tipos primitivos
- Creación de programas en Java
- Variables y Constantes
- Operadores
- Otras cuestiones: datos, comentarios y errores

Origen de los datos del programa

- Preprogramados
- Argumentos del programa
- Entrada *Standard*
- Fichero
- Aleatorios

Entrada *Standard* y fichero: clase `Scanner`

```
import java.util.Scanner; // se importa la clase Scanner


 class Ejemplo {


 public static void main(String arg[]){

      String variableString;

      Scanner entrada=new Scanner(System.in);

      System.out.print("Introduce un texto: ");

      variableString =  entrada.next();

      System.out.println("El texto es: "+variableString);

 }

}
```

Entrada *Standard* y fichero: clase `Scanner`

- Estos métodos leen un dato. Los datos se separan por espacios.
- Para leer un `String` se utiliza: `entrada.next()`
- Para leer enteros (`int`) se utiliza: `entrada.nextInt()`
- Existen más métodos de la clase `Scanner`

Entrada *Standard* y fichero: clase `Scanner`

- Estos métodos leen un dato. Los datos se separan por espacios.
- Para leer un `String` se utiliza: `entrada.next()`
- Para leer enteros (`int`) se utiliza: `entrada.nextInt()`
- Existen más métodos de la clase `Scanner`
- Ejemplo
  - Programa que lee un entero y un real en doble precisión (*double*) y los suma en un real en precisión simple (*float*)

Datos aleatorios

- Se utiliza `Math.random()`
- Este método devuelve un `double` mayor o igual que 0.0 y menor que 1.0

Comentarios

- El código **debe** tener comentarios
- Comentario de línea: //
- Comentario de bloque: /* ... */
- Comentario para *Javadoc*: /** ... */
- En la cabecera de cada fichero fuente se suele incluir:
  - Datos del creador
  - Versión
  - Fecha

Errores de programación

- Errores de compilación: por ejemplo errores de sintaxis, variables no declaradas/inicializadas, etc.,
- Errores de ejecución: por ejemplo memoria insuficiente, índices u operaciones fuera de rango, etc.
- No hay errores como tal, pero la lógica del programa es incorrecta

Parte V

Condicionales y bucles

Programación

PLG
Planning and Learning Group

Universidad Carlos III de Madrid

Tema 4: Condicionales y Bucles

Control de Flujo

- Instrucciones para romper el flujo de ejecución secuencial
- 3 tipos:
  - Condicionales: `if, switch`
  - Bucles: `while, do while, for`
  - Ramificación: `break, continue, return`

En este tema

Tema 4: Condicionales y Bucles

- Condicionales
- Bucles

Condicionales `if`

- Sintaxis

```
if ( condicion )
      sentencia;

if ( condicion )
      { sentencias }
```

- La condición siempre tiene que ser de tipo `boolean`
- Usar llaves `{}` para más de una sentencia
- Por convención el bloque condicionado de instrucciones va indentado a la derecha

Condicionales `if-else`

- Sintaxis

```
if ( condicion )
        sentencia;
else
        sentencia;

if ( condicion )
        { sentencias }
else
        { sentencias }
```

- Para sentencias o bloque de sentencias excluyentes

Encadenamiento de condicionales

```
public static void main(String[] args) {
    Scanner entrada = new Scanner(System.in);
    System.out.println("Introduzca nota del alumno");
    double nota = entrada.nextDouble();

    if (nota < 0 || nota > 10)
        System.out.println("Nota inválida");
    else if (nota >= 5)
        System.out.println("Alumno APROBADO");
    else
        System.out.println("Alumno SUSPENSO");
}
```

Condicionales `switch`

- Cuando hay que elegir entre varias alternativas
- Sobre variables `int` o compatibles. No con long.
- Vale con `String` a partir de Java7
- Cuando un valor se cumple se ejecuta hasta que encuentra un `break`
- Sintaxis:

```
switch (variable){
    case valor1:
        sentencia;
        sentencia;
        [ break; ]
    case valor2:
        sentencia;
        [break;]
    default:
        sentencia;
}
```

Ejemplo switch

```
double n1, n2, res=0;
String op;
Scanner sc = new Scanner (System.in);

System.out.println("CALCULADORA BASICA");
System.out.println("Primer número");
n1 = sc.nextDouble();
System.out.println("Operación");
op = sc.next();
System.out.println("Segundo número");
n2 = sc.nextDouble();

switch (op) {
    case "+":
        res = n1 + n2;
        break;
    case "-":
        res = n1 - n2;
        break;
    default:
        System.out.println("Operacion no reconocida");
}
System.out.println(" = " + res);
```

Ámbito de una variable

- Llamamos bloque al conjunto de sentencias entre llaves {}
- En todo programa hay al menos 2 bloques, el de la clase y el del main
- Los bloques establecen el ámbito de una variable, o porción de código donde se puede utilizar
- Ejemplo con bloques de `if` anidados

En este tema

Tema 4: Condicionales y Bucles

- Condicionales
- Bucles

Bucles

- Necesidad de repetir bloques de código
- Tipos de bucles:
  - Bucles que se repiten mientras se cumpla una condición: `while`, `do while`
  - Bucles que se repiten un número determinado de veces: `for`
- Realmente con un tipo es suficiente
- Elementos importantes:
  - Variable(s) de control: Para decidir si continuamos repitiendo o no. Debe cambiar en cada iteración
  - Condición de control: Se comprueba en cada repetición. Si es verdadera se repite el bloque otra vez

Bucles `while`

- Se repite mientras se cumpla la condición de control
- Si no se cumple inicialmente **no se ejecuta ninguna vez**
- Sintaxis:

```
while ( condicion )
      { sentencias }
```

Ejemplo `while`

Hacer un programa que sume los números pares hasta N

Ejemplo `while`

```
Scanner entrada = new Scanner(System.in);
System.out.println("Introduce número límite:");

int numeroLimite = entrada.nextInt();

int resultado = 0;
int numeroActual = 0;


while (numeroActual <= numeroLimite)
{

   resultado += numeroActual;
   numeroActual += 2;

}

System.out.println("La suma de los números pares hasta el " + numeroLimite +
                   " es " + resultado);

entrada.close();
```

Bucles `do while`

- Se repite mientras se cumpla la condición de control
- Siempre se ejecuta **al menos una vez**
- Sintaxis:

```
do
    { sentencias }
while ( condicion );
```

Ejemplo `do while`

Hacer un programa en el que el usuario introduzca un número entre 0 y 10. Si el número introducido esta fuera de ese intervalo se debe pedir nuevamente.

Ejemplo `while`

```
Scanner entrada = new Scanner(System.in);
int numero;


do
{
    System.out.println("Introduce un número entre 0 y 10:");
    numero = entrada.nextInt();
} while (numero < 0 || numero > 10);

System.out.println("El numero introducido es " + numero);

entrada.close();
```

Bucles `for`

- El más potente y versátil de las sentencias de bucles
- Suele utilizarse cuando conocemos el número de veces que queremos repetir un bloque
- Elementos:
  - Inicialización: declara y/o da valor inicial a la(s) variable(s) de control
  - Control: condición que debe cumplirse para permanecer en el bucle. Tipo `boolean`
  - Actualización: modifica las variables de control al final de cada ciclo
- Sintaxis:

`for`(*inicialización* ; *condicion_control* ; *actualización*)
{ *sentencias* }

Ejemplo `for`

Hacer un programa que sume los 100 primeros números naturales.

Ejemplo `for`

```
int suma = 0;

for (int i = 1; i <= 100; i++){
  suma = suma + i;
}
```

Bucles `for`

- Tiempo de vida de las variables de control
  - Si declaramos dentro del `for`, sólo existe en ese ámbito.

```
for ( int i = 0 ; ; )
    { sentencias }
```

- Si declaramos fuera, cualquier cambio dentro del bucle afecta la variable

```
int i = 5;
for ( i = 0 ; ; )
    { sentencias }
```

Bucles `for`

- En la inicialización y la actualización se pueden poner varias variables a la vez, separadas por comas

```
int i, j;
for(i = 0, j = 10 ; i <= j ; i++, j--)
    System.out.println(i + " " + j);
```

Bucles `for`

- En la inicialización y la actualización se pueden poner varias variables a la vez, separadas por comas

```
int i, j;
for(i = 0, j = 10 ; i <= j ; i++, j--)
    System.out.println(i + " " + j);
```

- Si se declaran varias variables en la inicialización, éstas tienen que ser del mismo tipo
- Se pueden dejar vacías, tanto inicialización como control o actualización, pero se mantienen los ";"
- Modificar variables de control dentro del bucle **NO** es una buena práctica

