Introducción a la programación - Transparencias

Licencia

Índice

Introducción a la programación

Fundamentos de la programación

Tema 1

- Introducción
- Problemas, Algoritmos y Programas
- Paradigmas y Lenguajes de Programación
- Ingeniería del Software

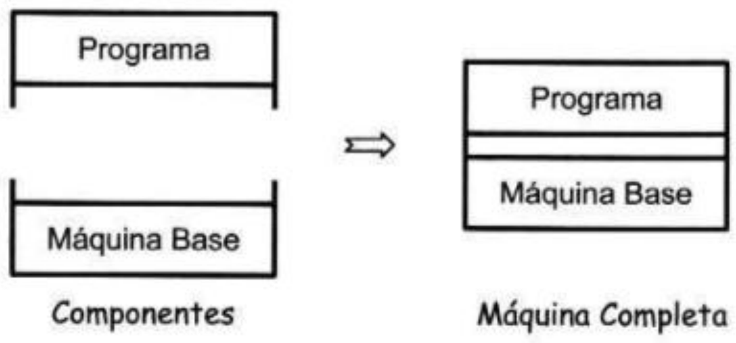
Objetivo

- Exponer los conceptos clave para la resolución de problemas por medio de sistemas computacionales

- **Máquina**
  - – Dispositivo o instrumento capaz de realizar un cierto trabajo u operación
  - – Un proceso de funcionamiento por el cual diferentes operaciones se van sucediendo a lo largo del tiempo sucesiva o simultáneamente
  - – Atendiendo a su control
    - Manuales (p.e.: máquina de escribir)
      - Operador o agente externo invoca operaciones
    - Automáticas (p.e: ascensor)
      - Actúan por si solas, pudiendo responder a estímulos externos

Máquinas Programables

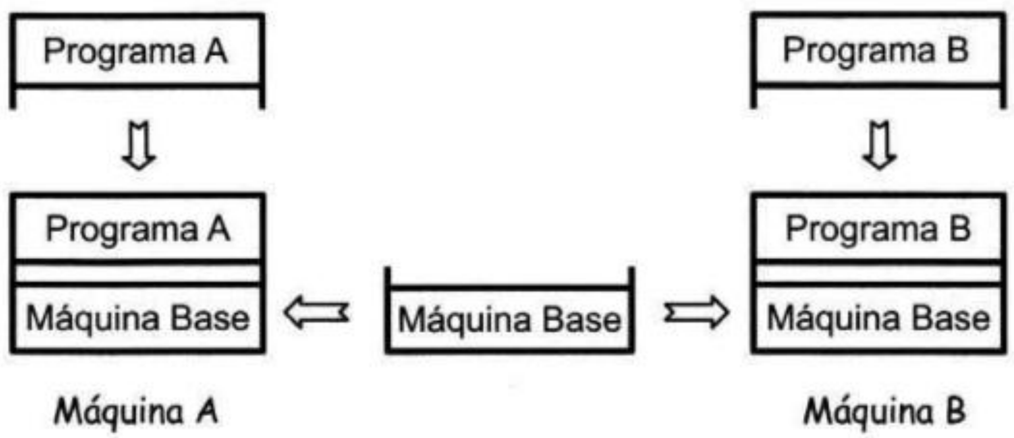
- Máquinas automáticas
  - Fijas
  - Programables



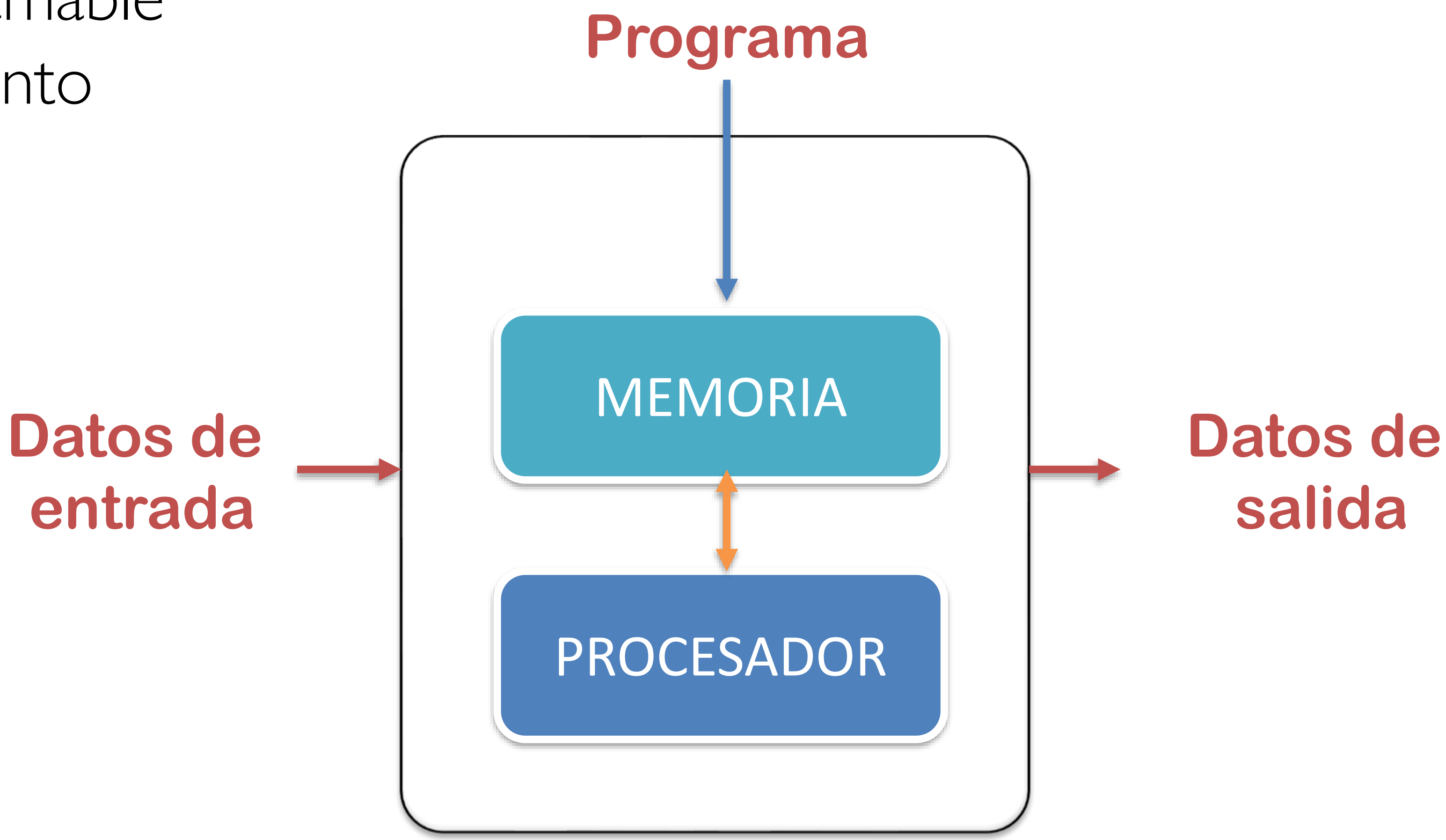
- Ejemplo
  - Piano: máquina manual para reproducir música
  - Caja de música: máquina automática para producir música
  - Reproductor MP3: máquina automática para producir música

Máquinas Programables

- Dependiendo del programa suministrado, la máquina se comporta como diferentes máquinas



- La máquina programable por excelencia es el ordenador (antes Computador):
  - Máquina programable para el tratamiento de la información



Computadores

- Para realizar un determinado tratamiento de la información necesitamos
  - Construir la máquina base → Hardware
  - Idear y desarrollar el programa → Software
  - Ejecutar dicho programa en el ordenador (dispositivo)

Computadores

- Para realizar un determinado tratamiento de la información necesitamos
  - Construir la máquina base → Hardware
  - Idear y desarrollar el programa → Software
  - Ejecutar dicho programa en el ordenador (dispositivo)

La labor de desarrollar programas recibe habitualmente el nombre de programación

Introducción

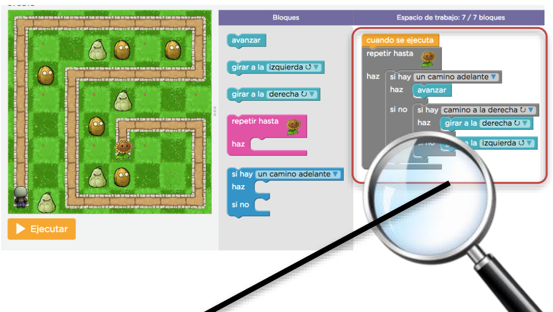
- ¿Qué es la programación?
  - Es un proceso mediante el cual se codifican una serie de **instrucciones** en un **lenguaje** determinado para ser decodificados y ejecutados por un sistema computacional, con el fin de resolver un problema o llevar a cabo una función específica.
  - Definir lo que debe hacer el ordenador para resolver un problema concreto, utilizando un lenguaje de programación.

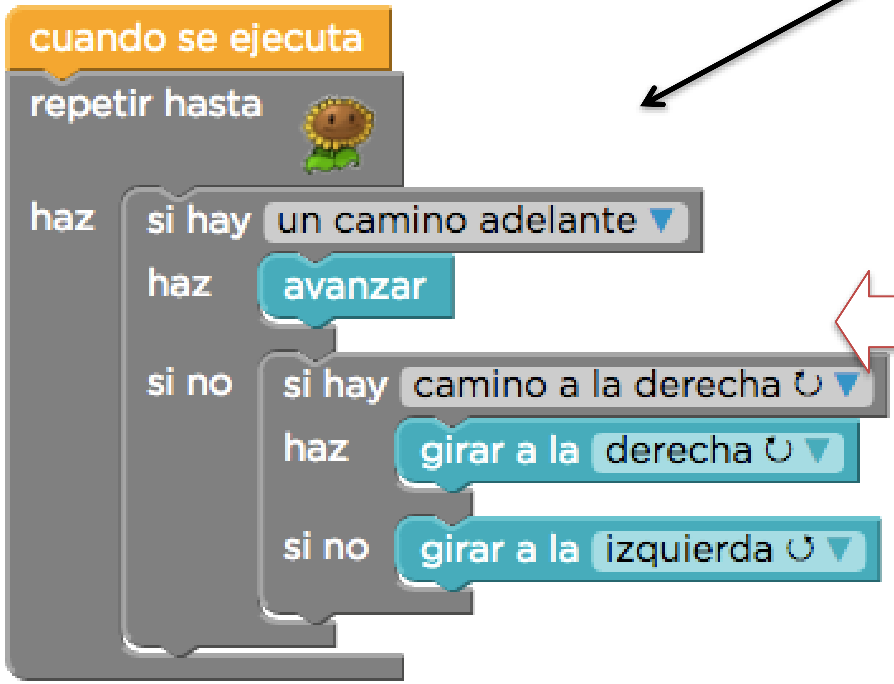
Introducción

- Ejemplo

Introducción

- Ejemplo





Incluso las mejores universidades enseñan programación basada en bloques (por ejemplo, **Berkeley**, **Harvard**). Pero, por debajo, los bloques que has programado también se pueden mostrar en JavaScript, el lenguaje de programación más utilizado en el mundo:

```
while (notFinished()) {
  if (isPathForward()) {
    moveForward();
  } else {
    if (isPathRight()) {
      turnRight();
    } else {
      turnLeft();
    }
  }
}
```

```
88 94 50 FF 76 0A FF 76 08 9A BA CD 3A 16 B8 01
00 EB E8 B8 88 94 50 2B C0 50 9A FA C5 3A 16 EB
ED B8 88 94 50 B8 01 00 EB EF B8 88 94 50 9A 48
D1 3A 16 EB D9 5D CA 0A 00 55 8B EC 83 EC 08 57
56 B8 01 00 50 9A 97 41 9B 34 8B D8 8B 47 14 89
```

Código Máquina

```cpp
void PintarPlazas(const TipoPlazas P) {

  printf("\n\n");
  printf("     A    B    C       D    E    F\n\n");
  for (int i = 0; i < NumFilas; i++) {
    printf("%3d",i+1);
    for (int j = 0; j < AsientosFila; j++) {
      if ( j == Pasillo ) {
        printf("   ");
      }
      if (P[i].AsientosOcupa[j] == ocupado) {
        printf("  (*)");
      } else if (P[i].AsientosOcupa[j] == reservado) {
        printf("  (R)");
      } else if (P[i].AsientosOcupa[j] == vacio) {
        printf("  ( )");
      }
    }
    printf("\n");
  }
  printf("\n");
}
```

Fragmento de programa en C++

- Definición: Problema

  – Proposición encaminada a averiguar el modo de obtener un resultado, cuando se conocen ciertos datos de partida

- Tipos de Problemas
  - Sin Solución
  - Determinados: con una única solución
  - Indeterminados: con un número indefinido de soluciones

- Fases para resolver un problema con un programa informático:
  - Estudio del problema (ANÁLISIS)
  - Descripción de un método (algoritmo) que lo resuelva (DISEÑO)
  - Escritura del algoritmo en un lenguaje de programación (CODIFICACIÓN)
  - Comprobación del correcto funcionamiento (PRUEBA)

La definición del problema

- Análisis del problema
  - Consiste en establecer con precisión qué se necesita
- Especificación
  - Descripción precisa del problema:
  - - datos de partida
  - - resultado

lenguaje natural ⟹ puede resultar impreciso

lenguajes formales ⟹ lógica, matemáticas

Un ejemplo

Ejemplo de Especificación
Problema de división euclídea

– Datos
  - 2 enteros, dividendo y divisor (D,d)
  - d no nulo
– Resultado
  - 2 enteros, cociente y resto (C,R)
  - $0 \leq R < d$, tal que $D = d*C + R$

- **Algoritmo - Etimología**
  - Alhuarizmí: sobrenombre del árabe Muhamed ibn Musa (al-Jwarizmi), matemático persa. Escribió un tratado sobre la manipulación de números y ecuaciones: "Kitab al-jabr w'almugabala". ¿Os suena?

- **Definición de Algoritmo (1):**
  - Descripción precisa de los pasos que nos llevan a la solución de un problema

Problemas, Algoritmos y Programas

- # Algoritmo – Ejemplo

Cambiar una rueda de coche pinchada

```
Inicio
PASO 1.    Aflojar los tornillos de la rueda pinchada con
           la llave inglesa.
PASO 2.    Ubicar el gato mecánico en su sitio.
PASO 3.    Levantar el gato hasta que la rueda pinchada
           pueda girar libremente.
PASO 4.    Quitar los tornillos y la rueda pinchada.
PASO 5.    Poner rueda de repuesto y los tornillos.
PASO 6.    Bajar el gato hasta que se pueda liberar.
PASO 7.    Sacar el gato de su sitio.
PASO 8.    Apretar los tornillos con la llave inglesa.
Fin
```

Algoritmo

- Definición (2):

  – Método tal que partiendo de datos apropiados, conduce sistemáticamente a los resultados requeridos en la especificación del problema

Caracterización

- La descripción de un algoritmo afecta a:
  - Entrada (Datos)
  - Proceso (Instrucciones)
  - Salida (Resultado)

Es constructivo: hay que precisar también el proceso de cálculo

- Se puede decir:
  - Algoritmo $\cong$ función matemática
  - Algoritmo: Entrada $\rightarrow$ Salida (proceso)
- Ejemplo: Suma Lenta: $N \times N \rightarrow N$
  - $a + b \rightarrow c, \quad c = a + b$

Características de los algoritmos

- **Precisión (sin ambigüedad) en cuanto a:**
  - Orden: secuencia de pasos que han de llevarse a cabo
  - Contenido: qué se realiza en cada paso
- **Determinismo:**
  - Debe responder del mismo modo ante las mismas condiciones
- **Finitud:**
  - Debe tener fin

Aspectos de un algoritmo

- **Obligatorios**
  - Corrección: respecto a las especificaciones
  - Complejidad: recursos que un algoritmo necesita. En máquinas secuenciales (tiempo y memoria)
- **Deseables**
  - Generalidad: sirva para una clase de problemas lo más amplia posible
  - Eficiencia: será más eficiente en la medida que necesita de menos pasos

Lenguajes Algorítmicos

- Sirven para describir un algoritmo
- Son más precisos que el lenguaje natural, pero menos rígidos (o formales) que un lenguaje de programación
  - Se les considera un lenguaje intermedio
  - Tienen cierta independencia de los lenguajes de programación

Algoritmo – Ejemplo (pseudocódigo)

Un estudiante se encuentra en su casa (durmiendo placenteramente) y debe ir a la URJC (a clase de programación!!!), ¿qué debe hacer?

```
Inicio
Dormir
hacer
   Dormir
hasta que suene el despertador.
Mirar la hora.
¿Hay tiempo suficiente?
      Si hay, entonces
          Ducharse.
          Vestirse.
          Desayunar.
      Si no,
            Vestirse.
Cepillarse los dientes.
Despedirse de la familia.
```

```
¿Hay tiempo suficiente?
      Si hay, entonces
         Caminar a la estación de metro.
      Si no,
         Correr hacia la estación de metro.
Hacer
    Esperar el metro
    Ver a las demás personas que esperan el metro y ver
     continuamente cuánto falta para que llegue el metro.
Hasta que pase un metro hacia Manuel Becerra
Subirse al metro.
Mientras no llegue a Manuel Becerra
hacer
    Seguir en el metro.
    Hacer cualquier cosa con el móvil como todos los demás.
Bajarse.
Salir de la estación y entrar a la universidad.
Fin
```

Ejemplo: Algoritmo SumaLenta

Partimos de dos cantidades: a y b. El método de suma lenta consiste en ir pasando de a a b una unidad cada vez, de forma que cuando a=0, el resultado será el valor de b

Algoritmo Suma lenta (Pseudocódigo)

Sean a, b $\in$ N

Leer a y b

Mientras a $\neq$ 0, hacer $\left\{ \begin{array}{l} a \leftarrow a-1 \\ b \leftarrow b+1 \end{array} \right.$

Escribir b

- Ejemplo
  - Pseudocódigo, diagramas de flujo

Algoritmo: definición formal

- Es una cuádrupla que contiene los siguientes elementos:
  - Conjunto de los estados que pueden presentarse en todo momento
  - Identificación de estados iniciales
  - Identificación de estados finales
  - Función de transición entre estados

Algoritmo: definición formal

- Un estado se define por una tupla de cuatro elementos
  - Marca de la posición del algoritmo en la que se define el estado
  - Datos de entrada
  - Resultados emitidos
  - Valores de las variables que entran en juego

Ejemplo: Algoritmo SumaLenta

Ejemplo
Estados de cómputo (Suma lenta)

Sean a, b $\in$ N

Leer a y b

Mientras a $\neq$ 0, hacer $\left\{ \begin{array}{l} a \leftarrow a-1 \\ b \leftarrow b+1 \end{array} \right.$

Escribir b

Datos de entrada → E

Resultados emitidos → S

Posición	***E***	***S***	***a***	***b***
1	[2 3↵]	[]	¿	¿
2	[]	[]	2	3
3	[]	[]	1	3
4	[]	[]	1	4
5	[]	[]	0	4
6	[]	[]	0	5
7	[]	[5↵]	0	5

Valores de los datos [a,b]

Algoritmo

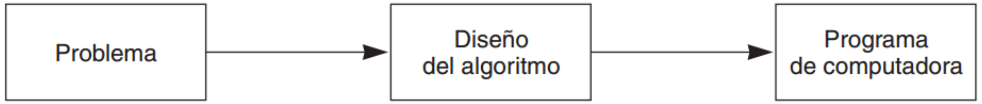
- Ejercicio
  - Escribir un algoritmo que realice la suma de todos los números pares entre 2 y 1000.