Bucles `for` anidados

```
int exterior = 0, interior = 0;

for (int i=0; i<3; i++){

    exterior++;

    for (int j=0; j<5; j++){

        interior++;
    }
}
System.out.println("El bucle exterior repitió "+exterior+" veces");
System.out.println("El bucle interior repitió "+interior+" veces");
```

`break`

- Se utiliza en el `switch` y en bucles
- En bucles sirve para salir del bucle
- Uso no recomendado, se puede sustituir por una condición extra en el bucle

Ejemplo `break`

```
for (int i=0; i<25; i++){
     if (i == 12)
         break;
     System.out.print(i+" ");
}
System.out.println("ABCD");
```

`continue`

- Vuelve a la condición del bucle, saltando lo que queda por ejecutar dentro de él
- En caso de `for` pasa a la siguiente iteración
- Uso no recomendado, se puede resolver con un condicional dentro del bucle

Ejemplo `continue`

```
for (int i=0; i<25; i++){
     if (i == 12)
         continue;
     System.out.print(i+" ");
}
System.out.println("ABCD");
```

Parte VI

Estructuras de datos sencillas

Programación

PLG
Planning and Learning Group

Universidad Carlos III de Madrid

Estructuras de Datos Simples

En este tema

Estructuras de Datos Simples

- Arrays

Estructuras de Datos

- **Estructura de Datos**: Conjunto de datos agrupados en una sola variable
  - Reservamos varias celdas de memoria, una para cada dato
  - Asignamos un nombre al grupo completo de datos
- Estructuras de Datos Básicas:
  1. Arrays: para agrupar datos del mismo tipo
  2. Registros: para agrupar datos de distinto tipo
     - Java no tiene registros. En su lugar utiliza objetos. Una especie de “super registros” que veremos más adelante.

Arrays

- **Arrays**: Conjunto de datos del mismo tipo que ocupan posiciones sucesivas de memoria y a los que se accede mediante una única variable.
- Definición de arrays
  - Definir la variable

```
int[] numeros;
int otroNumeros[];
```

  - Definir el número de elementos y reservar memoria para ellos

```
numeros = new int[12];
```

  - Multiples definiciones y asignaciones, igual que en tipos básicos

```
int[] a, b = new int[5], c = new int[8];
```

Arrays en la memoria

- Un array es un puntero, una referencia a otra celda de memoria
- Cuando se declara un array su contenido es el valor especial `null` (no apunta a ningún sitio).
- Cuando reservamos memoria nueva y la asignamos a un array, los datos individuales tienen valor por defecto.
  - 0 para los números
  - `false` para los `boolean`
  - la cadena vacía para los `char`
  - `null` para los `String`

Acceso a Elementos

- Podemos asignar valores iniciales a los elementos de un array

```
int[] nums = new int [] {2,4,8,10};
```

- Para acceder a los elementos de un array indicamos la posición entre corchetes

```
int a = 7, b;
int[] nums =new int[5];
nums[0] = 15;
nums[2] = 10;
b = nums[2];
```

- Las posiciones van de 0 a (num. elementos - 1)

Arrays y Bucles

- Se suele utilizar un bucle cuando tenemos que asignar o acceder a varios elementos de un array
- `nombreArray.length` nos indica el tamaño del array para reconocer el número de iteraciones que necesitamos.

```
// Los 10 primeros numeros de la serie Fibonacci
int i;
int [] fibonacci = new int [10];

fibonacci[0] = 1;
fibonacci[1] = 1;

for (i = 2; i < 10; i++)
{
    fibonacci[i] = fibonacci[i-1] + fibonacci[i-2];
}

for (i = 0; i < fibonacci.length; i++)
{
    System.out.print(fibonacci[i] + " ");
}
```

Asignaciones y Copias de Arrays

- Las variables de un tipo no básico almacenan siempre un puntero, una referencia a otra celda de memoria
- ¡Ojo! la asignación directa (con “=”) de un array a otro copia el puntero
- ¡Ojo! dos arrays son iguales (con “==”) solamente cuando apuntan a la misma dirección de memoria

Asignaciones y Copias de Arrays

- Para copiar el contenido de dos arrays podemos
  1. Copiar elemento a elemento utilizando un bucle
  2. Utilizar el `System.arraycopy(origen, pos, destino, pos, n_elementos)`
- Para ver si el contenido de dos arrays es igual hay que comparar elemento a elemento

Ejemplo copia de arrays

```
int[] nums = new int [] {2,4,8,10};

int[] numsCopy = new int[4];

System.arraycopy(nums, 0, numsCopy, 0, nums.length);
```

Más sobre Arrays

- Una vez definido el número de elementos, este no se puede cambiar. Hay que crear un nuevo array y copiar los elementos
- Si declaramos un array como `final`, lo que es constante es su dirección de memoria. Los elementos de dicho array se pueden cambiar

Bucles `for each` para Arrays

- A partir de Java 5 hay una forma alternativa de iterar sobre los elementos de un array
- La instrucción se lee como **for each**, o para cada elemento del array repetir el bucle.
- Sintaxis:

```
for(tipo_array variable_elemento : nombreArray)
    { sentencias }
```

- No vale para cambiar valores del array

Ejemplo `for` each para arrays

```
int[] nums = new int [] {2,4,8,10};
int[] numsCopy = new int[4];

System.arraycopy(nums, 0, numsCopy, 0, nums.length);

System.out.println("array nums");

for (int i = 0; i< nums.length; i++){
    System.out.println(nums[i]);
}

System.out.println("array numsCopy");

for (int elem : numsCopy){
    System.out.println(elem);
}
```

Matrices

- Matrices: Arrays de 2 dimensiones
- Declaración y creación en memoria similar a los de una dimensión

```
int[][] matriz = new int[3][3];
int[][] matriz2 = {{2,4},{3, 2}};
```

- La primera posición corresponde a las filas y la segunda a las columnas
- *En memoria cada fila es un array*

Matrices

- Matrices: Arrays de 2 dimensiones
- Declaración y creación en memoria similar a los de una dimensión

```
int[][] matriz = new int[3][3];
int[][] matriz2 = {{2,4},{3, 2}};
```

- La primera posición corresponde a las filas y la segunda a las columnas
- *En memoria cada fila es un array*
- Ejemplo: memoria para `int [][] matriz = new int [4][3];`

Matrices: acceso y recorrido

- Acceso a elementos (ejemplo): `matriz[1][1]`
- Recorrido: for anidados

Matrices: acceso y recorrido

- Acceso a elementos (ejemplo): `matriz[1][1]`
- Recorrido: for anidados

```
int i,j;

int[][] matriz1 = new int [3][2];

int numeroInicial = 1;

for ( i = 0; i < matriz1.length; i++){
     for (j = 0; j < matriz1[i].length; j++){
       matriz1[i][j] = numeroInicial ;
       numeroInicial++;
     }
}
```

Matrices: recorrido `for` each

```
int [][] matriz = {{1,2} , {3,4}, {5,6}};

for (i = 0; i < matriz.length; i++){
   for (j = 0; j < matriz[i].length; j++){
      System.out.print(matriz[i][j]+" ");
    }
System.out.println();
}

System.out.println();
for (int [] filas : matriz){
   for (int elemento: filas){
      System.out.print(elemento+" ");
   }
System.out.println();
}
```

Matrices: filas de distinta longitud

```
int [][] a;
a = new int [3][];

a[0] = new int [1];
a[1] = new int [3];
a[2] = new int [2];
```

Arrays multidimensionales

- Se pueden construir arrays de más de dos dimensiones

```
int multi[][][] = new int[3][3][3];
```

- Aunque la tendencia es usarlos poco para no complicar el código

Parte VII

Clases, objetos y métodos

Programación

PLG
Planning and Learning Group

Universidad Carlos III de Madrid

Funciones

¿Qué es una función?

- Fragmento de código etiquetado con un nombre
- ¿Por qué usar funciones?
  - Evitar duplicidad en el código. Código más organizado
  - Facilitar el mantenimiento. Las modificaciones sólo se realizan en un lugar.

Llamadas a funciones

1. El programa principal se ejecuta normalmente hasta que encuentra una función
2. Entonces salta al código de la función
3. Ejecuta la función
4. Cuando la ejecución termina, el programa continua a partir del punto en el que se invocó la función

Valor de retorno de una función

- Las funciones pueden devolver un resultado cuyo tipo debe coincidir con el tipo de retorno de la función

```
<tipo> NombreFuncion (){
      <cuerpo de la función>
      return resultado
}
```

- Si la función no devuelve nada:
  - El tipo de retorno es `void`
  - La función no tiene `return`

Ejemplo

```
void saludo (){
    System.out.println("Hola");
}
```

Parámetros de una función

- Permiten pasar a la función datos de entrada

Parámetros de una función

- Permiten pasar a la función datos de entrada

```
<tipo> NombreFuncion (<tipo> parametro1, <tipo> parametro2, ...){
      <cuerpo de la función>
      return resultado
}
```

Parámetros de una función

- Permiten pasar a la función datos de entrada

```
<tipo> NombreFuncion (<tipo> parametro1, <tipo> parametro2, ...){
      <cuerpo de la función>
      return resultado
}
```

- Son variables del ámbito (bloque) de la función que ya están inicializadas

Parámetros de una función

- Permiten pasar a la función datos de entrada

```
<tipo> NombreFuncion (<tipo> parametro1, <tipo> parametro2, ...){
      <cuerpo de la función>
      return resultado
}
```

- Son variables del ámbito (bloque) de la función que ya están inicializadas

```
int Suma (int a, int b){
     int suma = a + b;
     return suma;
}
```

- Se inicializan en la llamada a la función

Parámetros de una función

- Permiten pasar a la función datos de entrada

```
<tipo> NombreFuncion (<tipo> parametro1, <tipo> parametro2, ...){
      <cuerpo de la función>
      return resultado
}
```

- Son variables del ámbito (bloque) de la función que ya están inicializadas

```
int Suma (int a, int b){
     int suma = a + b;
     return suma;
}
```

- Se inicializan en la llamada a la función
- Por ejemplo, en el programa principal:

```
int sumando1 = 3;
int sumando2 = 4
int resultadoSuma =  Suma (sumando1, sumando2);
```

¿Funciones en java?

- Programación estructurada: tipos/estructuras de datos, estructuras de control (condicionales y bucles) y funciones

¿Funciones en java?

- Programación estructurada: tipos/estructuras de datos, estructuras de control (condicionales y bucles) y funciones
- Programación orientada a objetos (POO)
  - El código se organiza de otra manera
  - Uno de los objetivos es que los datos estén junto a las funciones que los manipulan

¿Funciones en java?

- Programación estructurada: tipos/estructuras de datos, estructuras de control (condicionales y bucles) y funciones
- Programación orientada a objetos (POO)
  - El código se organiza de otra manera
  - Uno de los objetivos es que los datos estén junto a las funciones que los manipulan
- En Java (POO) NO existen funciones como tal, el código se organiza en clases

Clases, objetos y métodos

En este tema

Clases

- Una clase define de forma abstracta:
  - Un conjunto de variables, cada una con su tipo y nombre: atributos
  - Un conjunto de funciones que por lo general manipulan esos datos: métodos

Clases

- Una clase define de forma abstracta:
  - Un conjunto de variables, cada una con su tipo y nombre: atributos
  - Un conjunto de funciones que por lo general manipulan esos datos: métodos
- Una clase se define en un fichero .java cuyo nombre debe coincidir con el nombre de la clase
- Los atributos pueden ser de distinto tipo
- El ámbito de los atributos es global a toda la clase

Ejemplo de definición de una clase y sus atributos

```
public class Fecha {
    public int dia;
    public String mes;
    public int anyo;
    public String diaSemana;
    public boolean festivo;
}
```

Objetos

- Una vez definida la clase, se pueden declarar variables del *"tipo de datos"* que define la clase
- Una variable cuyo tipo es una clase contiene una referencia a un objeto
- Un objeto es un elemento concreto que pertenece a esa clase: una instancia de esa clase

Declaración de variables de una clase

- Se hace en el código de una clase distinta
- Se expresa como si se declarara un tipo de dato cualquiera
- Hay que crear el objeto reservando el espacio de memoria necesario con el operador new

```
Fecha hoy = new Fecha();
Fecha ayer;

ayer = new Fecha();
```

Declaración de variables de una clase

- Se hace en el código de una clase distinta
- Se expresa como si se declarara un tipo de dato cualquiera
- Hay que crear el objeto reservando el espacio de memoria necesario con el operador new

```
Fecha hoy = new Fecha();
Fecha ayer;

ayer = new Fecha();
```

- Nos referiremos a las variables que contienen referencias a objetos directamente como objetos

Acceso a atributos

- La sintaxis para acceder a los atributos es:
  `<nombreObjeto>.<nombreAtributo>`
- Ejemplo:

```
hoy.dia = 24;
hoy.mes = "Octubre";
hoy.anyo = 2015;

ayer.dia = hoy.dia - 1;
ayer.mes = hoy.mes;
ayer.anyo = hoy.anyo;
```

Valores Iniciales de los Atributos

- Cuando se crea un objeto, los atributos tienen valor por defecto.
  - 0 para los números
  - `false` para los `boolean`
  - la cadena vacía para los `char`
  - `null` para los `String`

Valores por Defecto de los Atributos

- También se puede declarar la clase con valores por defecto para los atributos

```
public class Fecha {
    public int dia = 1;
    public String mes = "Enero";
    ...
}
```

- En este caso todos los objetos que se creen de esa clase, tendrán esos valores por defecto en sus atributos

Asignación y Copia de Objetos

- Un objeto es un puntero a una dirección de memoria que contiene los atributos del objeto y alguna otra información adicional
- Asignar objetos con '=' copia el puntero
- Comparar objetos con '==' compara el puntero
- Para copiar un objeto en otro hay que crear el nuevo objeto y copiar atributo a atributo
- Para comparar objetos hay que comparar atributo a atributo

En este tema

Funciones

Clases, objetos y métodos

- Clases y objetos
- Métodos
  - Constructores
  - Resto Métodos

Métodos

- Los métodos son funciones que se implementan dentro de una clase
- Los métodos implementan operaciones comunes a todos los objetos de la clase
- Los atributos son accesibles a todos los métodos de una clase

En este tema

Funciones

Clases, objetos y métodos

- Clases y objetos
- Métodos
  - Constructores
  - Resto Métodos

Constructores

- Los constructores son métodos especiales que sirven para asignar valores a los atributos de un objeto en el **momento en que éste se crea con** `new`
- Los constructores tienen el mismo nombre que la clase

Constructores

- Los constructores son métodos especiales que sirven para asignar valores a los atributos de un objeto en el **momento en que éste se crea con** `new`
- Los constructores tienen el mismo nombre que la clase
- Existen dos tipos:
  - Constructor por defecto: no tiene argumentos: se utiliza si los valores de los argumentos se obtienen mediante cálculos
  - Constructor completo: sus argumentos representan el valor de los atributos al crear el objeto

Ejemplo de constructor

```
public class Fecha {
    public int dia;
    public String mes;
    public int anyo;
    public String diaSemana;
    public boolean festivo;

    public Fecha (int d, String m, int a, String ds, boolean f) {
        dia = d;
        mes = m;
        anyo = a;
        diaSemana = ds;
        festivo = f;
    }
}
```

Otros ejemplos de constructor

```
public class Fecha {
    public int dia;
    public String mes;
    public int anyo;
    public String diaSemana;
    public boolean festivo;

    public Fecha (int dia, String mes, int anyo, boolean festivo) {
        this.dia = dia;
        this.mes = mes;
        this.anyo = anyo;
        this.festivo = festivo;

    public Fecha (int anyo) {
        this(1,”Enero”,anyo,,true);
    }
}
```

En este tema

Otros Métodos

- Se sitúan dentro de la clase
- Habitualmente después de los *atributos*
- Y antes del *main* si existe

Métodos – estructura básica: cabecera + cuerpo

```
<visibilidad> <modificador-static> <tipo> nombreMetodo (<tipo> param1, <tipo> param2, ...)
      <cuerpo del método>
      return resultado
```

Métodos – estructura básica: cabecera + cuerpo

```
<visibilidad> <modificador-static> <tipo> nombreMetodo (<tipo> param1, <tipo> param2, ...)
      <cuerpo del método>
      return resultado
```

- Son funciones comunes a todos los objetos de la clase:
  - Tipo de retorno: tipo del valor que devuelve el método
  - Nombre del método: en minúsculas (*camelCase*) por convención
  - Parámetros del método: lista de parámetros con su tipo

Métodos – estructura básica: cabecera + cuerpo

```
<visibilidad> <modificador-static> <tipo> nombreMetodo (<tipo> param1, <tipo> param2, ...)
      <cuerpo del método>
      return resultado
```

- Son funciones comunes a todos los objetos de la clase:
  - Tipo de retorno: tipo del valor que devuelve el método
  - Nombre del método: en minúsculas (*camelCase*) por convención
  - Parámetros del método: lista de parámetros con su tipo
- Con algunos elementos más específicos de POO:
  - <visibilidad>
  - <modificador-static>

Ejemplo de Métodos – en una clase cualquiera

```
public class Fecha {
    public int dia;
    public String mes;
    public int anyo;
    public String diaSemana;
    public boolean festivo;
```

Ejemplo de Métodos – en una clase cualquiera

```
public class Fecha {
    public int dia;
    public String mes;
    public int anyo;
    public String diaSemana;
    public boolean festivo;

    /** CONSTRUCTOR **/
    public Fecha (int dia, String mes, int anyo, boolean festivo) {
        this.dia = dia;
        this.mes = mes;
        this.anyo = anyo;
        this.festivo = festivo;
```