Problemas y Algoritmos

- **Algunos problemas tienen distintas soluciones algorítmicas**
  - Ejemplo
    - Máximo común divisor (MCD)
      - Por descomposición en factores primos
      - Usando el algoritmo de Euclides
      - Usando el mínimo común múltiplo
- **Algunos problemas NO tienen solución algorítmica**
  - Ejemplo
    - Problema de la parada (encontrar un algoritmo que determine si otro algoritmo finaliza o no con unos determinados datos de entrada)

- **Definición de Programa**
  - Conjunto de instrucciones precisas, en un lenguaje entendible por la computadora
- **Programación**
  - Proceso de construcción de programas
- **Fases:**
  - Análisis del problema
  - Solución conceptual del problema - Diseño
  - Escritura del algoritmo en un lenguaje de programación – Codificación
  - Comprobación de resultados - Prueba

- **Definición de Lenguaje de Programación:**
  - Un lenguaje artificial, diseñado para representar algoritmos de forma inteligible para las computadoras
- **LPs vs lenguaje natural**
  - LPs son más formales y rigurosos
  - LPs son más simples en su sintaxis y semántica

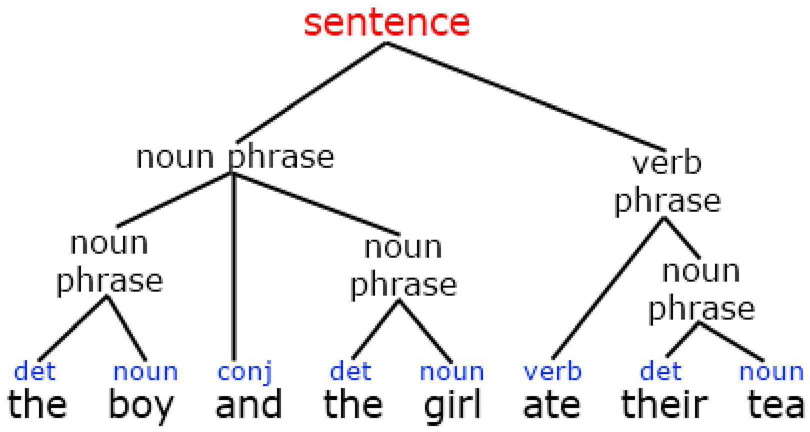


- Algunas características relevantes:
  - Sintaxis
  - Semántica
  - Traducción y Ejecución
  - Errores y cómo subsanarlos

Sintaxis

- Especifica inequívocamente cómo están construidos los programas de un LP
- Especificación de la sintaxis
  - Gramáticas (BNF)
  - Diagramas Sintácticos

```
<dirección postal> ::= <nombre> <dirección> <apartado postal>

<personal> ::= <primer nombre> | <inicial> "."

<nombre> ::= <personal> <apellido> [<trato>] <EOL>
           | <personal> <nombre>

<dirección> ::= [<dpto>] <número de la casa> <nombre de la calle> <EOL>

<apartado postal> ::= <ciudad> "," <código estado> <código postal> <EOL>
```



Semántica

- Asigna un significado a cada tipo de construcción de un LP
- Formas de especificación:
  - ejemplos (y contraejemplos) en los manuales
  - definición formal
- Ejemplo

```
write('hola');
write('hola');
```

holahola ✔

hola
hola ✘

Traducción y Ejecución

- El lenguaje de alto nivel ha de traducirse al lenguaje de la máquina
- Formas de traducción:
  - Compilación:
    - Todo el código fuente (en un archivo) se traduce a código ejecutable (en otro archivo)
    - Se ejecuta dicho código ejecutable

Código fuente

Código ejecutable

Traducción y Ejecución

– Interpretación:

1. Se traduce una instrucción del código fuente
2. Se ejecuta dicha instrucción
3. Se repiten 1 y 2 con todas las instrucciones del código

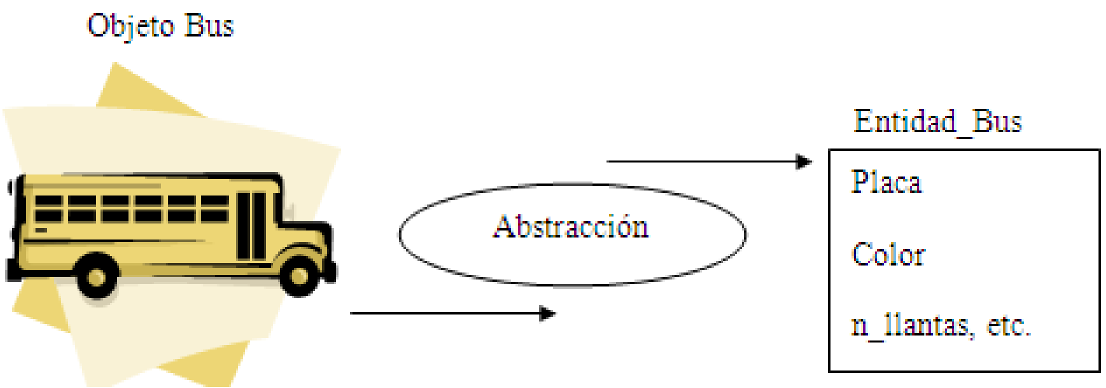
Errores

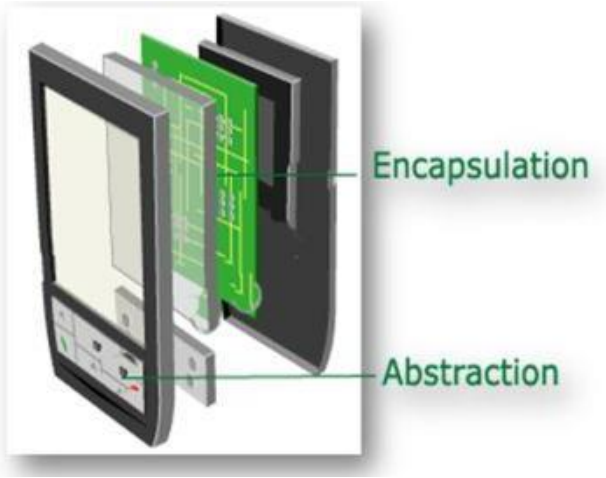
- Errores de compilación
  - Surgen a la hora de traducir (“compilar”) el código fuente
  - Errores sintácticos, de tipo, etc.
- Errores de ejecución
  - Surgen al ejecutar el código ejecutable
  - Operaciones ilegales (división por cero), errores lógicos etc.

Evolución de los LP

- Motores que impulsan la evolución de los lenguajes de programación:
  - Abstracción
  - Encapsulación
  - Modularidad
  - Jerarquía

- **Abstracción:**
  - Proceso mental por el que el ser humano extrae las características esenciales de algo, e ignora los detalles superfluos
- **Encapsulación:**
  - Proceso por el que se ocultan los detalles de las características de una abstracción

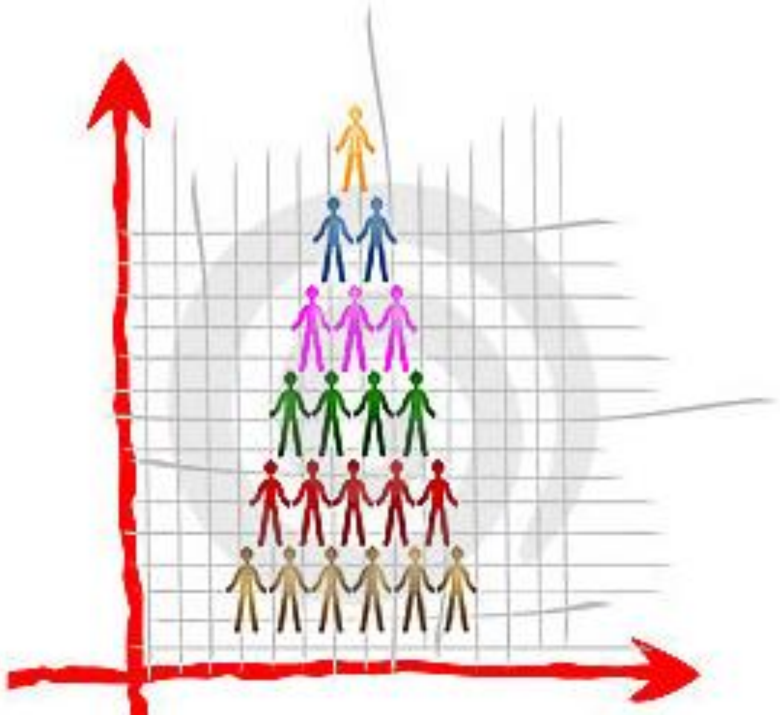




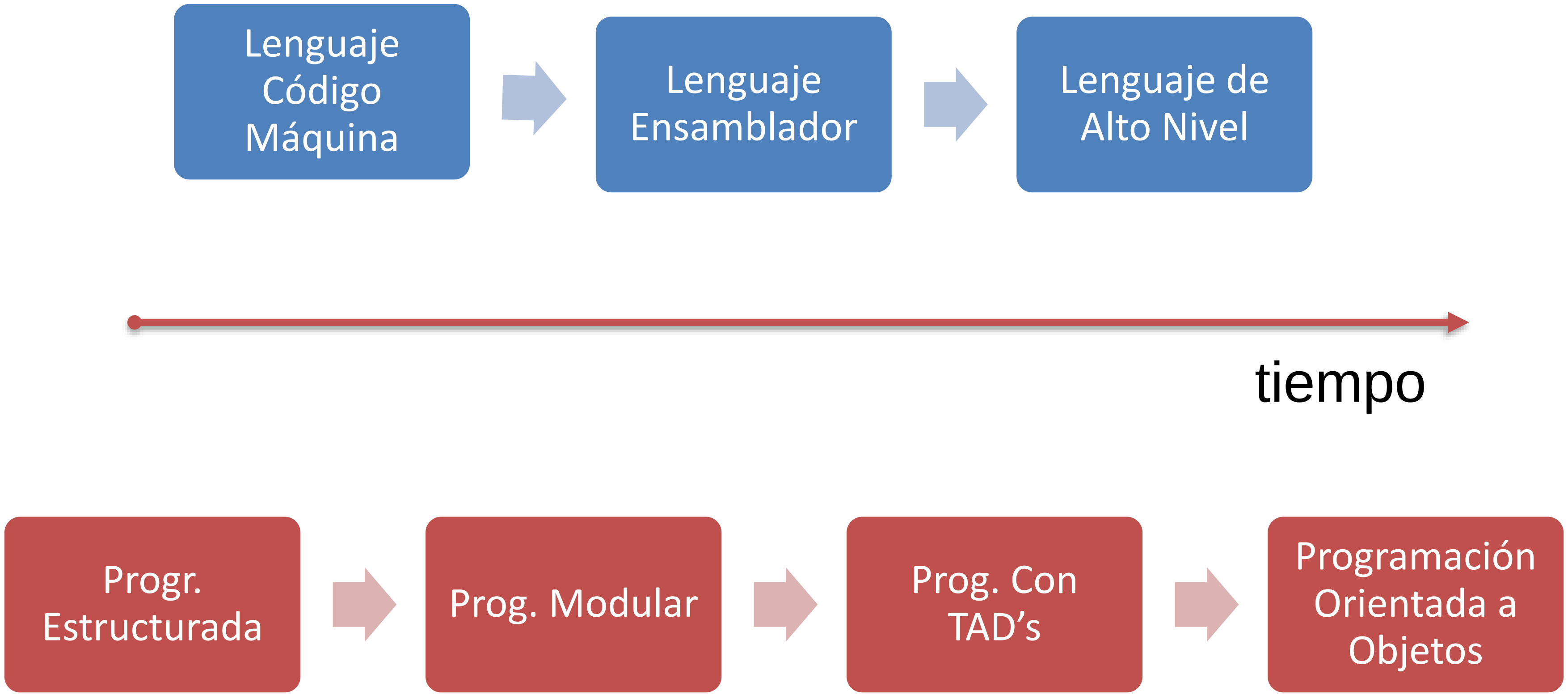
Evolución de los LP

- **Modularización:**
  - Proceso de descomposición de un sistema en un conjunto de elementos poco acoplados (independientes) y cohesivos (con significado propio)

- **Jerarquía:**
  – Proceso de estructuración por el que se organizan un conjunto de elementos en distintos niveles, atendiendo a determinados criterios (responsabilidad, composición, etc.)



Evolución de los LP



Paradigmas de programación

- **Definición:**
  - Una colección de patrones conceptuales que moldean la forma de razonar sobre problemas, de formular algoritmos y, a la larga, de estructurar programas
- **Paradigmas:**
  - Programación imperativa
  - Programación funcional
  - Programación lógica

Programación funcional

- **Basada en la noción de funcion matemática**
  - f: Dominio → Rango
- **Programar:**
  - Definir funciones básicas (con parámetros) (p.e. por enumeración)
  - Diseñar funciones complejas (p.e. por comprensión)
  - Evaluar las funciones sobre los datos de entrada

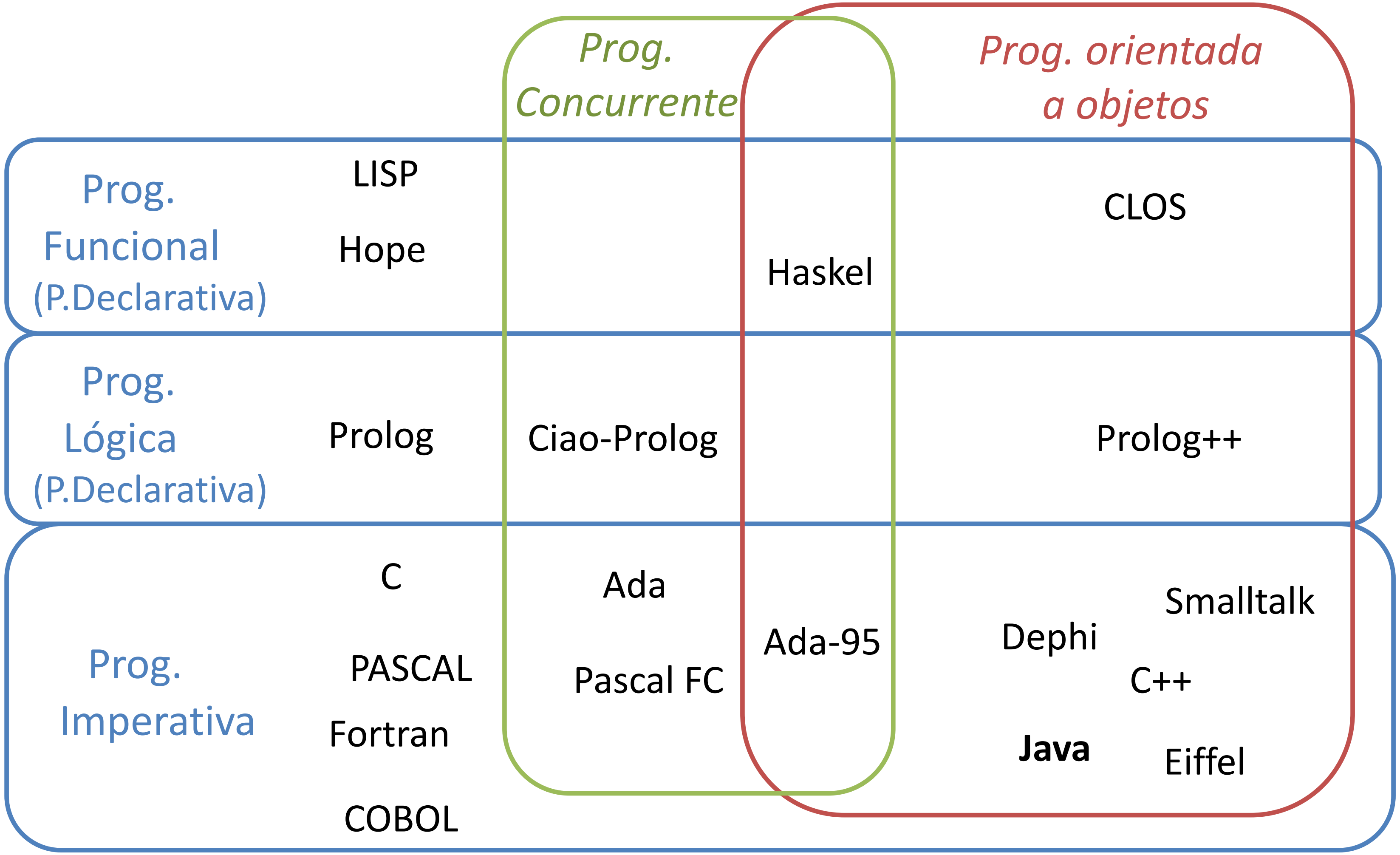
Programación lógica

- Basada en la inferencia automática en (un subconjunto de) lógica de 1er orden
- Programar:
  - Definir hechos (predicados básicos)
  - Diseñar implicaciones para definir predicados complejos
  - Determinar la verdad de los predicados para individuos concretos

Programación imperativa

- **Basada en el modelo von Neumann**
  - Un conjunto de operaciones primitivas
  - Ejecución secuencial
- **Abstracción**
  - Variables, expresiones, instrucciones
- **Programar:**
  - Declarar variables necesarias
  - Diseñar una secuencia adecuada de instrucciones (asignaciones)

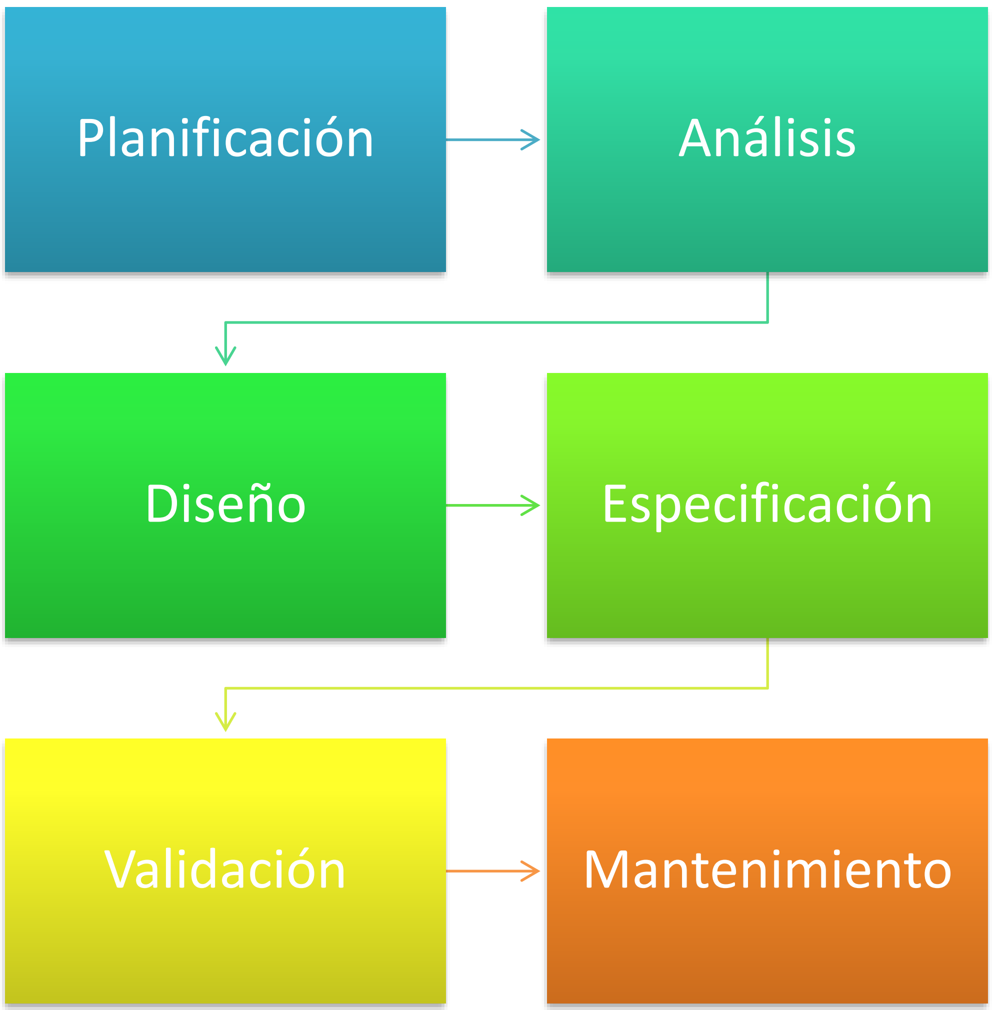
Paradigmas y lenguajes



- Definición (Bauer, 1969):
  - El establecimiento y uso de principios robustos de la ingeniería a fin de obtener económicamente software que sea fiable y que funcione eficientemente sobre máquinas reales

- Definición (IEEE, 1993):
  - La aplicación de un enfoque sistemático, disciplinado y cuantificable hacia desarrollo, operación y mantenimiento de software

Fases de un desarrollo sistemático



Planificación

- Determinar las necesidades de programación
- Estimación de recursos de desarrollo
- Predicción aproximada de coste y tiempo
- Determinar si el desarrollo del software es viable económicamente

Análisis de requerimientos

- Definir detalladamente las funciones de cada módulo, de acuerdo con los deseos del cliente
- Definir detalladamente el trabajo conjunto de los distintos módulos
- Definir criterios y sistema de validación
- Redactar especificaciones detalladas del funcionamiento general del software

Diseño

- Diseñar el conjunto de bloques o módulos
- Se dividen en partes o tareas
- Se asignan tareas a equipos de trabajo, que las desarrollan y prueban

Codificación

- Escribir los algoritmos en el lenguaje de programación elegido
- Integrar las partes para que formen un programa completo

Validación

- Aplicar el sistema de pruebas descrito en la fase de análisis de requerimientos
- Métodos de validación
  - Pruebas (tests), inspecciones …
  - Verificación formal
- Objetos de validación:
  - Los módulos de programa
  - Las conexiones entre ellos (integración)
  - La aplicación entera

Mantenimiento

- Redactar la documentación actualizada
- Iniciar la explotación
- Detectar y subsanar errores cometidos en etapas anteriores
- Adaptar la aplicación a requisitos cambiados

Wrap-up

- **Los ordenadores son capaces de desempeñar tareas porque alguien les ha dicho cómo hacerlas**
  - Alguien ha recogido las instrucciones en un **programa**

```
If the column number is greater than 60,
    then go to the next line.
Otherwise (if the column number isn't greater than 60),
    then stay on the same line.
```

```
if (columnNumber > 60) {
    wrapToNextLine();
}
else {
    continueSameLine();
}
```

  - Alguien que es capaz de:
    - Descomponer problemas grandes en problemas más pequeños que se resuelven con soluciones paso a paso
    - Expresar esos pasos en un lenguaje muy particular y preciso (un lenguaje de programación)

Wrap-up

- De nuestra mente al procesador
  - Compilador: código a código objeto (human-friendly a computer-friendly)
  - Máquina Virtual: recorre las instrucciones (computer-friendly)
  - API (Application Programming Interface): montones de código disponible para ser utilizado

Wrap Up

Compilador “normal”

Código Fuente

```java
outer:
for (int i = 2; i < 1000; i++) {
    for (int j = 2; j < i; j++) {
        if (i % j == 0)
            continue outer;
    }
    System.out.println (i);
}
```

MyFile.java

```
11001010 11111110 10111010 10111110 00000000 00000000
00000000 00101110 00000000 00010101 00001010 00000000
00000101 00000000 00010000 00001010 00000000 00000100
00000000 00010001 00001010 00000000 00000100 00000000
00010010 00000111 00000000 00010011 00000111 00000000
00010100 00000001 00000000 00000110 00111100 01101001
01101110 01101001 01110100 00111110 00000001 00000000
00000011 00101000 00101001 01010110 00000001 00000000
00000100 01000011 01101111 01100100 01100101 00000001
00000000 00001111 01001100 01101001 01101110 01100101
01001110 01110101 01101101 01100010 01100101 01110010
01010100 01100001 01100010 01101100 01100101 00000001
00000000 00001011 01100100 01101001 01110011 01110000
01101100 01100001 01111001 01010111 01101111 01110010
01100100 00000001 00000000 00000100 00101000 01001001
00101001 01010110 00000001 00000000 00001110 01110111
01110010 01100001 01110000 01010100 01101111 01001110
01100101 01111000 01110100 01001100 01101001 01101110
01100101 00000001 00000000 00010000 01100011 01101111
01101110 01110100 01101001 01101110 01110101 01100101
01010011 01100001 01101101 01100101 01001100 01101001
01101110 01100101 00000001 00000000 00001010 01010011
01101111 01110101 01110010 01100011 01100101 01000110
```

Código Objeto

Compilador Java

Bytecode

```
0:   iconst_2
1:   istore_1
2:   iload_1
3:   sipush  1000
6:   if_icmpge       44
9:   iconst_2
10:  istore_2
11:  iload_2
12:  iload_1
13:  if_icmpge       31
16:  iload_1
17:  iload_2
18:  irem
19:  ifne    25
22:  goto    38
25:  iinc    2, 1
28:  goto    11
31:  getstatic       #84; // Field java/lang/System.out:Ljava/io/PrintStream;
34:  iload_1
35:  invokevirtual   #85; // Method java/io/PrintStream.println:(I)V
38:  iinc    1, 1
41:  goto    2
44:  return
```

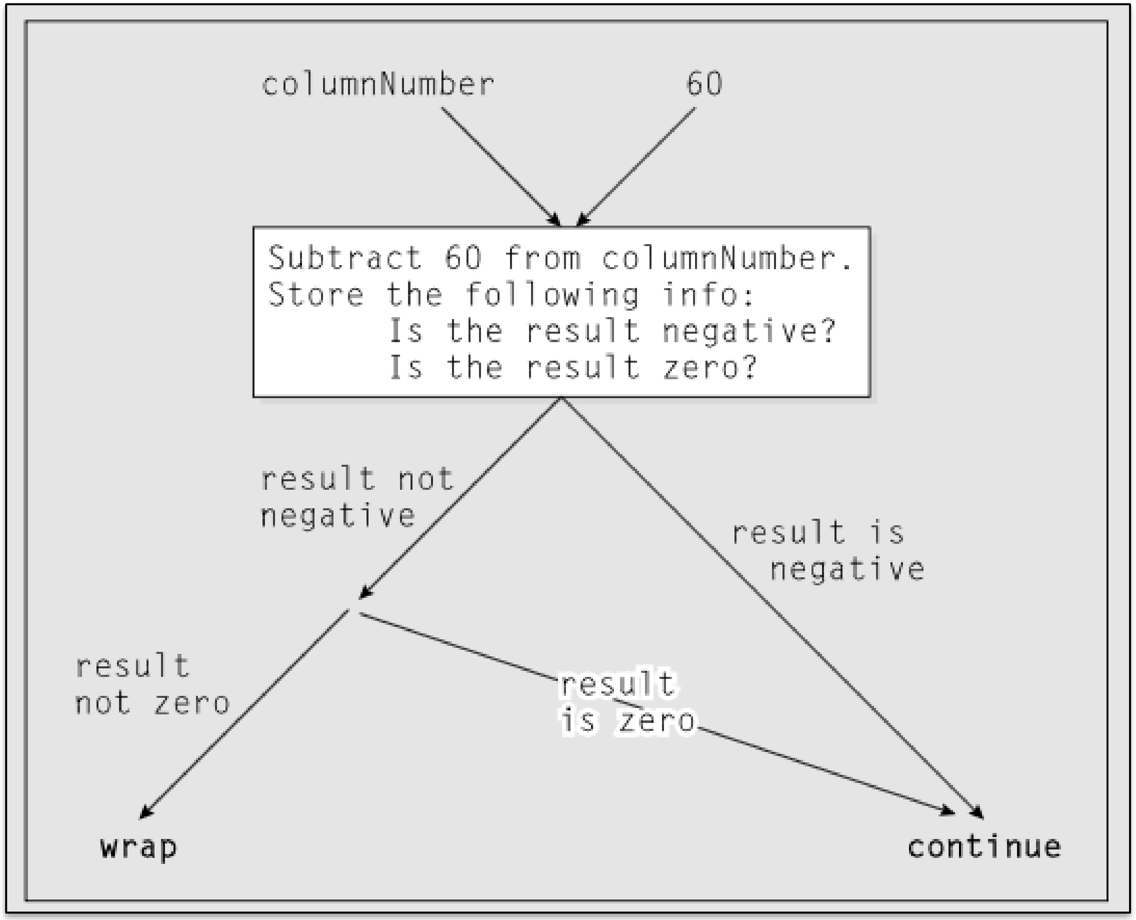
MyFile.class

Wrap Up

- **Bytecode**
  - El código fuente (.java) de nuestro programa describe las operaciones que debe hacer el ordenador
  - El compilador traduce el código fuente a Bytecode (.class) que *explota* cada instrucción en un conjunto de diminutos pasos que el procesador puede llevar a cabo
    - Trasiego de datos a memoria
    - Operaciones con esos datos (…)

Ejemplo Bytecode

```
if (columnNumber > 60) {
    wrapToNextLine();
}
else {
    continueSameLine();
}
```

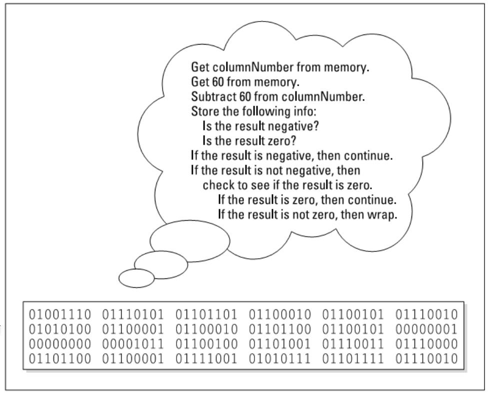


Wrap Up

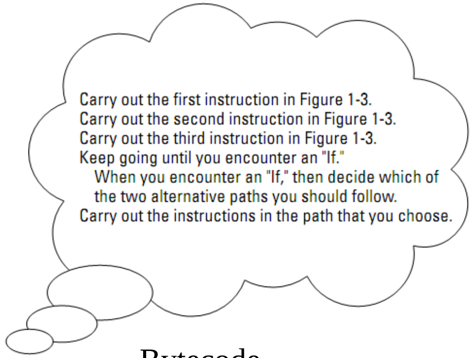
Ejecución de un programa con la mayoría de Lenguajes de Programación

```
if (columnNumber > 60) {
    wrapToNextLine();
}
else {
    continueSameLine();
}
```

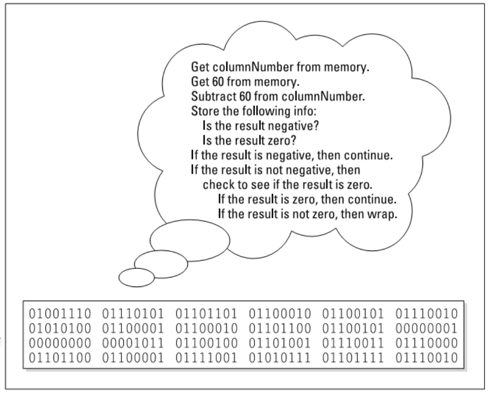
Código Fuente



Código Objeto



Bytecode

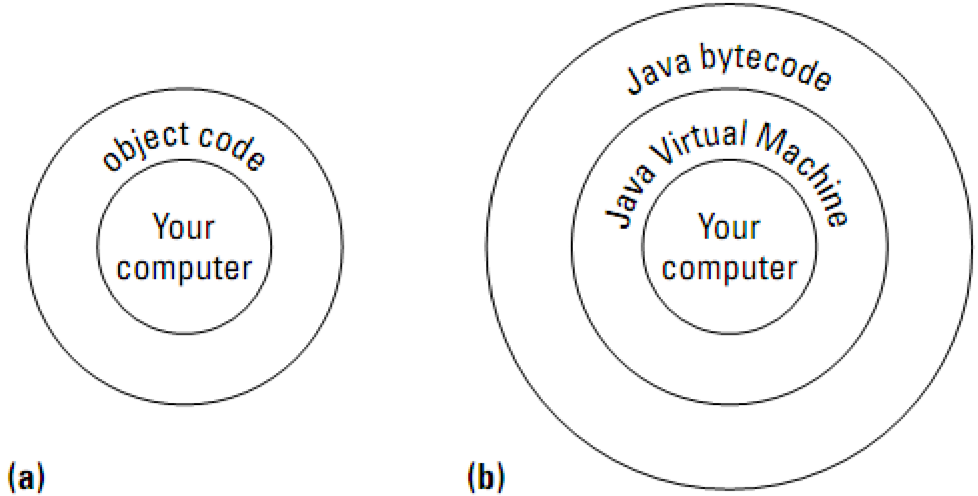


Código Objeto

Ejecución de
un programa Java

Wrap Up

- Ejecución de un programa en Java



Write Once, Run Anywhere

Introduccion a la programación

Fundamentos de la programación

Okybele Research Group

ISe Ingeniería de Servicios www.ssme.es

Universidad Rey Juan Carlos

1: Types, Variables, Operators

Goal

Learn enough Java to do something useful

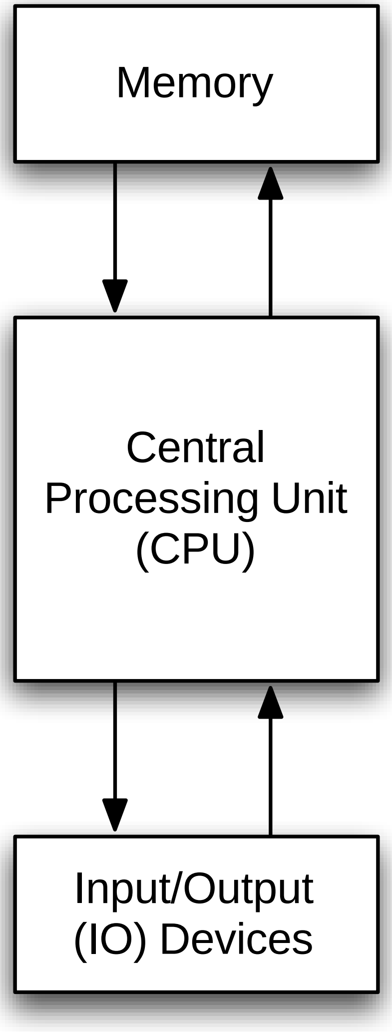
Examples

- Automate mathematical operations
- Process data
- Create and play around with objects
- Draw some graphics