Ejemplo de Métodos – en una clase cualquiera

```
public class Fecha {
    public int dia;
    public String mes;
    public int anyo;
    public String diaSemana;
    public boolean festivo;

    /** CONSTRUCTOR **/
    public Fecha (int dia, String mes, int anyo, boolean festivo) {
        this.dia = dia;
        this.mes = mes;
        this.anyo = anyo;
        this.festivo = festivo;

    /** OTROS MÉTODOS **/
    public int anyoSiguiente () {
        return anyo+1;
    }
```

Ejemplo de Métodos – en una clase cualquiera

```
public class Fecha {
    public int dia;
    public String mes;
    public int anyo;
    public String diaSemana;
    public boolean festivo;

    /** CONSTRUCTOR **/
    public Fecha (int dia, String mes, int anyo, boolean festivo) {
        this.dia = dia;
        this.mes = mes;
        this.anyo = anyo;
        this.festivo = festivo;

    /** OTROS MÉTODOS **/
    public int anyoSiguiente () {
        return anyo+1;
    }

    public void printDia () {
        System.out.println("El día es: " + dia);

    }
}
```

Ejemplos de Métodos – en la clase del programa principal

```
public class  ProgramaPrincipal {
```

Ejemplos de Métodos – en la clase del programa principal

```
public class  ProgramaPrincipal {

     static int  maximo (int a, int b) {
         int  max;
         if  (a > b)
             max = a;
         else
             max = b;
         return max;
      }
```

Ejemplos de Métodos – en la clase del programa principal

```
public class  ProgramaPrincipal {

     static int  maximo (int a, int b) {
         int  max;
         if  (a > b)
             max = a;
         else
             max = b;
         return max;
      }

     public static void main (String [] args) {
         int a = 2, b = 3, c = 4, max1, max2, max3;
         max1 = maximo (a, b);
         max2 = maximo (b, c);
         max3 = maximo (max1, max2);
      }
 }
```

Valor de retorno

- Es `void` en caso de que no devuelva nada (método sin `return`)
- El `return` sólo puede devolver un único dato

Paso de parámetros

- Los parámetros son variables locales al método que ya están inicializadas
- Se inicializan en la llamada al método
- Paso de parámetros por valor vs. paso de parámetros por referencia

Paso de parámetros

Paso de parámetros POR VALOR

- El parámetro contiene una copia del valor, variable o expresión que se utiliza en la llamada
- Cuando se modifica un parámetro se modifica la copia y no el dato original. ¡Los cambios no permanecen al salir de la función!
- Los tipos básicos y *Strings* en Java se pasan por valor

Paso de parámetros

Paso de parámetros POR REFERENCIA

- La variable que se utiliza en la llamada contiene una dirección (referencia)
- El parámetro contiene esa referencia
- La referencia permite acceder al valor original. ¡Los cambios permanecen al salir de la función!
- Los *arrays* y objetos se pasan por referencia

Ejemplo – paso de parámetros

```
public static int SumaUno(int a) {
        a++;
        return a; }
```

Ejemplo – paso de parámetros

```
public static int SumaUno(int a) {
        a++;
        return a; }

public static int [] SumaUno(int [] a) {
        a[0]++;
        return a; }
```

Ejemplo – paso de parámetros

```
public static int SumaUno(int a) {
        a++;
        return a; }


public static int [] SumaUno(int [] a) {
        a[0]++;
        return a; }


public static void main(String[] args)
        int a = 3;
        int [] b = {3};
        System.out.println("a: " + a);
        System.out.println("b[0]: " + b[0]);
        SumaUno(a);
        SumaUno(b);
        System.out.println("a: " + a);
        System.out.println("b[0]: " + b[0]);
```

Documentación Javadoc

Se deben documentar los métodos

- Explicar qué hace el método y qué devuelve
- Explicar qué representan los parámetros

Ejemplo – comentarios Javadoc

```
/**
* The AddNum program implements the addition of two given integers.
*
* @author  Zara Ali
* @version 1.0
* @since   2014-03-31
*/
public class AddNum {

   /**
   * This method is used to add two integers.
   * @param numA This is the first paramter to addNum method
   * @param numB  This is the second parameter to addNum method
   * @return int This returns sum of numA and numB.
   */
   public static int addNum(int numA, int numB) {
      return numA + numB;    }
   /**
   * This is the main method which makes use of addNum method.
   * @param args Unused.
   * @return Nothing.
   */
   public static void main(String args[]) {
     int sum = addNum(10, 20);
     System.out.println("Sum of 10 and 20 is :" + sum);
  }
}
```

Ejemplo – comentarios Javadoc

```
/**
* The AddNum program implements the addition of two given integers.
*
* @author  Zara Ali
* @version 1.0
* @since   2014-03-31
*/
public class AddNum {

   /**
   * This method is used to add two integers.
   * @param numA This is the first paramter to addNum method
   * @param numB  This is the second parameter to addNum method
   * @return int This returns sum of numA and numB.
   */
   public static int addNum(int numA, int numB) {
      return numA + numB;    }
   /**
   * This is the main method which makes use of addNum method.
   * @param args Unused.
   * @return Nothing.
   */
   public static void main(String args[]) {
     int sum = addNum(10, 20);
     System.out.println("Sum of 10 and 20 is :" + sum);
  }
}
```

Sobrecarga de métodos

- Dos o más métodos con el mismo nombre en el mismo ámbito
  - En una clase
  - En el programa principal
- Se distinguen por su número de parámetros y el tipo de los mismos
- Los métodos sobrecargados pueden devolver distinto tipo
- Aunque el tipo de retorno no se usa para diferenciarlos

Ejemplos – métodos sobrecargados

```
public class  ProgramaPrincipal {

     static int  maximo (int a, int b) {
         int  max;
         if  (a > b)
             max = a;
         else
             max = b;
```

Ejemplos – métodos sobrecargados

```
public class  ProgramaPrincipal {

     static int  maximo (int a, int b) {
         int  max;
         if  (a > b)
             max = a;
         else
             max = b;
         return max;
      }

     static int  maximo (int a, int b, int c) {
```

Ejemplos – métodos sobrecargados

```
public class  ProgramaPrincipal {

     static int  maximo (int a, int b) {
         int  max;
         if  (a > b)
             max = a;
         else
             max = b;
         return max;
      }

     static int  maximo (int a, int b, int c) {
         return maximo(maximo(a,b), maximo(b,c));
      }
```

Ejemplos – métodos sobrecargados

```
public class  ProgramaPrincipal {

     static int  maximo (int a, int b) {
         int  max;
         if  (a > b)
             max = a;
         else
             max = b;
         return max;
      }

     static int  maximo (int a, int b, int c) {
         return maximo(maximo(a,b), maximo(b,c));
      }

     public static void main (String [] args) {
         int a = 2, b = 3, c = 4, max1, max2, max3;
         max3 = maximo (a, b ,c);
      }
 }
```

Métodos estáticos (<modificador-static>)

- Cada objeto de una clase tiene una copia de los atributos y una copia de los métodos
- Si un método no utiliza los atributos puede ser `static`: no hace falta una copia en cada objeto
- Un método **static** es un método de clase
- ¡No hace falta un objeto para invocarlo!

Invocación de métodos y acceso a atributos en objetos

- Una vez creado un objeto podemos acceder a sus atributos y métodos utilizando un punto:

<objeto>.<atributo o método>

Ejemplo – acceso métodos y atributos

Fecha.java

```
public class Fecha {
    public int dia;
    public String mes;
    public int anyo;
    public String diaSemana;
    public boolean festivo;
```

Ejemplo – acceso métodos y atributos

Fecha.java

```
public class Fecha {
    public int dia;
    public String mes;
    public int anyo;
    public String diaSemana;
    public boolean festivo;

    public Fecha (int dia, String mes, int anyo, boolean festivo) {
        this.dia = dia;
        this.mes = mes;
        this.anyo = anyo;
        this.festivo = festivo;
```

Ejemplo – acceso métodos y atributos

Fecha.java

```
public class Fecha {
    public int dia;
    public String mes;
    public int anyo;
    public String diaSemana;
    public boolean festivo;

    public Fecha (int dia, String mes, int anyo, boolean festivo) {
        this.dia = dia;
        this.mes = mes;
        this.anyo = anyo;
        this.festivo = festivo;

    public int anyoSiguiente () {
        return anyo+1;} }
```

Ejemplo – acceso métodos y atributos

Fecha.java

```
public class Fecha {
    public int dia;
    public String mes;
    public int anyo;
    public String diaSemana;
    public boolean festivo;

    public Fecha (int dia, String mes, int anyo, boolean festivo) {
        this.dia = dia;
        this.mes = mes;
        this.anyo = anyo;
        this.festivo = festivo;

    public int anyoSiguiente () {
        return anyo+1;} }
```