Assignments

- View and submit via Codeboard.io
- Collaborate with others
- Write your **own** code
- Must submit every assignment

The Computer



CPU Instructions

$z = x + y$

Read location x

Read location y

Add

Write to location z

Programming Languages

- Easier to understand than CPU instructions
- Needs to be translated for the CPU to understand it

Java

- “Most popular” programming language
- Runs on a “virtual machine” (JVM)
- More complex than some others (eg. Python)
- Simpler than others (eg. C++)

What is Java

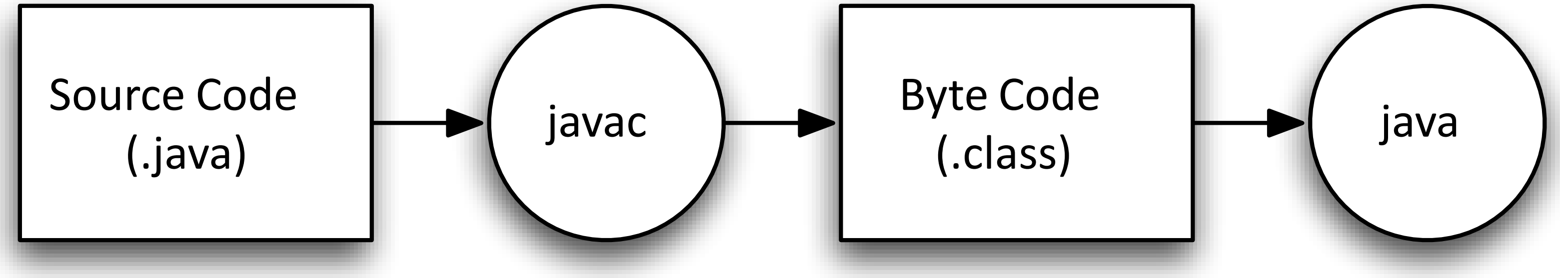
- Object Oriented programming language from the 90s
- A programming tool developd by 13 people manged by James Gosling for the *7 project
- Syntactically similar to C/C++ but much more simple
- Platform independent: ”write once, run anywhere”
- Oak → Green → Java
  - Just Another Vague Acronym
  - James Gosling, Arthur Van Hoff, Andy Bechtolsteim

Java

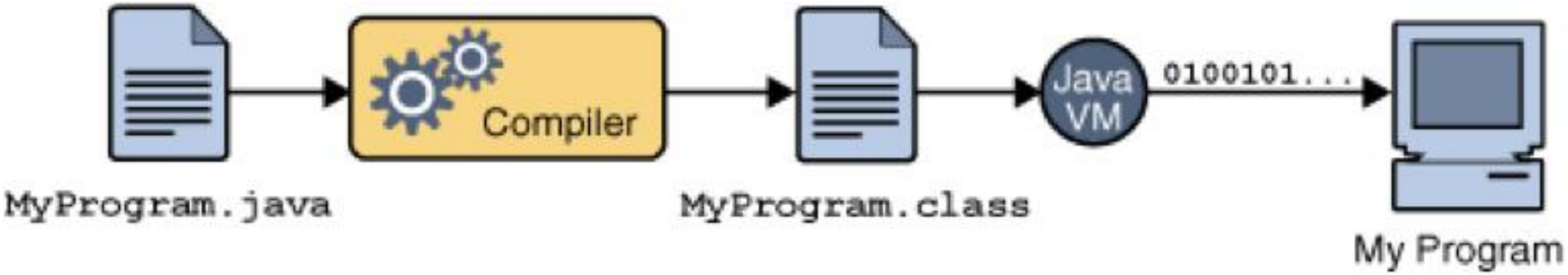
Java history

- Java 1.0 -1995
- Java 1.1 - 1997
- Java 1.2 - 1998 (Playground) → Java 2
- Java 1.3 - 2000 (Kestrel)
- Java 1.4 - 2002 (Merlín)
- Java 1.5 - 2004, (Tiger) → Java 5
- Java 1.6.0 - 2006, (Mustang) → Java 6
- Java SE7 - 2011, (Dolphin) → Java 7
- Java SE8 - 2014, (Spider) → Java 8

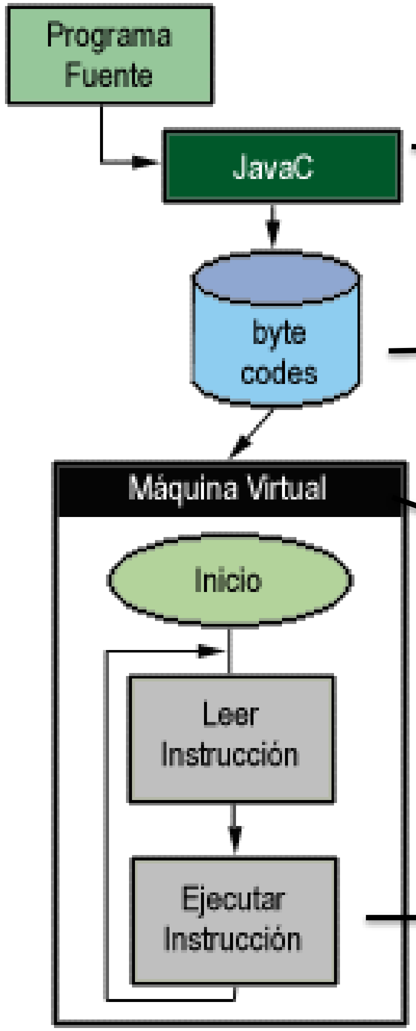
Java Virtual Machine



– A JVM is needed to run a Java program



Compiling Java



Compilador Java

Traduce código fuente a código intermedio.

Interprete de Java

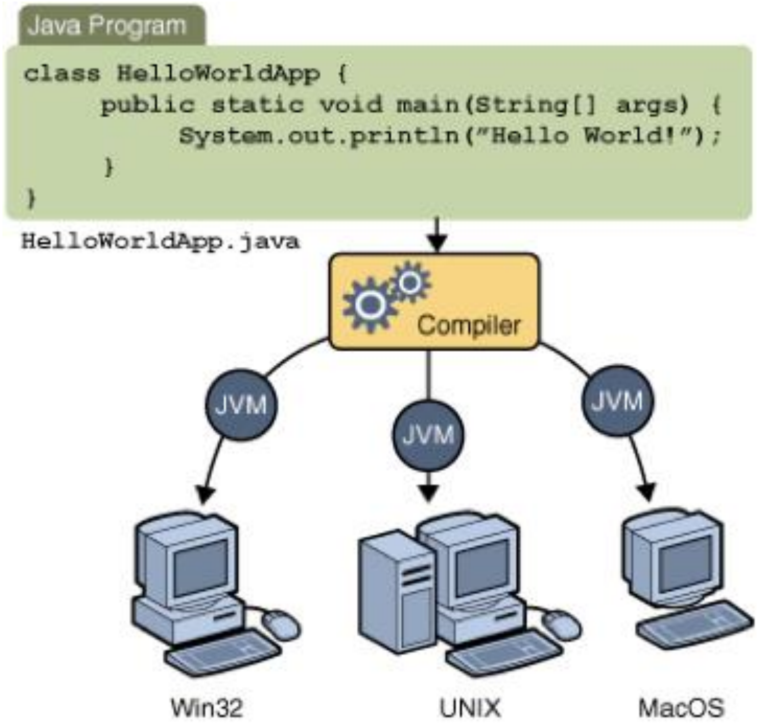
ByteCodes interpretados (ejecutados)

Compiling Java

Bytecode

- Native language for any JVM.
- A Java program runs in any platform

Write once,
Run anywhere



First Program

```
class Hello {
   public static void main(String[] arguments)
   {
      // Program execution begins here
      System.out.println("Hello
      world.");
   }
}
```

Program Structure

```
class CLASSNAME {
  public static void main(String[]
  arguments) {
    STATEMENTS
  }
}
```

Output

System.out.println(*some String*) outputs to the console

Example:

– System.out.println("output");

Second Program

```
class Hello2 {
   public static void main(String[] arguments) {
      System.out.println("Hello world."); // Print once
      System.out.println("Line number 2");   // Again!
   }
}
```

Types

Kinds of values that can be stored and manipulated.

- **boolean**: Truth value (true or false).
- **int**: Integer (0, 1, -47).
- **double**: Real number (3.14, 1.0, -2.1).
- **String**: Text (“hello”, “example”).

Variable declaration

Named location that stores a value of one particular type.

– Form:
  - *TYPE* NAME;
– Example

```
String foo;
```

Start with lower case and use an upper case for every new "word"

– Use intuitive and significative names for variables

Use = to give variables a value.

– Computes right part of the assignment and assigns the result to the variable on the left

Example:

```
String foo;
foo = "IAP 6.092";
```

foo

foo

“IAP 6.092”

Can be combined with a variable declaration.

Example

```
double badPi = 3.14;
boolean isJanuary = true;
```

```java
class Hello3 {
   public static void main(String[] arguments)
   {
       String foo = "IAP 6.092";
       System.out.println(foo);
       foo = "Something else";
       System.out.println(foo);
   }
}
```

Operators

Symbols that perform simple computations

– Simple arithmetic

Operador	Descripción	Ejemplo de expresión	Resultado del ejemplo
-	operador unario de cambio de signo	`-4`	`-4`
+	Suma	`2.5 + 7.1`	`9.6`
-	Resta	`235.6 - 103.5`	`132.1`
*	Producto	`1.2 * 1.1`	`1.32`
/	División (tanto entera como real)	`0.050 / 0.2` `7 / 2`	`0.25` `3`
%	Resto de la división entera	`20 % 7`	`6`

– Combined arithmetic

Operador	Descripción	Ejemplo de expresión	Resultado del ejemplo
+=	Suma combinada	`a+=b`	`a=a+b`
-=	Resta combinada	`a-=b`	`a=a-b`
*=	Producto combinado	`a*=b`	`a=a*b`
/=	División combinada	`a/=b`	`a=a/b`
%=	Resto combinado	`a%=b`	`a=a%b`

String Concatenation (+)

The + operator is overriden

- Conventional add operator when used with numbers
- String concatenation when used with Strings

```
int a = 3;
int b = 2;
int c = a + b;
// c = 5
String text = a + "000" + b;
// text = "30002"
System.out.println(c);
System.out.println(text);
```

Operators

Relational

Operador	Descripción	Ejemplo de expresión	Resultado del ejemplo
`==`	igual que	`7 == 38`	`false`
`!=`	distinto que	`'a' != 'k'`	`true`
`<`	menor que	`'G' < 'B'`	`false`
`>`	mayor que	`'b' > 'a'`	`true`
`<=`	menor o igual que	`7.5 <= 7.38`	`false`
`>=`	mayor o igual que	`38 >= 7`	`true`

Logical

Operador	Descripción	Ejemplo de expresión	Resultado del ejemplo
`!`	Negación - NOT (unario)	`!false` `!(5==5)`	`true` `false`
`|`	Suma lógica – OR (binario)	`true | false` `(5==5)|(5<4)`	`true` `true`
`^`	Suma lógica exclusiva – XOR (binario)	`true ^ false` `(5==5)|(5<4)`	`true` `true`
`&`	Producto lógico – AND (binario)	`true & false` `(5==5)&(5<4)`	`false` `false`
`||`	Suma lógica con cortocircuito: si el primer operando es `true` entonces el segundo se salta y el resultado es `true`	`true || false` `(5==5)||(5<4)`	`true` `true`
`&&`	Producto lógico con cortocircuito: si el primer operando es `false` entonces el segundo se salta y el resultado es `false`	`false && true` `(5==5)&&(5<4)`	`false` `false`

Truth tables

A	B	A OR B
F	F	F
F	V	V
V	F	V
V	V	V

A	B	A AND B
F	F	F
F	V	F
V	F	F
V	V	V

A	NOT A
F	V
V	F

Order of Operations

Follows standard (math) rules:

1. Parentheses
2. Arithmethic operators
    1. Multiplication and division
    2. Addition and subtraction
3. Relational operators
4. Logic operators

Operators

Which is the resut of the following expression?

```
3 + 5 < 5 * 2 || 3 > 8 && 7 > 6 - 2
```

Operators

Which is the resut of the following expression?

```
10 <= 2 * 5 && 3 < 4 || !(8 > 7) && 3 * 2 <= 4 * 2 - 1
```

Operators

Which is the resut of the following expressions?

Datos	Expresión	Resultado

`int x = 3;`
`int y = 6;`
	`!(x<5)&&!(y>=7)`	`!(3 < 5) && !(6 >= 7)`
		`!(TRUE) && !(FALSE)`
		`FALSE && TRUE`
		`FALSE`

`int i = 22;`
`int j = 3;`
	`!((22>4)||(3<=6))`	`!((22>4)||(3<=6))`
		`!( (TRUE) || (TRUE))`
		`! (TRUE)`
		`FALSE`

`int a = 34;`
`int b = 12;`
`int c = 8;`
	`!(a+b==c)||(c!=0)&&(b-c>=19)`	`!(a+b==c)||(c!=0)&&(b-c>=19)`
		`! (34 + 12 == 8) || (8 != 0) && (12 - 8 >= 19)`
		`! (FALSE) || (TRUE) && (FALSE)`
		`FALSE`

Operators

Which is the resut of the following expressions?

```
int i = 7;
float f = 5.5F;
char c = 'w';
```

Expresión	Resultado
`(i >= 6) && (c == 'w')`	
`(i >= 6) || (c == 119)`	
`(f < 11) && (i > 100)`	
`(c != 'p') || ((i + f) <= 10)`	
`i + f <= 10`	
`i >= 6 && c == 'w'`	
`c != 'p' || i + f <= 10`	

```java
class DoMath {
   public static void main(String[] arguments) {
      double score = 1.0 + 2.0 * 3.0;
      System.out.println(score);
      score = score / 2.0;
      System.out.println(score);
   }
}
```

```java
class DoMath2 {

   public static void main(String[] arguments) {
      double score = 1.0 + 2.0 * 3.0;
      System.out.println(score);
      double copy = score;
      copy = copy / 2.0;
      System.out.println(copy);
      System.out.println(score);
   }
}
```

Hands on

Every Java program must contains at least one Class

- Java methods container

A Java file could contain several classes, but just one of them being **Public**

- File name must be that of the Public class (ClassName.java)

```
public class PayrollApp {

    public static void main(String[] args) {
        int hours = 40;
        double payRate = 25.0, grossPay;

        grossPay = hours * payRate;
        System.out.print("Gross Pay: $");
        System.out.println(grossPay);
    }
}
```

Hands on

Defining classes

- Class definition starts with the **Class** keyword
- Every declaration and instruction is located between the start and end brackets of the class
- They are referred to as the **body** of the class

```
public class PayrollApp {

    public static void main(String[] args) {
        int hours = 40;
        double payRate = 25.0, grossPay;

        grossPay = hours * payRate;
        System.out.print("Gross Pay: $");
        System.out.println(grossPay);
    }
}
```

Body Class

Hands on

The main method

- Every program needs a **main** method
- It is the starting point of the program
- Always the same heading
- Always invoked when the program is run

```
public class PayrollApp {

    public static void main(String[] args) {
        int hours = 40;
        double payRate = 25.0, grossPay;

        grossPay = hours * payRate;
        System.out.print("Gross Pay: $");
        System.out.println(grossPay);
    }
}
```

Punctuation | Indentation

Each pair of brackets identifies a code block

```
class DoMath2 {

    public static void main(String[] arguments) {
        double score = 1.0 + 2.0 * 3.0;
        System.out.println(score);
        double copy = score;
        copy = copy / 2.0;
        System.out.println(copy);
        System.out.println(score);
    }
}
```

Code inside each block is indented to ease identificaction

Comments

Human readable information in the code

- Comments are intended to introduce information that will be ignored by the compiler inside the code
- End line comment (//)

```
int usu = 0; // el número de usuarios
```

- Block comment (/* … */)

```
/*
 * A program to list the good things in life
 * Author: Barry Burd, BeginProg2@BurdBrain.com
 * February 13, 2005
 */

class ThingsILike {

    public static void main(String args[]) {
        System.out.println("Chocolate, royalties, sleep");
    }
}
```

- Javadoc comment (/** …*/)

```
/**
 * Print a String and then terminate the line.
 */
```

Hands on

Punctuation symbols have well-defined purposes

- ; → end of instruction
- “” → Strings are located between
- {} → blocks delimitation
- () → methods arguments
- [ ] → arrays indexing

They also guide indentation and ease the identification of the different blocks

Kybele Research Group

ISe Ingeniería de Servicios www.ssme.es

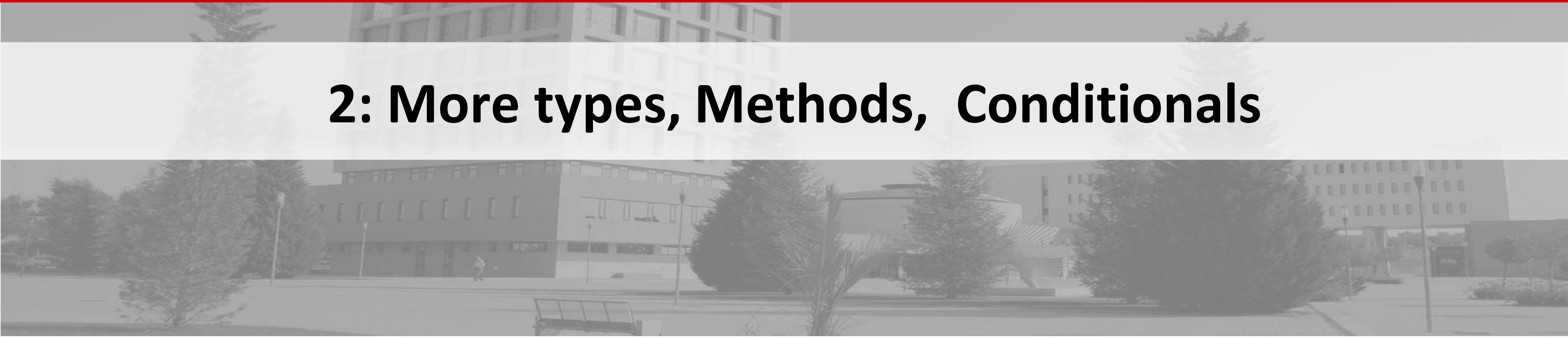
Universidad Rey Juan Carlos

1: Types, Variables, Operators

ISe
Ingeniería de Servicios
www.ssme.es

Universidad
Rey Juan Carlos

2: More types, Methods, Conditionals



Outline

- Lecture 1 Review
- More types
- Methods
- Conditionals

Types

Kinds of values that can be stored and manipulated.

boolean: Truth value (**true** or **false**).

int: Integer (0, 1, -47).

double: Real number (3.14, 1.0, -2.1).

String: Text (“hello”, “example”).

Variables

Named location that stores a value

Example:

```
String a = "a";
String b = "letter b";
a = "letter a";
String c = a + " and " + b;
```

Operators

Symbols that perform simple computations

- Assignment: =
- Addition: +
- Subtraction: -
- Multiplication: *
- Division: /

Exercise 1

```
class GravityCalculator {
 public static void main(String[] args) {
    double gravity = -9.81;
    double initialVelocity = 0.0;
    double fallingTime = 10.0;
    double initialPosition = 0.0;
    double finalPosition = .5 * gravity * fallingTime *
                                fallingTime;
    finalPosition = finalPosition +
                    initialVelocity * fallingTime;
    finalPosition = finalPosition + initialPosition;
    System.out.println("An object's position after " +
    fallingTime + " seconds is " +
     finalPosition + “ m.");
      }
}
```

Exercise 1

```
double finalPosition = .5 * gravity * fallingTime *
                                        fallingTime;
finalPosition = finalPosition + initialVelocity
                                  * fallingTime;
finalPosition = finalPosition + initialPosition;
```

OR

```
double finalPosition = .5 * gravity * fallingTime *
                                        fallingTime;
finalPosition = finalPosition + initialVelocity
                                  * fallingTime;
finalPosition += initialPosition;
```

Questions from last lecture?

Outline

- Lecture 1 Review
- More types
- Methods
- Conditionals

Division

Division ("/") operates differently on integers and on doubles!

Example:

```
double a = 5.0/2.0;  // a =2.5
int b = 4/2;  // b = 2
int c = 5/2;  // c = 2
double d = 5/2;   // d= 2.0
```

Order of Operations

Precedence like math, left to right

Right hand side of = evaluated first

Parenthesis increase precedence

```
double x = 3 / 2 + 1; // x = 2.0
double y = 3 / (2 + 1); // y = 1.0
```

Mismatched Types

Java verifies that types always match

```
String five = 5; // ERROR!
```

```
./Root/Main.java:8: error: incompatible types: int cannot be converted to String
        String five = 5;
                      ^
1 error
```

What is a casting?

– Taking an Object of one particular type and "turning it into" another Object type.

```
int a = 2;                          // a = 2
double a = 2;                       // a = 2.0 Implicit

int a = 18.7;                       // ERROR
int a = (int)18,7:                  // a = 18

double a = 2/3;                     // a = 0.0
double a = (double)2/3;             // a = 0.666 ...

double d = 5.25;
int i = (int) d;                    // d = 5 (Explicit) DOWNCAST

int d = 5;
double i = d;                       // i = 5.0 (Implicit) UPCAST
```

Outline

- Lecture 1 Review
- More types
- Methods
- Conditionals

Java Methods

– A collection of statements that are grouped together to perform an operation.
  - System.out.println()→
    - The system actually executes several statements in order to display a message on the console.

– The only required elements of a method declaration are the method's return type, name, a pair of parentheses, (), and a body between braces, {}.

Methods

Parts

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

Methods

Method declarations have six components:

- Modifiers.
- The return type (or void).
- The method name.
- The parameter list in parenthesis.
- An exception list.
- The method body, enclosed between braces.

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

Adding Methods

```
public static void NAME() {
    STATEMENTS
}
```

To call a method:

```
NAME();
```

Methods

```
class NewLine {
        public static void newLine() {
                System.out.println("");
        }

        public static void threeLines() {
                newLine();
                newLine();
                newLine();
        }

        public static void main(String[] arguments) {
                System.out.println("Line 1");
                threeLines();
                System.out.println("Line 2");
        }
}
```

Parameters

```
public static void NAME(TYPE NAME) {
   STATEMENTS
}

To call:

NAME(EXPRESSION);
```

Parameters

```
class Square {
      public static void printSquare(int x) {
            System.out.println(x*x);
      }

      public static void main(String[] arguments) {
            int value = 2;
            printSquare(value);
            printSquare(3);
            printSquare(value*2);
      }
}
```

Parameters

What’s wrong here?

```
class Square {
	public static void printSquare(int x) {
		System.out.println(x*x);
	}

	public static void main(String[] arguments) {
		printSquare(”hello”);
		printSquare(5.5);
	}
}
```

Parameters

What's wrong here?

```
class Square {
       public static void printSquare(double x) {
              System.out.println(x*x);
       }

       public static void main(String[] arguments) {
              printSquare(5);
       }
}
```

Multiple Parameters

```
[…] NAME(TYPE NAME, TYPE NAME) {
    STATEMENTS
}
```

To call:

```
NAME(arg1, arg2);
```

Multiple Parameters

```
class Multiply {
	public static void times (double a, double b) {
		System.out.println(a * b);
	}

	public static void main(String[] arguments) {
		times (2, 2);
		times (3, 4);
	}
}
```

Return Values

```
public static TYPE NAME() {
    STATEMENTS
    return EXPRESSION;
}
```

`void` means “no returned value”

Return Values

```
class Square3 {
	public static void printSquare(double x) {
		System.out.println(x*x);
	}
	public static void main(String[] arguments) {
		printSquare(5);
	}
}
```

```
class Square4 {
	public static double square(double x) {
		return x*x;
	}
	public static void main(String[] arguments){
		System.out.println(square(5));
		System.out.println(square(2));
	}
}
```

Variable Scope

Variables live in the block ({}) where they are defined (**scope**)

– Scope starts where the variable is declared
– … and ends whith the block where it was declared
– (the variable lives within the block)

Method parameters are like defining a new variable in the method

Variable Scope

```
class SquareChange {
  public static void printSquare(int x) {
      System.out.println("printSquare x = " + x);
      x = x * x;
      System.out.println("printSquare x = " + x);
  }

  public static void main(String[] arguments) {
      int x = 5;
      System.out.println("main x = " + x);
      printSquare(x);
      System.out.println("main x = " + x);
  }
}
```

Variables Scope

```
class Scope {
  public static void main(String[] arguments) {
      int x = 5;
      if (x == 5) {
            int x = 6;
            int y = 72;
            System.out.println("x = " + x + "
                  y = " + y);
      }
  }
  System.out.println("x = " + x + " y = " + y);
}
```

Methods: Building Blocks

Methods as the way of encapsulating functionality

- Big programs are built out of small methods
- Methods can be individually developed, tested and reused
- User of method does not need to know how it works
  - Black box operations
- In Computer Science, this is called "abstraction"

Mathematical Functions

Encapsulated functionality that we can use without having to master inner details

Overview Package Class Use Tree Deprecated Index Help — Java™ Platform Standard Ed. 7

Prev Class Next Class Frames No Frames All Classes

Summary: Nested | Field | Constr | Method Detail: Field | Constr | Method

java.lang

Class Math

java.lang.Object
    java.lang.Math

```
public final class Math
extends Object
```

The class `Math` contains methods for performing basic numeric operations such as the elementary exponential, logarithm, square root, and trigonometric functions.

Unlike some of the numeric methods of class `StrictMath`, all implementations of the equivalent functions of class `Math` are not defined to return the bit-for-bit same results. This relaxation permits better-performing implementations where strict reproducibility is not required.

By default many of the `Math` methods simply call the equivalent method in `StrictMath` for their implementation. Code generators are encouraged to use platform-specific native libraries or microprocessor instructions, where available, to provide higher-performance implementations of `Math` methods. Such higher-performance implementations still must conform to the specification for `Math`.

The quality of implementation specifications concern two properties, accuracy of the returned result and monotonicity of the method. Accuracy of the floating-point `Math` methods is measured in terms of *ulps*, units in the last place. For a given floating-point format, an ulp of a specific real number value is the distance between the two floating-point values bracketing that numerical value. When discussing the accuracy of a method as a whole rather than at a specific argument, the number of ulps cited is for the worst-case error at any argument. If a method always has an error less than 0.5 ulps, the method always returns the floating-point number nearest the exact result; such a method is *correctly rounded*. A correctly rounded method is generally the best a floating-point approximation can be; however, it is impractical for many floating-point methods to be correctly rounded. Instead, for the `Math` class, a larger error bound of 1 or 2 ulps is allowed for certain methods. Informally, with a 1 ulp error bound, when the exact result is a representable number, the exact result should be returned as the computed result; otherwise, either of the two floating-point values which bracket the exact result may be returned. For exact results large in magnitude, one of the endpoints of the bracket may be infinite. Besides accuracy at individual arguments, maintaining proper relations between the method at different arguments is also important. Therefore, most methods with more than 0.5 ulp errors are required to be *semi-monotonic*: whenever the mathematical function is non-decreasing, so is the floating-point approximation, likewise, whenever the mathematical function is non-increasing, so is the floating-point approximation. Not all approximations that have 1 ulp accuracy will automatically meet the monotonicity requirements.

```
public class Main {
  public static void main(String[] arguments) {
      int x = 90;
      Math.sin(x);
      Math.cos(Math.PI / 2);
      Math.pow(2, 3);
      System.out.println(...);}
  }
}
```

Outline

- Lecture 1 Review
- More types
- Methods
- Conditionals

```
if (CONDITION) {
   STATEMENTS
   /* statements performed
   when the boolean expression
   results true */
}
```

if statement

```java
public static void test(int x) {
    if (x > 5) {
        System.out.println(x + " is > 5");
    }
}

public static void main(String[] arguments) {
    test(6);
    test(5);
    test(4);
}
```

Comparison operators

```
x > y: x is greater than y
x < y: x is less than y
x >= y: x is greater than or equal to x
x <= y: x is less than or equal to y

x == y: x equals y
  ( equality: ==, assignment: =)
```

Boolean operators

```
&&: logical AND
||: logical OR
```

```
if (x > 6) {
   if (x < 9) {
    …
  }
}
```

```
if ( x > 6 && x < 9) {
…
}
```

else

```
if (CONDITION) {
   STATEMENTS
} else {
   STATEMENTS
   /* performed when CONDITION is
   not true */
}
```

else

```java
public static void test(int x) {
   if (x > 5) {
      System.out.println(x + " is > 5");
   } else {
      System.out.println(x + " is not > 5");
   }
}

public static void main(String[] arguments) {
   test(6);
   test(5);
   test(4);
}
```

else if

```
if (CONDITION) {
   STATEMENTS
} else if (CONDITION) {
   STATEMENTS
} else if (CONDITION) {
   STATEMENTS
} else {
   STATEMENTS
}
```

else

```
public static void test(int x) {
   if (x > 5) {
      System.out.println(x + " is > 5");
   } else if (x == 5) {
      System.out.println(x + " equals 5");
   } else {
      System.out.println(x + " is < 5");
   }
}

public static void main(String[] arguments) {
   test(6);
   test(5);
   test(4);
}
```

Questions?

Kybele Research Group

ISe Ingeniería de Servicios www.ssme.es

Universidad Rey Juan Carlos

3: Loops, Arrays

Assignment 2

Foo Corporationneeds a program to calculate how much to pay theiremployees.

1. Pay = hours worked x base pay
2. Hours over 40 get paid 1.5 the base pay
3. The base pay must be no less than $8.00
4. The number of hours must be no more than 60

Frequent Issues (I)

The signature of the *main* method *cannot* be modified.

```
public static void main(String[] arguments) {
                    ...
}
```

Frequent Issues (II)

Return values: if you declare that the method is not *void*, then it has to return something!

```
public static int pay(double basePay, int hours){
      if (basePay < 8.0)
            return -1;
      else if (hours > 60)
            return -1;
      else {
            int salary = 0;
            …
            return salary;
      }
```

Don't create duplicate variables with the same name

```
public static int pay(double basePay, int hours){

      int salary = 0; // OK

      int salary = 0; // salary already defined!!

      int salary = 0; // salary already defined!!


      }
```

Frequent Issues (III)

```
class WeeklyPay {
  public static void pay(double basePay, int hours) {
        if (basePay < 8.0) {
                System.out.println("You must be paid at least $8.00/hour");
                } else if (hours > 60) {
                        System.out.println("You can't work
                                        more than 60 hours a week");
        } else {
                int overtimeHours = 0;
                if (hours > 40) {
                overtimeHours = hours - 40;
                hours = 40;
                }
                double pay = basePay * hours;
                pay += overtimeHours * basePay * 1.5;

                System.out.println("Pay this employee $" + pay);
        }
  }
  public static void main(String[] arguments) {
        pay(7.5, 35);
  pay(8.2, 47);
  pay(10.0, 73);
  }
}
```

Whatwe have learned so far

- Variables & types
- Operators
- Type conversions & casting
- Methods & parameters
- *If* statement

Today's Topics

- Good programming style
- Loops
- Arrays

Good Programming Style

Good programming style

The goal of good style is to make your code more readable.

By you and by others.

Rule #1: use good (meaningful) names

```
String a1;
int a2;
double b;                    // BAD!!

String firstName;            // GOOD
String lastName;             // GOOD
Int temperature;             // GOOD
```

Rule #2: Use indentation

```
public class test {

public static void main (String[] arguments) { int x = 5;
x = x * x;
if (x > 20) {
System.out.println(x + " is greater than 20.");
}
double y = 3.4; }
}
```

```
public class test {

    public static void main(String[] arguments) {
        int x = 5;
        x = x * x;
        if (x > 20) {
            System.out.println(x + " is greater than 20.");
        }
        double y = 3.4;
    }
}
```

Rule #3: Use whitespaces

Put whitespaces in complex expressions

```
// BAD!!
double cel=fahr*42.0/(13.0-7.0);

// GOOD
double cel = fahr * 42.0 / (13.0 - 7.0);
```

Rule #4: Use whitespaces

Put blank lines to improve readability:

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

      int x = 5; x = x * x;
      if (x > 20) {
            System.out.println(x + " is > 20.");
      }

      double y = 3.4;
}
```

Rule #5: Do not duplicate tests

```
if (basePay < 8.0) {
        ...
}
else
        if (hours > 60) {
                ...
        }
else
        if (basePay >= 8.0 && hours <= 60) {
                ...
        }
```

Rule #5: Do not duplicate tests

```
if (basePay < 8.0) {
        ...
}
else
        if (hours > 60) {
                ...
        }
else
        ~~if (basePay >= 8.0 && hours <= 60)~~ {
                ...
        }
```

Good programming style (summary)

Use good names for variables and methods

Use indentation

Add whitespaces

Don't duplicate tests

Loops

Loops

What if you want to do it for200 Rules?

```
static void main (String[] arguments) {
    System.out.println("Rule #1");
    System.out.println("Rule #2");
    System.out.println("Rule #3");
}
```

Loops

Loop operators allow to loop through a block of code.

There are several loop operators in Java.

The *while* operator

Allows a programmer to state that an action (or a block of them) will be executed as long as certain condition is met

```
while (condition){
      STATEMENTS;
}
```

Must be a boolean expression

The *while* operator

```
int i = 0;
while (i < 3) {
      System.out.println("Rule #" + i);
      i = i+1;
}
```

Count carefully

Make sure that your loop has a chance to finish

– Meeting the condition has to be closer as the number of iterations grows

The *for* operator

Execute an statement (or block of them) a given number of times

```
for (initialization; condition; update){
        STATEMENTS;
}
```

The *for* operator

```
for (int i = 0; i < 3; i = i + 1){
        System.out.println("Rule #" + i);
}
```

i = i+1 may be replaced by i++

Condition is a boolean expression, which is computed at the end of each iteration.

If it yields true, another iteration comes

The initialization expression marks the start of the loop.

In general, it consists of declaring and initializing a variable so-called *control variable*

The update expression is executed at the end of each iteration.

In general, it consists of increasing the control variable

For

– Print all the integers between 1 and 20

```
for(int i = 1; i <= 10; i++)
{
  System.out.println(i);
}
```

– Print all the even numbers between 20 and 2)

```
for(int i = 20; i >=0; i -= 2)
{
  System.out.println(i);
}
```

Branching Statements

One might want to leave the loop, even though the condition has not been met yet

- break terminates a *for* or while *loop*

```
for (int i=0; i<100; i++) {
  if(i == 50)
     break;
  System.out.println("Rule #" + i);
}
```

Branching Statements

One might want to leave the current statement and go directly to the next one

- continue skips the current iteration of a loop and proceeds directly to the next iteration

```
for (int i=0; i<100; i++) {
   if(i == 50)
        continue;
   System.out.println("Rule #" + i);
}
```

Embedded loops

Scope of the variable defined in the initialization (control variable): respective *for* block

```
for (int i = 0; i < 3; i++) {
  for (int j = 2; j < 4; j++) {
    System.out.println (i + " " + j);
  }
}
```

do - while

Regarding while loops, the statements are **always** executed at least once

- Since condition is not computed until the end of the first iteration

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

do - while

Example

– Write down numbers between 1 and 10;

```
int i= 1;
do {
        System.out.println(i);
        i++;
} while (i <= 10);
```

Tips

Just while loop ends with ‘;’