ProgramaPrincipal.java

```
public class  ProgramaPrincipal {
     public static void main (String [] args) {
         Fecha miFecha= new Fecha(19, ''Mayo'', 2015, false);
         int anyo = miFecha.anyo();
         int anyoSig = miFecha.anyoSiguiente(); }
```

Modificadores de acceso (<visibilidad>)

- Por defecto, los atributos y métodos son accesibles (visibles) dentro del paquete en el que se define la clase
- Para limitar este acceso se pueden utilizar modificadores de acceso
  - `public`
  - `private`

Modificadores de acceso (<visibilidad>)

- Modificador **public**
  - En una clase, significa que es accesible fuera del paquete donde está definida. Para acceder a ella hay que importarla mediante `import`
  - En un atributo o método, significa que es accesible donde lo sea la clase. Si no se pone nada el acceso es `public` por defecto

Modificadores de acceso (<visibilidad>)

- Modificador **private**
  - Sólo en atributos y métodos
  - En atributos
    - No permite que se acceda a ellos desde fuera de la clase ni para leer ni para modificar el atributo
    - El acceso a atributos privados se puede realizar mediante métodos públicos
  - En métodos impide invocarlos desde fuera de la clase

¿Por qué existen los modificadores de acceso?

¿Por qué existen los modificadores de acceso?

Encapsulación

Características de la POO

- Reutilización
- Organización del código
- Específicas de la POO
  - Encapsulación
  - Herencia
  - Polimorfismo

Ejemplo – modificador `private` en atributos

```
public class SerieAritmetica {
```

Ejemplo – modificador `private` en atributos

```
public class SerieAritmetica {

     private int a1; // primer término
     private int inc;//incremento
```

Ejemplo – modificador `private` en atributos

```
public class SerieAritmetica {

     private int a1; // primer término
     private int inc;//incremento

     SerieAritmetica (int a1, int inc){
         this.a1 = a1;
         this.inc = inc; }
```

Ejemplo – modificador `private` en atributos

```
public class SerieAritmetica {

     private int a1; // primer término
     private int inc;//incremento

     SerieAritmetica (int a1, int inc){
         this.a1 = a1;
         this.inc = inc; }

     public int Suma (int n){
         int suma = 0;
         for(int i = 0; i< n; i++)
               suma += a1 + i * inc;
         return suma;}

     }
```

Métodos comunes – acceso a atributos privados

- Método **set**: para dar valor a un atributo privado
- Método **get**: para obtener el valor de un atributo privado

Ejemplo – métodos `set` y `get`

```
public class SerieAritmetica {
     private int a1; // primer término
     private int inc;//incremento

     SerieAritmetica (int a1, int inc){
         this.a1 = a1;
         this.inc = inc; }
```

Ejemplo – métodos `set` y `get`

```
public class SerieAritmetica {
     private int a1; // primer término
     private int inc;//incremento

     SerieAritmetica (int a1, int inc){
         this.a1 = a1;
         this.inc = inc; }


      /* EJEMPLO set */
     public void seta1 (int a1){
         this.a1 = a1;}
```

Ejemplo – métodos `set` y `get`

```
public class SerieAritmetica {
     private int a1; // primer término
     private int inc;//incremento

     SerieAritmetica (int a1, int inc){
         this.a1 = a1;
         this.inc = inc; }


      /* EJEMPLO set  */
     public void seta1 (int a1){
         this.a1 = a1;}

      /* EJEMPLO get */
     public int geta1 (){
         return a1;}

     }
```

Métodos comunes – escribir un objeto en un String

- Método **toString**: para escribir un objeto en un `String`
- Se suele utilizar para imprimir el objeto en un formato adecuado

Ejemplo – métodos `toString`

```
public class SerieAritmetica {
     private int a1; // primer término
     private int inc;//incremento

     SerieAritmetica (int a1, int inc){
         this.a1 = a1;
         this.inc = inc; }


      /* EJEMPLO toString */
     public String toString (){
         String string = "";
         for(int i = 0; i< n; i++)
               string += a1 + i * inc+ " ";
         return string;}

     }
```

Métodos comunes – comparación de objetos

- Método **equals**: para determinar si dos objetos son iguales

Ejemplo – métodos `equals`

```
public class SerieAritmetica {
     private int a1; // primer término
     private int inc;//incremento

     SerieAritmetica (int a1, int inc){
         this.a1 = a1;
         this.inc = inc; }

      /* EJEMPLO equals */
     public boolean equals (SerieAritmetica otraSerie){
            if (a1 == otraSerie.a1 && inc== otraSerie.inc)
                  return true;
            return false;}
    }
```

Parte VIII

Clases útiles (Scanner, String, Envoltorios y Math

Algunas clases útiles de la librería de Java

Este documento hace un resumen somero de cuatro clases útiles de la librería de Java (`Scanner`, `String`, Envoltorios y `Math`). Sólo se comentan algunos de los métodos más interesantes de estas clases, para un análisis más detallado se recomienda visitar la API de Java.

1. Clase `Scanner`

Esta clase se usa entre otras cosas para que el usuario pueda introducir datos por el teclado. Para usarla hay que importarla desde la librería `java.util`. A continuación se muestra un ejemplo de código con su uso (se omite la declaración de la clase y del método main):

```
import java.util.Scanner;
...
Scanner s = new Scanner(System.in);
int b=0;
while (!s.hasNextInt())
      s.next();
b= s.nextInt();
System.out.println(b);
```

Por defecto los datos se separan mediante un espacio. Se puede cambiar el carácter separador, por ejemplo para cambiarlo por Enter usamos el método: `<objeto scanner>.useDelimiter(System.getProperty("line.separator"));`

Se pueden usar otros delimitadores e incluso combinaciones de ellos por medio de expresiones regulares. En internet se pueden encontrar más ejemplos.

También por defecto, si el teclado del ordenador está en español, se usa la coma en lugar del punto para los números decimales. Si queremos que el usuario introduzca los datos usando el punto utilizamos (hay que importar la clase `Locale` de `java.util`): `<objeto scanner>.useLocale(Locale.ENGLISH);`

Algunos de los métodos más interesantes de esta clase son:

Nombre método	Descripción
`int nextInt()`	Devuelve un valor de tipo `int` que el usuario debe introducir por teclado. Hay un método para cada tipo básico, excepto para `char`. Ej. `nextDouble()`, `nextBoolean()`...
`String next()`	Devuelve el siguiente dato introducido por el usuario en forma de `String`
`String nextLine()`	Devuelve todo lo que ha introducido el usuario, independientemente de cuál sea el separador (da problemas al usarse, mejor no utilizarlo)
`boolean hasNext()`	Devuelve `true` si hay algún dato listo para ser leído
`boolean hasNextInt()`	Devuelve `true` si lo siguiente que va a leer es un `int` (si lo siguiente que ha introducido el usuario es un `int`). Hay un método similar para cada tipo básico, excepto `char`. Ej. `hasNextDouble()`, `hasNextBoolean()`...

2. Clase `String`

String, además de comportarse como un tipo básico es una clase, por lo que tiene métodos que se pueden utilizar para hacer operaciones con cadenas. Para llamar a los métodos se pone <variable de tipo String>.metodo

Nombre método	Descripción
`char charAt(int index)`	Devuelve el carácter que está en esa posición
`int compareTo(String anotherString)`	Devuelve 0 si ambas cadenas son iguales, un valor negativo si la cadena es anterior alfabéticamente que el argumento y un valor positivo si es mayor. Si son diferentes devuelve la diferencia de código ASCII entre las dos primeras letras en que se diferencian. Si solo se diferencian en que son de distinta longitud lo que devuelve es la diferencia en la longitud. ¡Ojo la ñ no va después de la n!
`int compareToIgnoreCase(String str)`	Igual al anterior ignorando la diferencia entre mayúsculas y minúsculas
`boolean contains (charSequence s)`	Devuelve `true` si la cadena contiene a la subcadena
`boolean endsWith(String suffix)`	Devuelve `true` si la cadena acaba de esa forma. Hay otro equivalente si la cadena empieza de esa forma (`startsWith`)
`boolean equals (String str)` `boolean equalsIgnoreCase(String str)`	Devuelve verdadero si las dos cadenas son iguales, en el segundo caso ignorando mayúsculas y minúsculas.
`int indexOf(String str)` `int indexOf(String str, int ind)`	Devuelve un entero con la posición en la que aparece el carácter o la subcadena por primera vez (-1 si no existe). La segunda versión empieza a buscar desde un lugar determinado. También hay 2 versiones equivalentes buscando de atrás hacia delante (`lastIndexOf`)
`int length()`	Devuelve la longitud de la cadena
`String replace (String , String)`	Reemplaza todas las ocurrencias de una subcadena por otra. También se puede usar `replaceAll` o `replaceFirst`. La primera se comporta igual pero además de recibir una cadena puede recibir una expresión regular. La segunda, también puede recibir expresiones regulares, y solo cambia la primera ocurrencia de la subcadena.
`String[ ] split (String regex)`	Devuelve un array de `String` resultado de partir la cadena usando como separador el argumento (cadena o expresión regular). Ej. `"hola como estás".split(" ")` ␣ `new String[] {"hola","como","estás"}`