```
for (int i=0; i<10; i++);
{
  System.out.println("i is " + i);
}

int i=0;
while (i < 10);
{
  System.out.println("i is " + i);
  i++;
}

int i=0;
do {
  System.out.println("i is " + i);
  i++;
} while (i<10);
```

BAD

GOOD

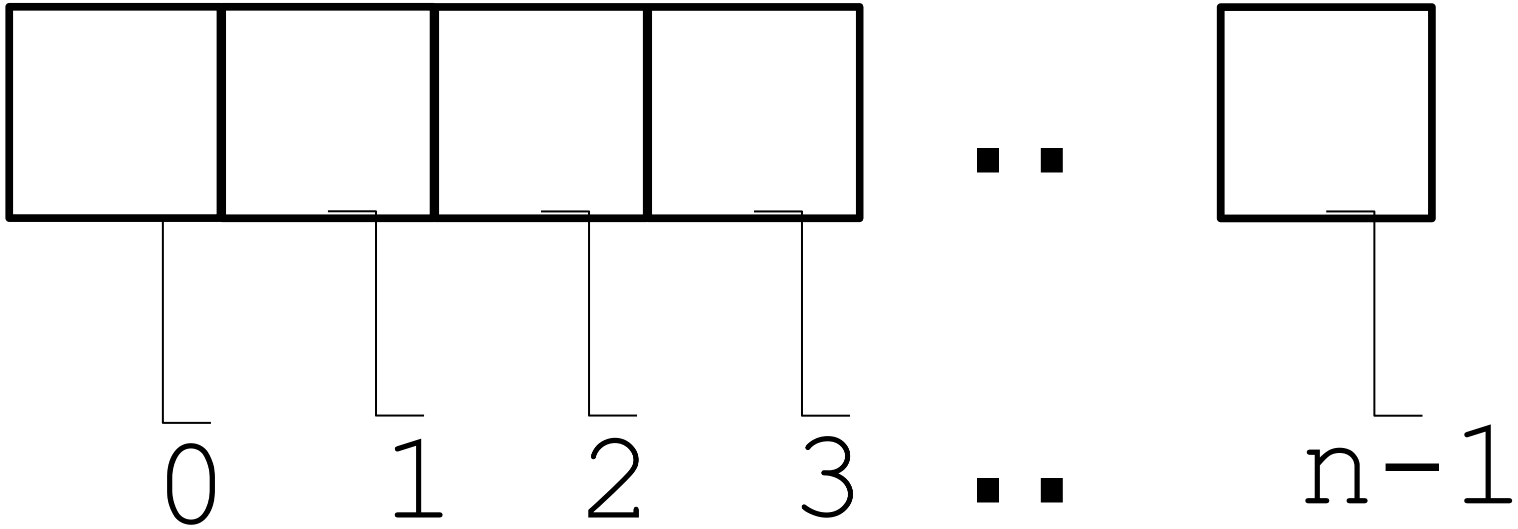
Arrays

Arrays

An array is an indexed list of values

- You can make an array of **any type** (int, double, String, etc.)
- All elements of an array must have the **same type**
- We can refer to the whole list of values (the array variable) …
- ... or to one specific value
- We can, as well, modify the list of values by adding, deleting of modifying each specific value

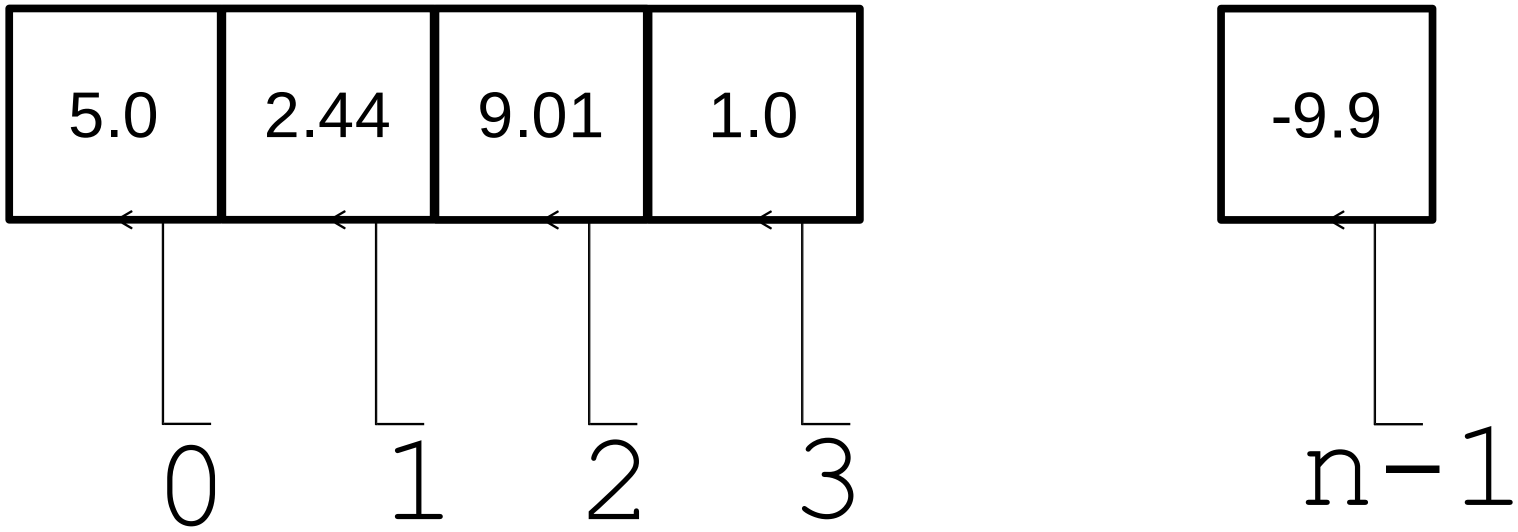
The first element of the array is located at **index 0**, while the last one is located at **index n - 1**



Arrays

The first element of the array is located at **index 0**, while the last one is located at **index n – 1**

– Example: `double []`;

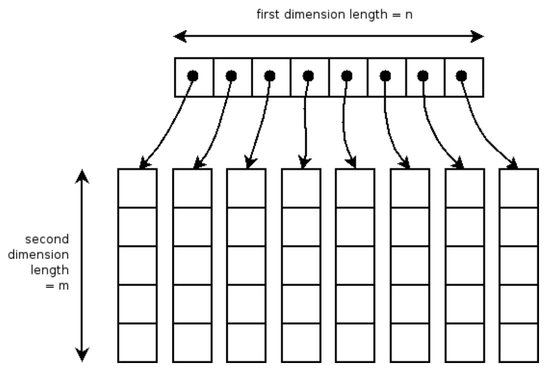


Array definition

– TYPE []

Arrays are just another type

– Arrays of array type can be defined




```
int [] values; // array of int values
int [][] values;
// array of array of int values
```

`int []` **is a data type**

Arrays

To create an array of a given size, use the new operator

- Or you may use a variable to specify the size:

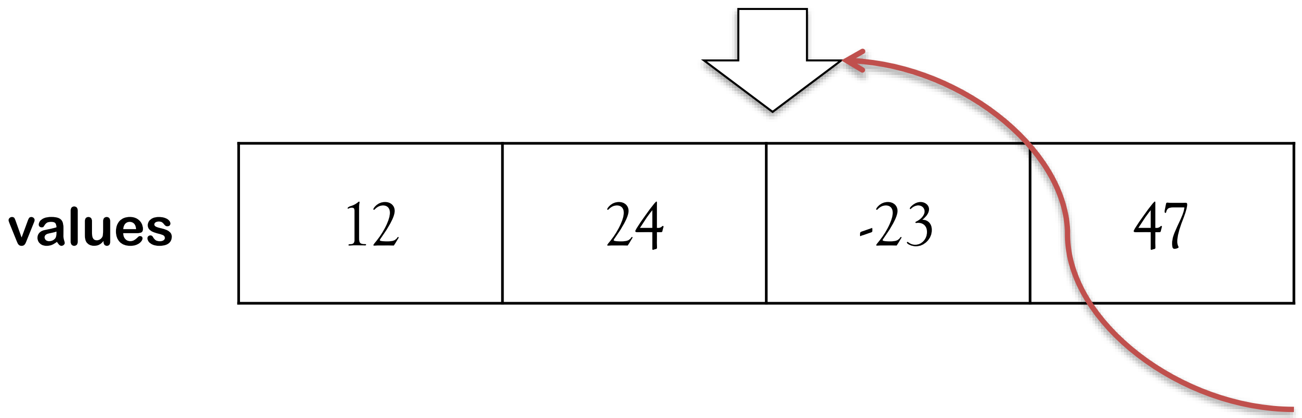
```
int[] values = new int[5];

// using a variable
int size = 12;
int[] values = new int[size];
```

Array Initialization

Curly braces can be used to initialize an array.

– It can ONLY be used when you declare the variable.

```
int[] values = {12, 24, -23, 47};
```

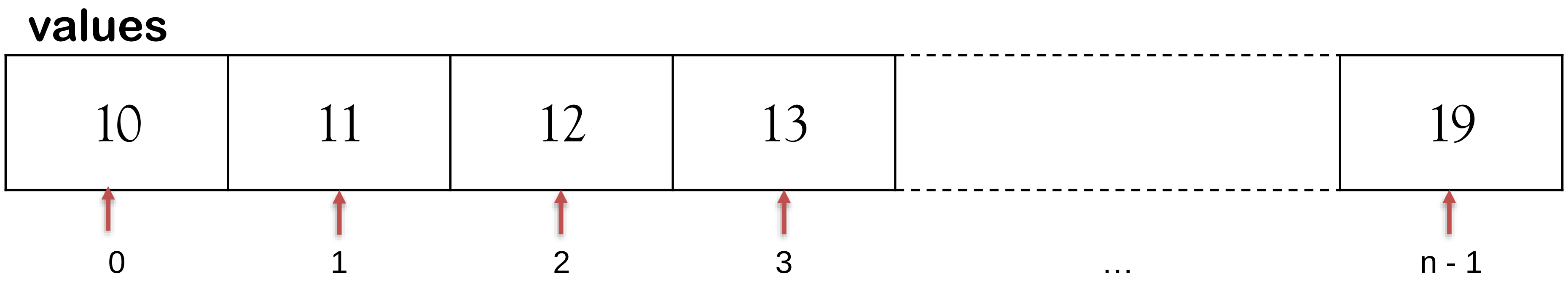


The size of the array is implicitly set to 4

Arrays

The first element of the array is located at **index 0**, while the last one is located at **index n - 1**

```
int[] values = new int[10];
values[0] = 10; // CORRECT
values[1] = 11; // CORRECT
values[2] = 12; // CORRECT
values[3] = 13; // CORRECT
values[9] = 19; // CORRECT
values[10] = 20; // WRONG!!
// compiles but throws an Exception
// at run-time (demo)
```



Quiz time!

Is there an error in this code?

```
int[] values = {1, 2.5, 3, 3.5, 4}
```

Accesing Arrays

To access the elements of an array:

– Use the [ ] operator and state the position needed

```
int[] values = {12, 24, -23, 47};
values[3] = 18; // {12, 24, -23, 18}
int x = values[1] + 3; // {12, 24, -23, 18}
```

Array starts at position 0 and ends at position length - 1

The *length* variable

Each array has a length variable built-in that contains the length of the array.

```
int[] values = new int[12];
int size = values.length; // size = 12

int[] values2 = {1,2,3,4,5}
int size2 = values2.length; // size = 5
```

String arrays

A side note

```
public static void main (String[] arguments){
    System.out.println(arguments.length);
    System.out.println(arguments[0]);
    System.out.println(arguments[1]);
}
```

A side note

```
public class Greet{
  public static void main(String[] args) {
            System.out.println("Good morning" + args[0]);
   }
}
```

>java Greet *aName*

For example:

> java Greet John

Good morning John

Using Arrays

Arrays as arguments

```
// method to print an Array
public static void printArray(int[] array) {
  for (int i = 0; i < array.length; i++) {
    System.out.print(array[i] + " ");
  }
}
(…)

// method call
int[] list = {3, 1, 2, 6, 4, 2};
printArray(list);

// method call (another shape of)
printArray(new int[]{3, 1, 2, 6, 4, 2});
```

Arrays as method output

```
// Array inversion method
public static int[] reverse(int[] list) {
  int[] result = new int[list.length];
  // declaring the array to be returned

  for (int i = 0, j = result.length - 1; i < list.length; i++, j--) {
    result[j] = list[i];
  }

  return result; // devolvemos el array
}
(…)
// method call
int[] list1 = new int[]{1, 2, 3, 4, 5, 6}; // array to invert
int[] list2 = reverse(list1); // inverted array
```

Array Utils

– Arrays copy → `System.arrayCopy(…)`

```
public static void arraycopy(Object src, int srcPos,
                  Object dest, int destPos, int length)
```

```
class ArrayCopyDemo {
        public static void main(String[] args) {
                char[] copyFrom = { 'd', 'e', 'c', 'a', 'f', 'f', 'e',
                         'i', 'n', 'a', 't', 'e', 'd' };
                char[] copyTo = new char[7];

                System.arraycopy(copyFrom, 2, copyTo, 0, 7);

                System.out.println(Arrays.toString(copyTo));
                // output: [c, a, f, f, e, i, n]
                System.out.println(new String(copyTo));
                // output: caffein
        }
}
```

Array Utils (II)

- java.util.Arrays

A method for each primitive type

```
public static char[] copyOfRange(char[] original,
                                       int from, int to)
```

```
class ArrayCopyDemo {
      public static void main(String[] args) {
            char[] copyFrom = { 'd', 'e', 'c', 'a', 'f', 'f', 'e',
                  'i', 'n', 'a', 't', 'e', 'd' };
            char[] copyTo = java.util.Arrays.copyOfRange(
                                    copyFrom, 2, 9);
            // no necesitamos crear el Array

            System.out.println(Arrays.toString(copyTo));
            // salida: [c, a, f, f, e, i, n]
      }
}
```

Array Utils (III)

- [java.util.Arrays]

```
int binarySearch(tipo[] a, tipo key)
// returns the position of 'key' in 'a' array

boolean equals(tipo[] a, tipo[] a2)
// yields true if 'a' and 'a2' contain the same values

void fill(tipo[] a, tipo val)
// set every position of array 'a' to 'val'

void sort(tipo[] a)
// orders 'a' array (ASC)
```

Combining Loops and Arrays

Example 1: iterating over an array (for)

```
int[] valores = new int[5];

for (int i=0; i < valores.length; i++) {
      valores[i] = i;
      int y = valores[i] * valores[i];
      System.out.println(y);
}
```

Looping through an array

Example 2: iterating over an array

```
int[] valores = new int[5];
int i = 0;
while (i < valores.length) {
     valores[i] = i;
     int y = valores[i] * valores[i];
     System.out.println(y);
     i++;
}
```

Provided we are going to iterate over an array, a for loop seems more appropiate

Iterating an array (improved for)

```
for(tipo variable_iteración: array)
   instrucciones;
```

```
int[] valores = {1,2,3,4,5};
int suma = 0;

for (int x: valores) {
      suma += x; // suma = suma + valores[i]
      System.out.println(suma);
}
```

Avoiding control variables, and array limits issues

Looping through an array

Iterating an array (improved `for`)

- Can leave the loop using the break statement

```
int[] valores = {1,2,3,4,5};
int suma = 0;

for (int x: valores) {
	suma += x; // suma = suma + valores[i]
	System.out.println(suma);
	if (suma > 5)
		break;
}
```

- But cannot modify the array
  - ~~x = …~~

Summary for today

1. Programming Style

2. Loops

3. Arrays

Assignment

A group of friends participate in the Boston Marathon.

Find the best performer.

Find the second-best performer.

Research Group

ISe
Ingeniería de Servicios
www.ssme.es

Universidad Rey Juan Carlos

3: Loops, Arrays

IV: Strings

Allows the creation of objects which chains of chars

```
String x, y, z;
String myName = "Paul";
```

Basic Operators

```
x = "Móstoles";
x = "Manuel Becerra"; // Móstoles desaparece
y = x; // y contendrá Manuel Becerra
x = ""; // x ahora es una cadena vacía
```

```
String x, y;
x = "Hola ";
y = "Mundo";
System.out.println(x+y);
// x.concat(y)
```

```
System.out.println("La suma total es: " + 25 + 30);
System.out.println("La suma total es: " + (25 + 30));
```

¿?

What if we want to use quotation marks in the text chain?

```
System.out.print(“El Atlético es un equipo “intenso” donde los hayan”);
// obtendríamos un ERROR de compilación
```

Scape character \ marks the beginning of a sequence that needs a special interpretation from the compiler

\t	Inserts a tabulator	\f	Inserts a new page
\b	Inserts a Backspace	\'	Inserts an angle bracket
\n	Inserts a new line	\"	Inserts inverted commas
\r	Inserts a carriage return	\\	Inserts the scape character

Clase String: Métodos

int length()

- Length of the string

int indexOf(String cad)

- This method returns the index within this string of the first occurrence of the specified character or -1, if the character does not occur.

char charAt(int ind)

- This method returns the character located at the String's specified index. The string indexes start from zero.

Clase String: Métodos

Boolean equals(String cad)

- This method compares this string to the specified object. The result is true if and only if the argument is not null and is a String object that represents the same sequence of characters as this object

int compareTo(String cad)

- This method compares two strings lexicographically
  - 0 if the argument is a string lexicographically equal to this string
  - < 0 if the argument is a string lexicographically greater than this string
  - > 0 if the argument is a string lexicographically less than this string.

String replaceAll(String oldCad, String newCad)

- This method replaces each substring of this string that matches the given regular expression with the given replacement

String toLowerCase()

String toUpperCase():

- Converts all of the characters in this String to upper case using (respectively to lower case).

String substring(int indexInicial, int indexFinal)

– This method has two variants and returns a new string that is a substring of this string.

– The substring begins with the character at the specified index and extends to

- the end of this string or
- up to endIndex – 1, if the second argument is given.

IV: Strings

V: Classes and Objects

E4 - FooCorporation

```
public static int getMinIndex(int[]
values) {
  int minValue = Integer.MAX_VALUE;
  int minIndex = -1;
  for(int i=0; i<values.length; i++)
    if (values[i] < minValue) {
       minValue = values[i];
       minIndex = i;
     }
  return minIndex;
}
```

E4 - FooCorporation

```
public static int getSecondMinIndex(int[] values) {
        int secondIdx = -1;
        int minIdx= getMinIndex(values);

        for(int i=0; i<values.length; i++) {
                if (i == minIdx)
                        continue;
                if (secondIdx == -1 ||
                    values[i] < values[secondIdx])
                        secondIdx = i;
        }
        return secondIdx;
}
```

What happens if

values = {0}?

values = {0, 0}?

values = {0,1}?

- Array Index vs Array Value

```
int[] valores = {99, 100, 101};
System.out.println(valores[0] ); // 99
```

Values	99	100	101
Indexes	0	1	2

Popular Issues 2

- Curly braces { ... } after if/else, for/while

```
for (int i = 0; i < 5; i++)
    System.out.println("Hi");
    System.out.println("Bye");
```

- ; after for/while

```
public static void main(String[] args) {
    for (int i = 0; i < 5; i++);
        System.out.println("Index: " + i);
}
```

Popular Issues 3

- Variable Initialization

```java
public static int getMinValue(int[] vals) {
    int min = 0;
    for (int i = 0; i < vals.length; i++) {
        if (vals[i] < min) {
            min = vals[i];
        }
    }
    return min;
}

// What if vals = {1,2,3}? ← Problem
// Set min = Integer.MAX_VALUE or vals[0]
```

Popular Issues 4

- Declare aux variable inside the loop

```
public static int getMinValue(int[] vals) {
        for (int i = 0; i < vals.length; i++) {
                int min = 0; // PROBLEM
                if (vals[i] < min) {
                        min = vals[i];
                }
        }
        return min;
}
```

Debugging Notes 1

- **Use** System.out.println() **throughout your code to see what it's doing**

```
public static int getMinValueDebugging(int[] vals) {
    int min = vals[0];
    for (int i = 0; i < vals.length; i++) {
        if (vals[i] < min) {
            System.out.println("Current minimun value: " + min);
            System.out.println("New minimum value: " + vals[i]);
            min = vals[i];
        }
    }
    return min;
}
```

Debugging Notes 2

- **Format / Indent code appropiately**

```
for (int i = 0; i < vals.length; i++)
      { if (vals[i] < vals[minIdx]) {
minIdx=i;}
return minIdx;}
```

  – Is there any error? Hard to find this way

Index

- **So far …**
  - Variables and data types
  - Operators
  - Type conversion and castings
  - Methods and parameters
  - *If* sentence
  - Loops and Arrays
- **En este tema veremos**
  - Object oriented programming
  - Defining Classes
  - Using Classes
  - References vs Values
  - Static types and methods

Object Oriented Programming

- **Emulate real world**
  - The idea is to define moulds representing the different types of entities found in real world (Class)
  - These are later used to create entities and put them to work by asking them to perform actions

Baby

Name

Sex

Weight

Decibels

#poopsSoFar

```java
public class Baby {

        // Properties
        String name;
        boolean male;
        int weight;
        int decibels;
        int numPoops;

}
```

- Why creating moulds if the only need is to handle different values?

```
// little baby alex
String nameAlex;
double weightAlex;
```

- Why creating moulds if the only need is to handle different values?

```
// little baby alex
String nameAlex;
double weightAlex;
// little baby david
String nameDavid;
double weightDavid;
```

- Why creating moulds if the only need is to handle different values?

```
// little baby alex
String nameAlex;
double weightAlex;
// little baby david
String nameDavid;
double weightDavid;
// little baby david
String nameDavid2;
double weightDavid2;
```

David2? ☹

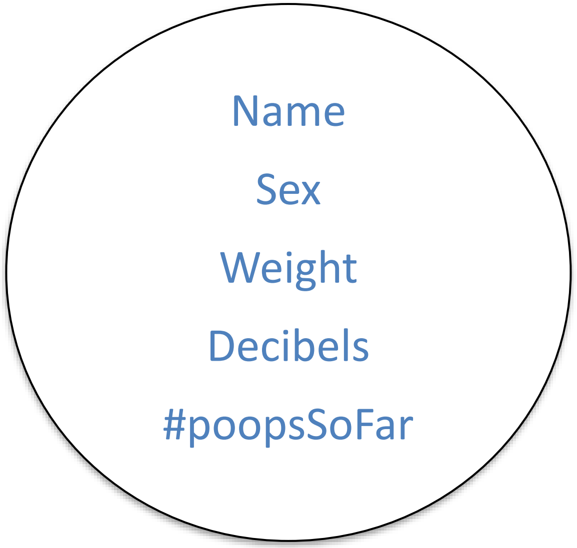
- Why creating moulds if the only need is to handle different values?

```
// little baby alex
String nameAlex;
double weightAlex;
// little baby david
String nameDavid;
double weightDavid;
// little baby david
String nameDavid2;
double weightDavid2;
```

David2? ☹

What if 500 babies are to be created?!!!

- Why bothering with Class definition?



Baby1

- Why bothering with Class definition?

Name
Sex
Weight
Decibels
#poopsSoFar

Baby1

Name
Sex
Weight
Decibels
#poopsSoFar

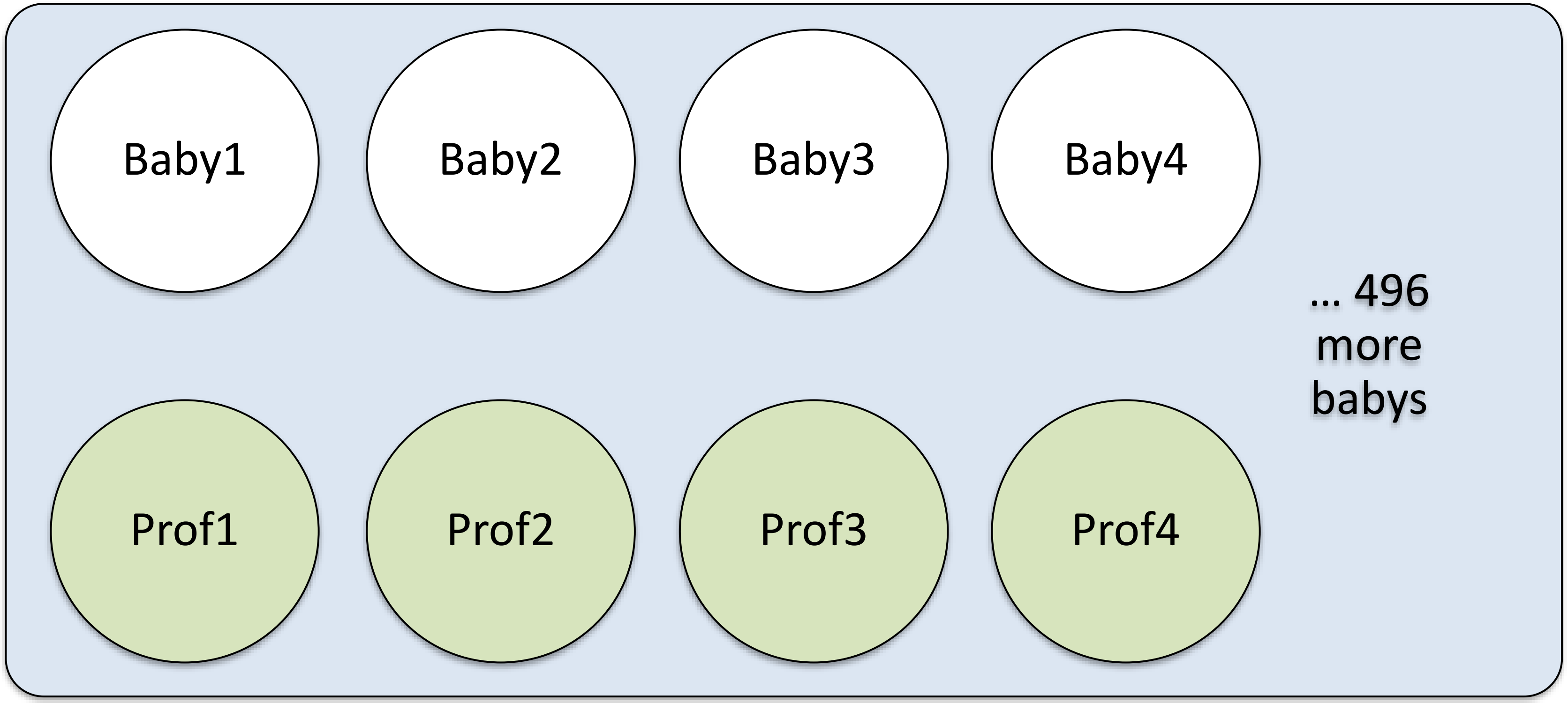
Baby2

Name
Sex
Weight
Decibels
#poopsSoFar

Baby3

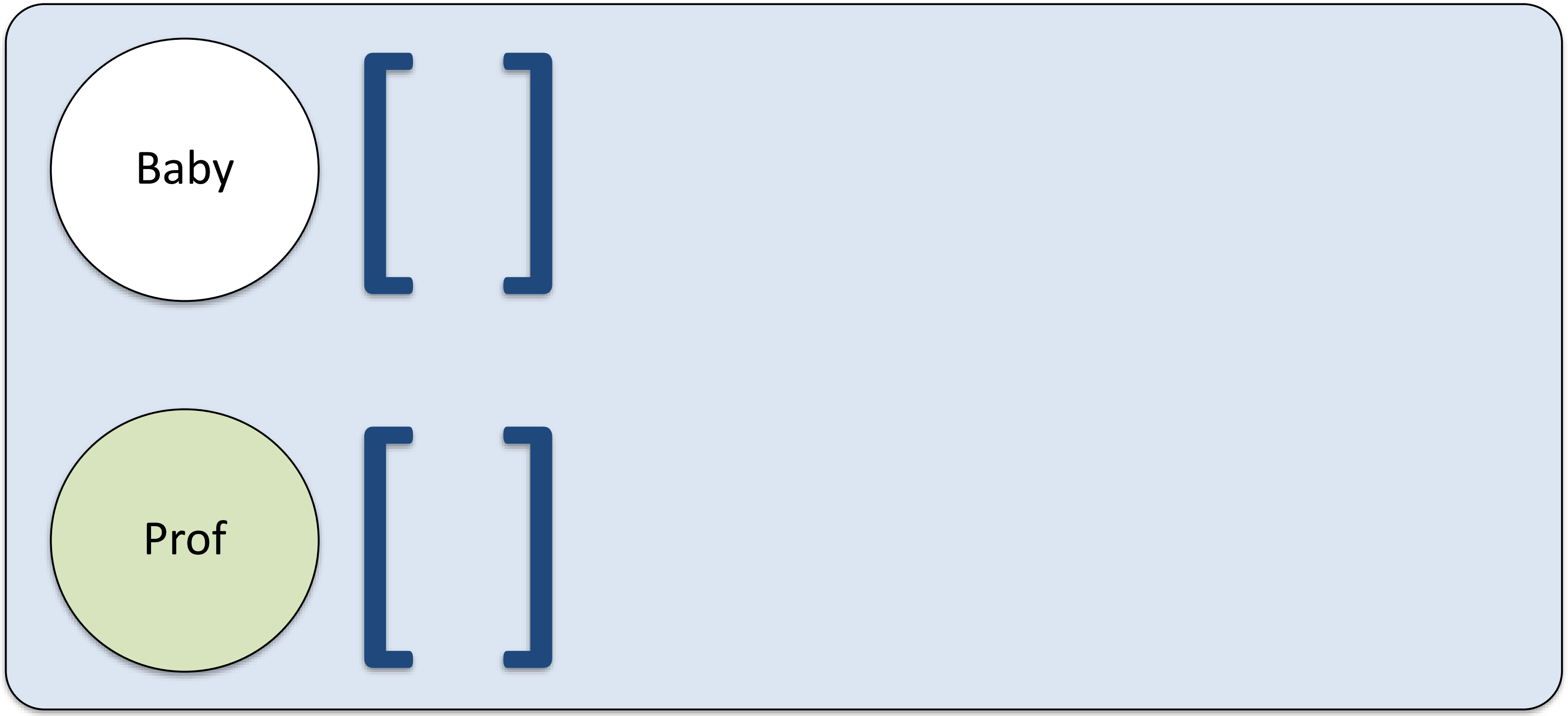
… 496 more babys

- Why bothering with Class definition?



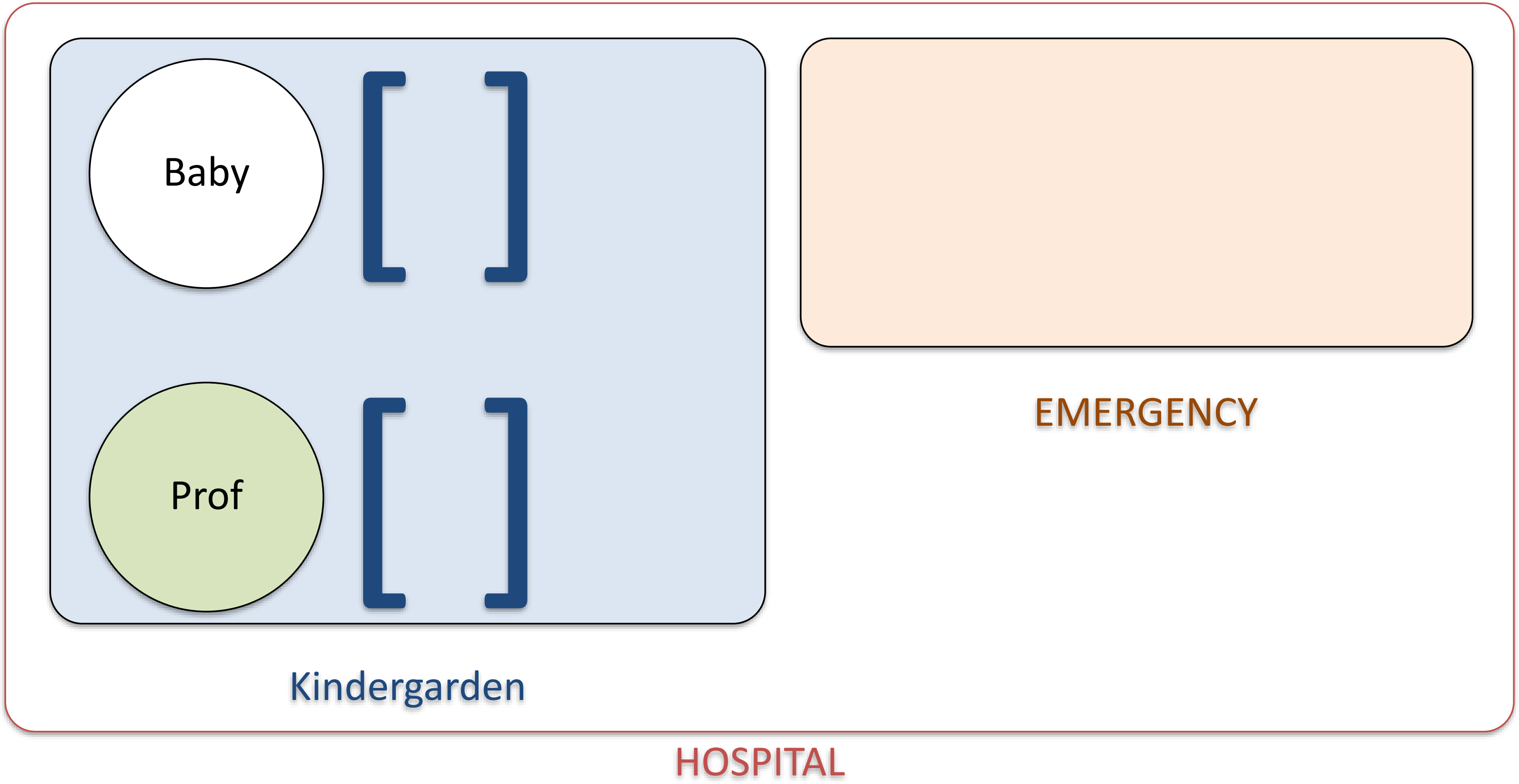
Kindergarden

- Why bothering with Class definition?



Kindergarden

- Why bothering with Class definition?



CLASS DEFINITION

- Class Definition

```
public class Baby {

    PROPERTIES (FIELDS)

    ACTIONS (METHODS)

}
```

- Class Definition

```java
public class Baby {

        // Properties
        String name;
        boolean isMale;
        double weight;
        double decibelis;
        int numPoops;
                                                          FIELDS

        void poop() {
                numPoops = numPoops + 1;
                System.out.println(”Dear mother, "
                        + ”I have pooped. Ready the diaper.");

        }
                                                          METHODS
}
```

Class Definition - Notes

- Class names are Capitalized
- 1 Class = 1 file
- Having a main method means the class can be run

- Class Definition

```
public class Baby {

    TYPE field_name
    TYPE field_name = init_value;

    METHODS

}
```

- Class Definition

```
public class Baby {

    // Properties
    String name;
    boolean isMale;
    double weight;
    double decibels;
    int numPoops = 0;

    METHODS

}
```

- Class Definition

```
public class Baby {

    // Properties
    String name;
    boolean male;
    double weight;
    double decibelis;
    int numPoops = 0;
    XXXX YYYY;

    METHODS

}
```

What if the baby has some siblings?

- Class Definition

```
public class Baby {

    // Propiedades
    String name;
    boolean male;
    double weight;
    double decibelis;
    int numPoops = 0;
    Baby[] siblings;

    METHODS

}
```

- Class Definition

```java
public class Baby {

        // Propierties
        String name;
        boolean male;
        double weight;
        double decibels;
        int numPoops = 0;
        Baby[] siblings;

        void poop() {
                numPoops = numPoops + 1;
                System.out.println(”Dear mother I have pooped!!!");
        }

}
```

Using Classes

- Let's make a baby

```
public class PlayingWithBabies {

    public static void main(String[] args) {
        // Let's make a baby!!
        Baby miBaby = new Baby();
        miBaby.poop();
        miBaby.poop();
    }
}
```

Our baby has no name, no sex … ☹

Constructors

- **(Class) Methods with particular features**
  - Constructor name == the class name
  - No return type – never returns anything
  - All classes need at least one constructor
  - If you don't write one, defaults to …