`String substring(int beginIndex, int endIndex)` `String substring(int beginIndex)`	Devuelve la subcadena que empieza en el índice pasado por parámetro (incluido), si solo se le da un parámetro devuelve desde el índice hasta el final de la cadena original, si tiene dos devuelve entre los dos índices, con el primero incluido y el segundo excluido.
`String toLowerCase()`	Convierte la cadena a minúsculas, y `toUpperCase` a mayúsculas.
`static String valueOf(tipo básico o char [])`	Convierte el tipo básico que se le pase a `String`
`String trim ()`	Elimina espacios antes y después

Los métodos no cambian el valor de la cadena (si hacemos por ejemplo un `cadena.toLowerCase()`, la cadena original no cambia).

En `replaceAll`, `replaceFirst` y `split` los siguientes caracteres no se pueden reemplazar directamente: `$ ^ . * + ? [ ] ()`. Hay que poner `\\?` y similares (la razón es que no se busca una cadena sino una expresión regular)

3. Envoltorios

Los envoltorios se utilizan cuando queremos guardar datos de tipos básicos como si fueran objetos (hay situaciones en las que Java precisa que el dato esté en forma de objeto y no admite tipos básicos). Para cada tipo básico existe un envoltorio:

Tipo básico	**Envoltorio**
`byte`	`Byte`
`short`	`Short`
`int`	`Integer`
`long`	`Long`
`float`	`Float`
`double`	`Double`
`char`	`Character`
`boolean`	`Boolean`

En cualquier caso Java es capaz de empaquetar/desempaquetar automáticamente los datos, de forma que si en algún sitio se necesita un tipo objeto y en lugar de ello se proporciona un tipo básico, se convierte automáticamente el básico a su correspondiente envoltorio y viceversa.

Además, los envoltorios permiten convertir de String al tipo básico correspondiente, para ello tienen un método `parse`, (`parseInt()`, `parseBoolean()`, `parseFloat()` …) que funciona como en el siguiente código:

```
String s = "33";
int a = Integer.parseInt(s); //a vale 33
```

Si el valor que hay en la cadena no se puede convertir al tipo de destino, tenemos un error de ejecución

4. Clase `Math`

Clase especial que contiene funciones y constantes matemáticas.

Se usa poniendo `Math.<metodo>`

Atributos: `Math.E` y `Math.PI`

Nombre método	Descripción
`static int abs(int a)`	Devuelve el valor absoluto del número pasado como parámetro. También se puede usar con cualquier otro tipo numérico (si le damos un `double` devolverá un `double`, etc.)
`static long round (double a)`	Redondea el número pasado como parámetro. Si el número es `double` devuelve `long`, si es `float` devuelve `int`.
`static double ceil (double a)`	Trunca el número hacia arriba (`Math.ceil(3.2)` devuelve `4.0`), ¡ojo devuelve un `double`!
`static double floor (double a)`	Trunca el número, ¡ojo devuelve un `double`!
`static double sin (double a)`	Devuelve el seno del ángulo `a` (`a` debe estar en radianes). También hay `cos`, `tan`, `asin`, `acos`, `atan`, `sinh`, `cosh`, `tanh`
`static int max (int a, int b)`	Devuelve el máximo de los dos números. También hay versiones para los otros tipos numéricos. También existe `min (int a, int b)`.
`static double log (double a)`	Devuelve el logaritmo neperiano de `a`, para el logaritmo decimal se usa `log10(double a)`
`static double pow (double a, double b)`	Eleva `a` a `b`
`static double exp (double a)`	Eleva el número `e` a `a`
`static double sqrt (double a)`	Raíz cuadrada
`static double cbrt (double a)`	Raíz cúbica
`static double random()`	Devuelve un número aleatorio entre `0.0` (incluido) y `1.0` (no incluido)

Parte IX

Más POO - Herencia y Polimorfismo

Programación

PLG
Planning and Learning Group

Universidad Carlos III de Madrid

Curso 2015-2016

Más POO: herencia y polimorfismo

Concepto de herencia

Herencia

Concepto de herencia

Herencia

- Las propiedades pasan de padres a hijos
- Relación SUBCLASE-DE y SUPERCLASE-DE
- Ejemplos

¿Por qué herencia en programas?

Reutilización y centralización de código

¿Por qué herencia en programas?

Reutilización y centralización de código

- Evita duplicidad
- Disminuye volumen de código
- Facilita mantenimiento
- Permite el polimorfismo

Herencia en Java

- Clases derivadas de otras clases que heredan sus atributos y métodos
- Se utiliza la palabra reservada `extends`

```
class Subclase extends SuperClase {
      ...
}
```

Modificador protected

```
class A {
      public int a;
      private int b;
      protected int c;
}

class B extends A {
      ... }
```

Modificador protected

```
class A {
      public int a;
      private int b;
      protected int c;
}

class B extends A {
      ... }
```

- `a` es público: accesible desde fuera de la clase y el paquete. Visible para todos, incluída la clase hija
- `b` es privado: no visible desde fuera de la clase, ni desde sus clases hijas
- `c` es protegido: accesible desde dentro del paquete. Además pasa a las clases hijas: la clase B dispone de ese atributo

Resumen restricciones de acceso

VISIBILIDAD	public	protected	nada	private
Propia clase	SI	SI	SI	SI
Mismo paquete	SI	SI	SI	NO
Otro paquete	SI	NO	NO	NO
Subclase en paquete	SI	SI	SI	NO
Subclase en otro paquete	SI	SI	NO	NO

super y super()

- La palabra reservada `this` sirve para indicar la propia clase
- La palabra reservada `super` sirve para hacer referencia a la superclase de la clase en que se utiliza

super y super()

```
class A {
      int a;
      int b;
}

class B extends A {
      String b;
}
```

super y super()

```
class A {
      int a;
      int b;
}

class B extends A {
      String b;
}
```

- Si estamos en B, `super.b` es el atributo `b` de `A`

super y super()

- Si en A hay un método `mostrarDatos()` que muestra `a` y `b`,
- y en B reimplementamos ese método para que muestre también el atributo `b` de B:

```
class B {
      ...

      void mostrarDatos () {
          super.mostrarDatos();
          System.out.println(b);
      }

}
```

super y super()

- `super()` invoca al constructor de la superclase

```
class Persona {
   String nombre;
   byte edad;
   double altura;

   Persona (String nombre, byte edad, double altura) {
      this.nombre = nombre;
      this.edad = edad;
      this.altura = altura;
   }
}
```

super y super()

- `super()` invoca al constructor de la superclase

```
class Persona {
   String nombre;
   byte edad;
   double altura;

   Persona (String nombre, byte edad, double altura) {
      this.nombre = nombre;
      this.edad = edad;
      this.altura = altura;
   }
}

class Empleado extends Persona {
   double salario;
```

super y super()

- `super()` invoca al constructor de la superclase

```
class Persona {
   String nombre;
   byte edad;
   double altura;

   Persona (String nombre, byte edad, double altura) {
      this.nombre = nombre;
      this.edad = edad;
      this.altura = altura;
   }
}

class Empleado extends Persona {
   double salario;

   Empleado (String nombre, byte edad, double altura, double salario)
      super(nombre, edad, altura);
      this.salario= salario;
   }
}
```

Sustitución de métodos y atributos

- Las clases heredan de sus superclases todos los métodos y atributos no privados
- En la subclase se pueden redefinir tanto atributos cómo métodos
  - Utilizando el mismo nombre en atributos
  - En métodos, utilizando el mismo nombre, mismo valor de retorno y mismos parámetros
- Cuando se redefine un atributo o un método, el nuevo oculta al heredado

Sustitución de métodos y atributos

Clases A y B

```
class A {
      int a = 3;
}

class B extends A {
      double a = 2.5;
}
```

Programa principal u otra clase

```
B objetoB = new B();
System.out.println(objetoB.a);
```

Sustitución de métodos y atributos

Clases A y B

```
class A {
      int a = 3;
}

class B extends A {
      double a = 2.5;
}
```

Programa principal u otra clase

```
B objetoB = new B();
System.out.println(objetoB.a); // mostrará por pantalla 2.5
```

Sustitución de métodos y atributos

Clases A y B

```
class A {
     void mostrar() {
         System.out.println("Soy una A");
     }
}

class B extends A {
}
```

Programa principal

```
B objetoB = new B();
objetoB.mostrar();
```

Sustitución de métodos y atributos

Clases A y B

```
class A {
      void mostrar() {
          System.out.println("Soy una A");
      }
}

class B extends A {
}
```

Programa principal

```
B objetoB = new B();
objetoB.mostrar(); // mostrará por pantalla soy una A
```

Sustitución de métodos y atributos

Clases A y B

```
class A {
    void mostrar() {
        System.out.println("Soy una A");
    }
}

class B extends A {
    void mostrar() {
        System.out.println("Soy una B");
    }
}
```

Programa principal u otra clase

```
B objetoB = new B();
objetoB.mostrar();
```

Sustitución de métodos y atributos

Clases A y B

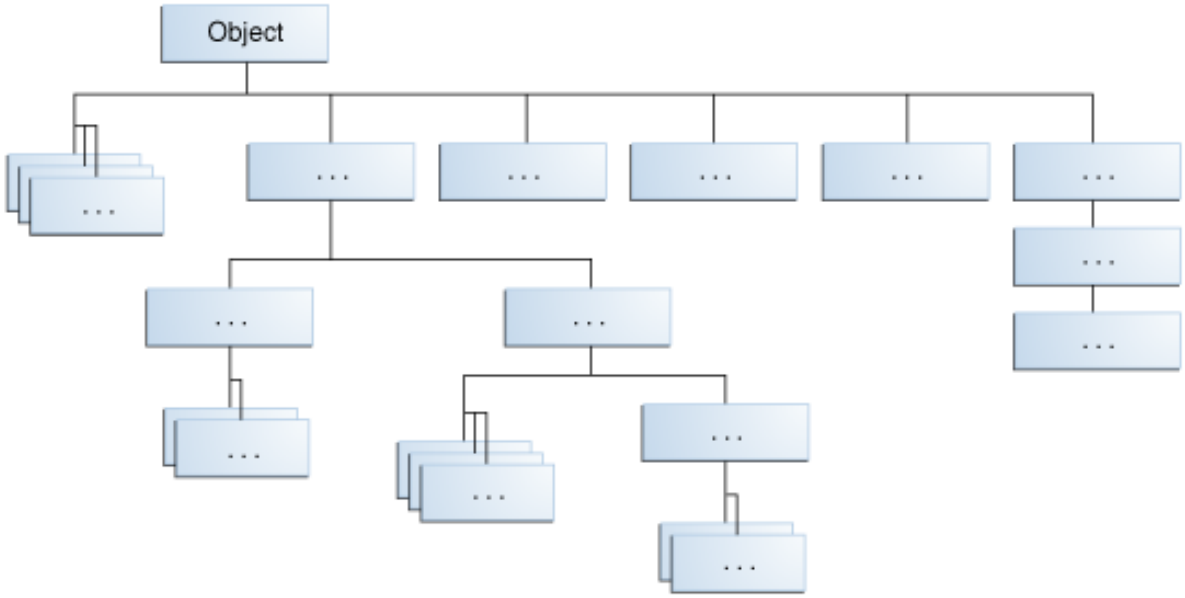
```
class A {
      void mostrar() {
          System.out.println("Soy una A");
      }
}

class B extends A {
      void mostrar() {
          System.out.println("Soy una B");
      }
}
```

Programa principal u otra clase

```
B objetoB = new B();
objetoB.mostrar(); // mostrará por pantalla soy una B
```

Clase `Object`

En Java todo objeto hereda implícitamente de la clase `Object`



- Esta clase tiene métodos que pueden ser redefinidos, por ejemplo
  - `clone`: crear una copia del objeto
  - `equals`: determinar si dos objetos son iguales
  - `toString`: devuelve un `String` que representa al objeto

Concepto de polimorfismo

Polimorfismo

Concepto de polimorfismo

Polimorfismo

- Definición de la RAE: *Cualidad de lo que tiene o puede tener distintas formas*

Concepto de polimorfismo en POO

- Hemos visto que en Java el contenido de las variables debe ser del tipo del que fueron declaradas
- Pero ...
- cuando hay herencia, ¡una variable del tipo de la Superclase puede contener también cualquier objeto del tipo de una Subclase!
- Esto se denomina polimorfismo en POO

Concepto de polimorfismo en POO

Clase Persona y subclase Empleado

```
class Persona {
    ... }

class Empleado extends Persona {
    ... }
```

Programa principal u otra clase

```
Persona persona1 = new Persona();
Empleado empleado1 = new Empleado();
```

Concepto de polimorfismo en POO

Clase Persona y subclase Empleado

```
class Persona {
    ... }

class Empleado extends Persona {
    ... }
```

Programa principal u otra clase

```
Persona persona1 = new Persona();
Empleado empleado1 = new Empleado();
Persona persona1 = new Empleado();
```

Selección dinámica de métodos

Una de las herramientas más potentes que el polimorfismo proporciona a Java es la selección dinámica de métodos en tiempo de ejecución

Selección dinámica de métodos

Clases A y B

```
class A {
     void mostrar() {
         System.out.println("Soy una A");
     }
}
class B extends A {
     void mostrar() {
         System.out.println("Soy una B");
     }
}
class C extends B {
     void mostrar() {
         System.out.println("Soy una C");
     }
}
```

Programa principal u otra clase

```
A variableA;
variableA = new A();
variableA.mostrar();
```

Selección dinámica de métodos

Clases A y B

```
class A {
     void mostrar() {
         System.out.println("Soy una A");
     }
}
class B extends A {
     void mostrar() {
         System.out.println("Soy una B");
     }
}
class C extends B {
     void mostrar() {
         System.out.println("Soy una C");
     }
}
```

Programa principal u otra clase

```
A variableA;
variableA = new A();
variableA.mostrar(); // mostrará por pantalla Soy una A
```

Selección dinámica de métodos

Clases A y B

```
class A {
     void mostrar() {
         System.out.println("Soy una A");
     }
}
class B extends A {
     void mostrar() {
         System.out.println("Soy una B");
     }
}
class C extends B {
     void mostrar() {
         System.out.println("Soy una C");
     }
}
```

Programa principal u otra clase

```
A variableA;
variableA = new A();
variableA.mostrar(); // mostrará por pantalla Soy una A
variableA = new B();
variableA.mostrar();
```

Selección dinámica de métodos

Clases A y B

```
class A {
     void mostrar() {
         System.out.println("Soy una A");
     }
}
class B extends A {
     void mostrar() {
         System.out.println("Soy una B");
     }
}
class C extends B {
     void mostrar() {
         System.out.println("Soy una C");
     }
}
```

Programa principal u otra clase

```
A variableA;
variableA = new A();
variableA.mostrar(); // mostrará por pantalla Soy una A
variableA = new B();
variableA.mostrar(); // mostrará por pantalla Soy una B
```

Selección dinámica de métodos

Clases A y B

```
class A {
     void mostrar() {
         System.out.println("Soy una A");
     }
}
class B extends A {
     void mostrar() {
         System.out.println("Soy una B");
     }
}
class C extends B {
     void mostrar() {
         System.out.println("Soy una C");
     }
}
```

Programa principal u otra clase

```
A variableA;
variableA = new A();
variableA.mostrar(); // mostrará por pantalla Soy una A
variableA = new B();
variableA.mostrar(); // mostrará por pantalla Soy una B
variableA = new C();
variableA.mostrar();
```

Selección dinámica de métodos

Clases A y B

```
class A {
     void mostrar() {
         System.out.println("Soy una A");
     }
}
class B extends A {
     void mostrar() {
         System.out.println("Soy una B");
     }
}
class C extends B {
     void mostrar() {
         System.out.println("Soy una C");
     }
}
```

Programa principal u otra clase

```
A variableA;
variableA = new A();
variableA.mostrar(); // mostrará por pantalla Soy una A
variableA = new B();
variableA.mostrar(); // mostrará por pantalla Soy una B
variableA = new C();
variableA.mostrar(); // mostrará por pantalla Soy una C
```

Selección dinámica de métodos

¡Se ejecutan distintos métodos según el tipo de objeto referenciado!

Clases abstractas

- Jerarquía de herencia de clases
  - Cuánto más abajo en la jerarquía, más específica es la clase
  - Cuanto más arriba, más general es la clase
- Si la clase de arriba es muy general puede que haya métodos que no se puedan implementar sin conocer la subclase
- Ejemplo: juego piedra, papel tijera
  - Clase Jugador, subclases JugadorEsCiclica, JugadorEsAleatoria
  - El método `Eleccion` no se puede implementar sin conocer el tipo de jugador

Clases abstractas

```
abstract class Jugador {
        int puntuacion;
        ...
        int getPuntuacion() {
             return puntuacion;
        }
        abstract int eleccion();
        // El método elección no se implementa, sólo se
        // declara como abstracto para ser implementado en las subclases
}
```

- Las clases con métodos abstractos deben ser declaradas como abstractas
- ¡No se pueden crear objetos del tipo de una clase abstracta!

Clases abstractas

```
class JugadorEsCiclica extends Jugador {
        ...
        int eleccion() {
        // < Implementación del método aquí>
        }
}

class JugadorEsAleatoria extends Jugador {
        ...
        int eleccion() {
        // < Implementación del método aquí>
        }
}
```

Clases abstractas

```
class JugadorEsCiclica extends Jugador {
        ...
        int eleccion() {
        // < Implementación del método aquí>
        }
}

class JugadorEsAleatoria extends Jugador {
        ...
        int eleccion() {
        // < Implementación del método aquí>
        }
}
```

Programa Principal

```
Jugador jugador1 = new JugadorEsCiclica();
Jugador jugador2 = new JugadorEsAleatoria();
int eleccion1 = jugador1.eleccion();
int eleccion2 = jugador2.eleccion();
System.out.println(jugador1.getPuntuacion());
System.out.println(jugador2.getPuntuacion());
```

Parte X

Algoritmos sobre listas

Programación

PLG
Planning and Learning Group

Universidad Carlos III de Madrid

Algoritmos sobre Listas

Complejidad computacional

- Para resolver un problema unos algoritmos son mejores que otros
- Complejidad computacional: mide los recursos (tiempo, memoria) requeridos por un algoritmo
- Complejidad de un algoritmo: la del mejor algoritmo descubierto para resolverlo
- Notación BigO (peor caso)

Complejidad computacional – complejidad temporal (idea)

- Cuantas más instrucciones de CPU necesite un algoritmo más tardará
- Idea: Contar instrucciones de CPU
  - Asignar un valor a una variable
  - Comparar dos valores
  - Realizar una operación aritmética
  - Etc.
- Observación: habitualmente el número de instrucciones de CPU depende del tamaño de la entrada

Complejidad computacional – complejidad temporal (idea)

- Imaginemos un programa que encuentra el máximo elemento de un array de tamaño n

```
int max = A[0];
for (int i = 0; i < n; i++){
  if (A[i] > max)
     max = A[i]
}
```

- Si n es más grande el programa ejecuta más instrucciones
- La idea es determinar como aumenta el número de instrucciones, $f(n)$, a medida que crece n

Complejidad computacional – complejidad temporal (idea)

- No se trata de contar el número de instrucciones, sino de fijarse en lo que hace que $f(n)$ aumente rápidamente: bucles
- Se considera una cota superior (ajustada) de $f(n)$ para el peor caso
- Esto se denomina determinar el comportamiento asintótico del programa. Notación $O()$

Complejidad computacional – complejidad temporal (idea)

- $f(n) = 5n + 12$, el comportamiento asíntotico es $O(n)$
- $f(n) = 3n^2 + 5n + 12$, el comportamiento asíntotico es $O(n^2)$
- $f(n) = 3^n + 5n + 12$, el comportamiento asíntotico es $O(3^n)$

Complejidad computacional – complejidad temporal (idea)

- En un programa sin bucles es $O(1)$ (constante)
- En un programa con un bucle que puede recorrer completamente un array, como el del ejemplo anterior, es $O(n)$
- Si el array se recorre de nuevo por cada elemento (dos bucles anidados) es $O(n^2)$
- Si un algoritmo es $O(n^2)$ y para aplicarlo a 1000 elementos se tarda 3 segundos, para 2000 elementos se tardará $3 \times 2^2 = 12$ segundos y para 3000 serán $3 \times 3^2 = 27$ segundos, etc.

Ejemplos

- Extracción del elemento con un determinado índice de una array:

Ejemplos

- Extracción del elemento con un determinado índice de una array: $O(1)$

Ejemplos

- Extracción del elemento con un determinado índice de una array: $O(1)$
- Buscar una palabra en un diccionario:

Ejemplos

- Extracción del elemento con un determinado índice de una array: $O(1)$
- Buscar una palabra en un diccionario:
  - Elementos restantes
    - Iteración 0: n
    - Iteración 1: $n/2$
    - Iteración 3: $n/4$
    - ...
    - Iteración i: $n/2^i$
  - ¿En qué iteración estamos cuando queda 1 elemento?: $1 = n/2^i$
  - $i = log_2(n)$
  - Luego es $O(log\, n)$

Listas

- Diferentes tipos:
  - Ordenadas o no ordenadas
  - De longitud fija o dinámica
  - Con elementos repetidos o sin elementos repetidos

Algoritmos sobre Listas

- Vamos a trabajar sobre arrays (longitud fija)
- Tres tipos de algoritmos:
  - Búsqueda: buscar un elemento en una lista
  - Ordenación: ordenar una lista
  - Inserción: insertar un elemento en una lista

En este tema

Algoritmos sobre Listas

- Algoritmos de Búsqueda en Listas
- Algoritmos de Ordenación de Listas
- Algoritmos de Inserción en Listas

Algoritmos de búsqueda

- Búsqueda secuencial
  - Explicación y ejemplo
  - O(n)
  - Si la lista está ordenada tarda menos

Algoritmos de búsqueda

- Búsqueda secuencial
  - Explicación y ejemplo
  - O(n)
  - Si la lista está ordenada tarda menos
- Búsqueda binaria
  - Sólo para listas ordenadas
  - Explicación y ejemplo
  - O(log n)

En este tema

Algoritmos sobre Listas

- Algoritmos de Búsqueda en Listas
- Algoritmos de Ordenación de Listas
- Algoritmos de Inserción en Listas

Ordenación de Listas

- El criterio de ordenación determina el orden de los elementos. ej (“>” para números, orden alfabético para palabras)
- Algoritmos de Ordenación:
  - de ordenación interna: la ordenación se hace sobre la misma lista, sin utilizar ninguna otra auxiliar
  - de ordenación externa: se utiliza una lista auxiliar para realizar la ordenación

Métodos de Ordenación Interna

- Utilizamos la misma lista en el proceso de ordenación
- Se intercambian unos elementos por otros
- Algoritmos más conocidos:
  1. Burbuja, $O(n^2)$
     - `https://www.youtube.com/watch?v=JP5KkzdUEYI`

Métodos de Ordenación Interna

- Utilizamos la misma lista en el proceso de ordenación
- Se intercambian unos elementos por otros
- Algoritmos más conocidos:
  1. Burbuja, $O(n^2)$
     - `https://www.youtube.com/watch?v=JP5KkzdUEYI`
  2. Inserción directa, $O(n^2)$
     - `https://courses.cs.vt.edu/csonline/Algorithms/Lessons/InsertionCardSort/insertioncardsort.swf`

Métodos de Ordenación Interna

- Utilizamos la misma lista en el proceso de ordenación
- Se intercambian unos elementos por otros
- Algoritmos más conocidos:
  1. Burbuja, $O(n^2)$
     - `https://www.youtube.com/watch?v=JP5KkzdUEYI`
  2. Inserción directa, $O(n^2)$
     - `https://courses.cs.vt.edu/csonline/Algorithms/Lessons/InsertionCardSort/insertioncardsort.swf`
  3. Selección directa, $O(n^2)$
     - `https://courses.cs.vt.edu/csonline/Algorithms/Lessons/SelectionCardSort/selectioncardsort.swf`

Métodos de Ordenación Interna

- Utilizamos la misma lista en el proceso de ordenación
- Se intercambian unos elementos por otros
- Algoritmos más conocidos:
  1. Burbuja, $O(n^2)$
     - `https://www.youtube.com/watch?v=JP5KkzdUEYI`
  2. Inserción directa, $O(n^2)$
     - `https://courses.cs.vt.edu/csonline/Algorithms/Lessons/InsertionCardSort/insertioncardsort.swf`
  3. Selección directa, $O(n^2)$
     - `https://courses.cs.vt.edu/csonline/Algorithms/Lessons/SelectionCardSort/selectioncardsort.swf`
  4. Otros más eficientes y también más complicados: QuickSort, $O(n \log n)$

En este tema

Algoritmos sobre Listas

- Algoritmos de Búsqueda en Listas
- Algoritmos de Ordenación de Listas
- Algoritmos de Inserción en Listas

Inserción en listas

- Lista ordenada: inserción ordenada de un elemento
- Lista no ordenada: inserción de un elemento en una posición
- Tipos de listas
  - Longitud fija: array con posiciones suficientes. Sólo las *n* primeras se consideran ocupadas
  - Longitud variable: se podría redimensionar el array, pero en realidad ésto se hace con otras estructuras de datos
- Algoritmo: insertar en el lugar correspondiente y desplazar los siguientes elementos en una posición