```
public class CLASSNAME{
        CLASSNAME ( ) {
        }

        CLASSNAME ([ARGUMENTS]) {
        }
}


CLASSNAME obj1 = new CLASSNAME();
CLASSNAME obj2 = new CLASSNAME([ARGUMENTS])
```

Default constructor if no other is invoked / provided

Constructores

```
public class Baby {

        // Fields
        String name;
        boolean male;
        double weight;
        double decibels;
        int numPoops;

        Baby (){
        }

        Baby (String baby_name) {
                name = baby_name;
        }

        Baby (String baby_name, Boolean baby_male) {
                name = baby_name;
                isMale = baby_male;
        }
}
```

METHODS

```java
public class Baby {

        // Fields
        String name;
        boolean isMale;
        double weight;
        double decibels;
        int numPoops;

        void poop() {
                numPoops = numPoops + 1;
                System.out.println("Dear mother I have pooped!!!");
        }

        void sayHi(){
                System.out.println("Gu Gu Ta Ta, I'm " + name);
        }

        void eat(double foodWeight) {
                weight = weight + foodWeight;
        }
}
```

Using Classes

Using Classes

- Class instantiation → Object creation

```
public class Baby { … }

public class PlayingWithBabies {
	public static void main(String[] args) {
		// Let's make a baby!!
		Baby miBaby = new Baby("Iker", true, 2.400);

		// Let's make the Simpson
		Baby maggie = new Baby("Maggie Simpson", false);
		Baby bart = new Baby("Bart Simpson", true);
	}

}
```

Accessing fields

- Object.FIELDNAME

```
public class Baby { … }

public class PlayingWithBabies {
	public static void main(String[] args) {
		// Let's make a baby!!
		Baby miBaby = new Baby("Iker", true, 2.400);

		// Let's make the Simpson
		Baby maggie = new Baby("Maggie Simpson", false);
		Baby bart = new Baby("Bart Simpson", true);

		// Let's ask them about their names
		System.out.println(maggie.name);
		System.out.println(bart.name);
	}

}
```

Calling METHODS

- Object.METHODNAME

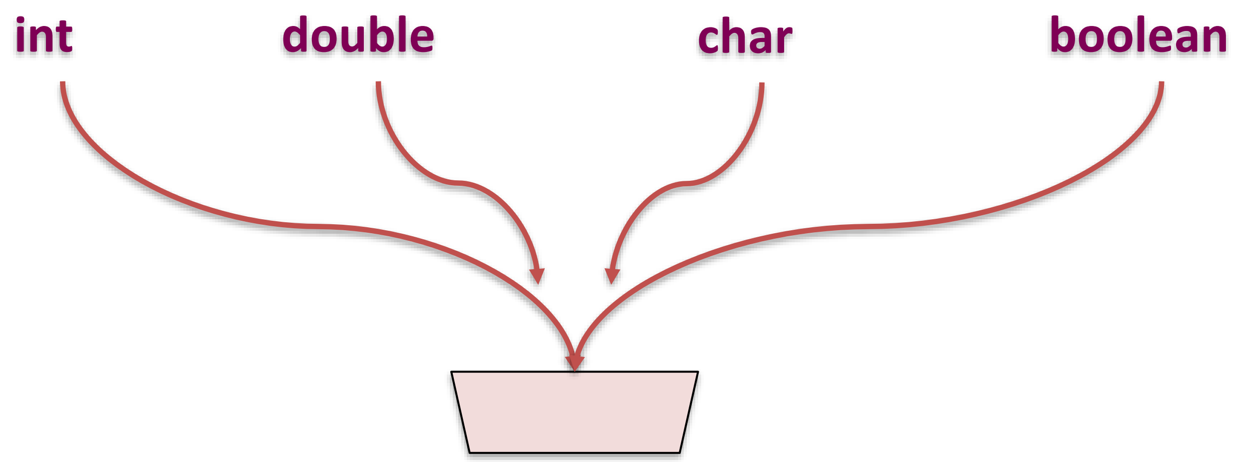
```java
public class Baby { … }

public class PlayingWithBabies {
	public static void main(String[] args) {
		// Let's make a baby!!
		Baby miBaby = new Baby("Iker", true, 2.400);

		// Let's make the Simpson
		Baby maggie = new Baby("Maggie Simpson", false);
		Baby bart = new Baby("Bart Simpson", true);

		// Let's ask them about their names
		System.out.println(maggie.name);
		System.out.println(bart.name);
		// Let's make them do something
		maggie.sayHi();
		bart.eat(1);
	}
}
```

References vs Values

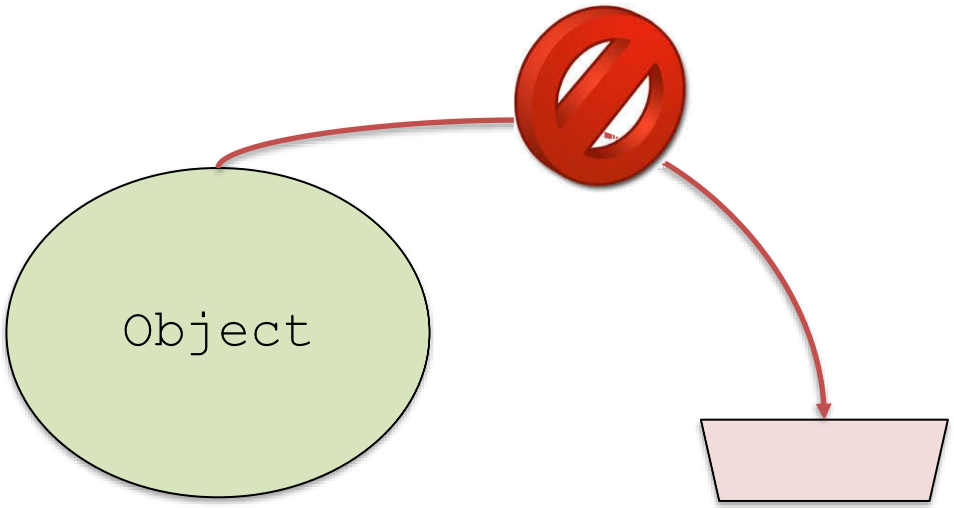
Primitives vs References

- **Primitive types are basic java types**
  - int, long, double, boolean, char, short, byte, float
  - The actual values are stored in the variable
- **Reference types are arrays and objects**
  - A reference variable does not store any value but a pointer to a value or set of values
  - Access to the value requires from some operator
    - ([ ] | .)

- **Variables are like fixed size cups**
  - Primitives are small enough that they just fit into the cup

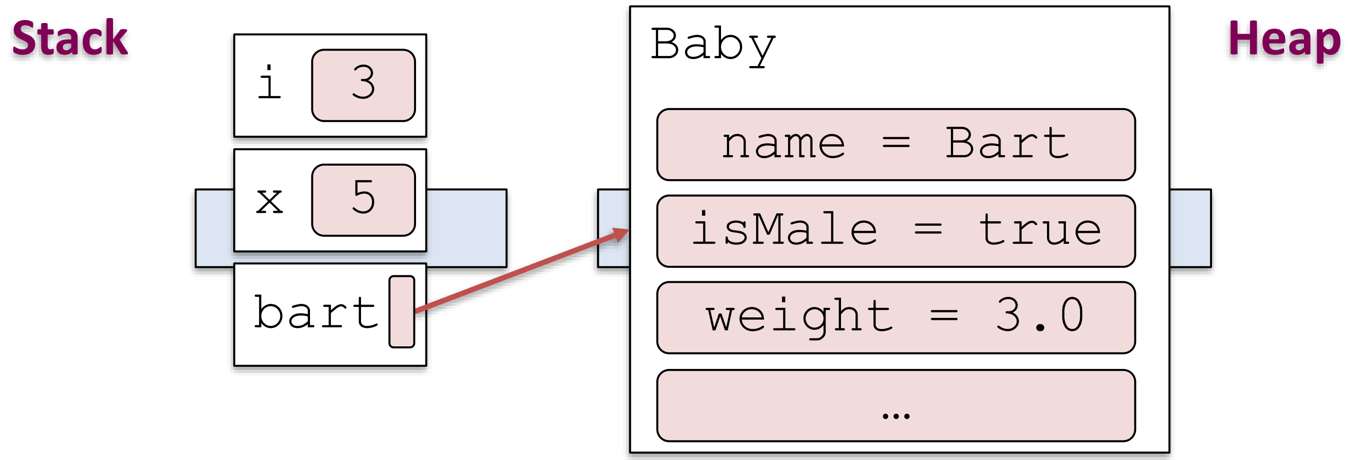


- **Objects are too big to fit in a variable**
  - Stored somewhere else (in memory → heap)
  - Variable (stack) stores a number that locates the object

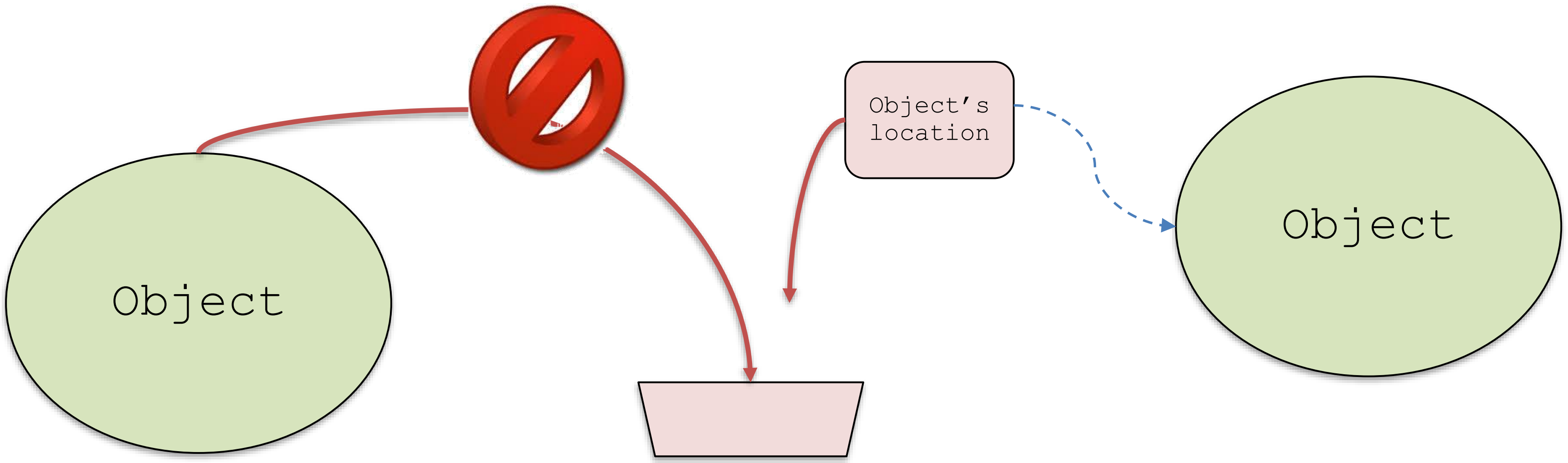


How java stores objects

- **Objects are too big to fit in a variable**
  - Stored somewhere else (in memory → heap)
  - Variable (stack) stores a number that locates the object

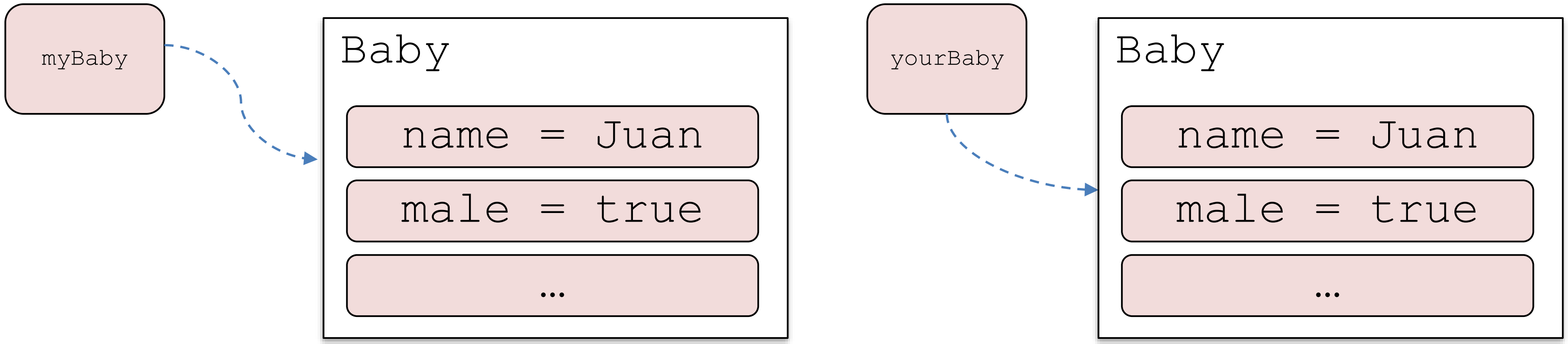


- **Objects are too big to fit in a variable**
  - Stored somewhere else (in memory → heap)
  - Variable (stack) stores a number that locates the object



- **The object's location is called a reference**
  - Operator **==** compares the reference
    - That is to say, if both variables point to the same object …

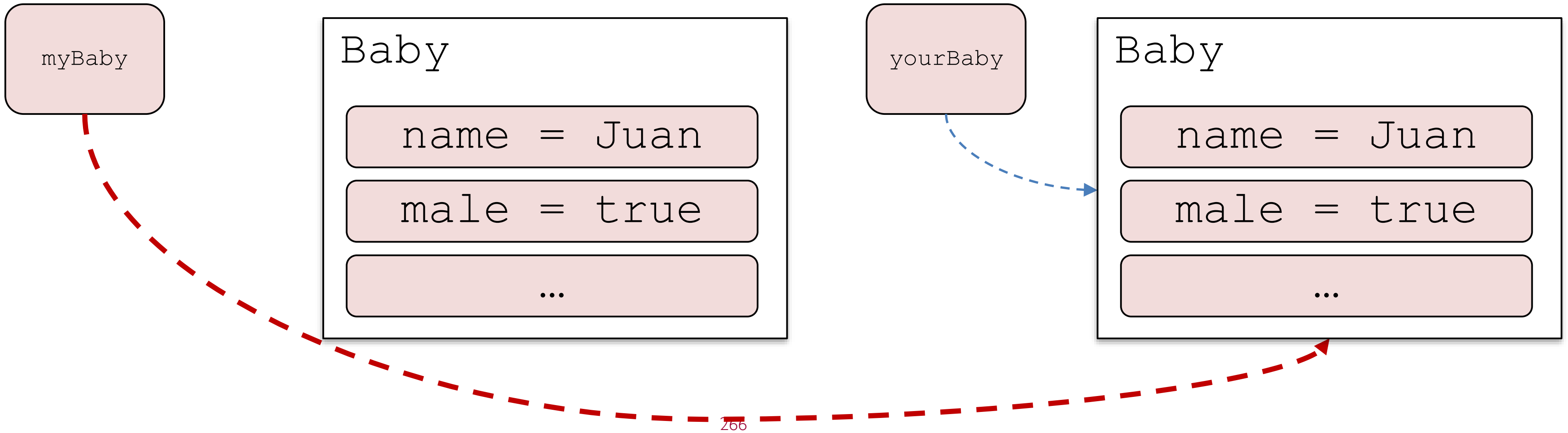
```
Baby myBaby = new Baby("Juan", false);
Baby yourBaby = new Baby("Juan", true);

myBaby == yourBaby ¿? // NO
```



References

- **The object's location is called a reference**
  - Operator **=** updates the reference

```
Baby myBaby = new Baby("Juan", false);
Baby yourBaby = new Baby("Juan", true);

myBaby = yourBaby ¿? // NO
```



References VS Values (summary)

- =
  - Copy the content from the variable on the right to the one on the left
    - Primitive types: actual value is copied
    - Reference types: object location is copied
      - The object is not duplicated but a new alias to access it is created
- ==
  - Compares the content of both variables
    - Primitie values: actual values are compared
    - Reference types: locations are compared
      - Objects' state is not compared
- [] | .
  - Navigates the reference until the referenced object
  - Can update the object state (FIELDs' values) but not the reference

Imagine
– Following directions to a house
– Moving the furniture around
Analogous to
– Following the reference to an object
– Changing fields in the object

- **Parameters passing**
  - Formal parameter : the one specified (including its type) in the method signature
  - Actual parameter: the one used when the method is called (variable)
  - When the method is called, the content of the actual parameter is copied to the formal parameter (pass-by-value)

```
static void doSomething(int x, int[] list, Baby b) {
        x = 99;
        lista[0] = 99;
        b.weight = 99;
}
...

int v = 0;
int[] values = {0, 0};
Baby miBaby = new Baby("Juan", true, 3.25);
doSomething(v, values, miBaby);
```

METHODS and References

- **Parameters passing (II)**
  - Primitive types: value is copied
  - Reference types: location is copied
    - The object is not copied (a new alias is created)

```
static void doSomething(int x, int[] list, Baby b) {
        x = 99;
        lista[0] = 99;
        b.weight = 99;
}
...

int v = 0;
int[] values = {0, 0};
Baby miBaby = new Baby("Juan", true, 3.25);
doSomething(v, values, miBaby);
```

- **Implications**
  - Primitive types: modifications over the formal parameter does not affect the actual one
  - Reference types: modifications over the location of the formal parameter does not affect the actual one but … alterying the object referenced by the formal parameter does affect the one referenced by the actual one (since there both the same)

Static Methods and Types

Static

- **Aplies to FIELDS and METHODS**
  - Means the field/method
    - Is defined for the class declaration
    - Is not unique for each instance
  - Commonly used for
    - Carrying out COMMON operations and/or data storage that apply to every object of the class
  - We refer to them
    - In the class in which they are declared: using their name
    - In another class: preceding their name with that of the class in which they were declared

Static

- **Static Methods**
  - Behave the same, despite the particular class instance (current state of the object)
    - E.g.: method to generate ramdon numbers
- **Static Fields**
  - Filed value does not depend on any particular objects: the same value for EVERY object of the class
  - There is not a copy of the value for each object but a sole copy which is shared by all the objects of the class
    - E.g.: counting class instances

… also known as "Class members"

Static Fields

- **If we want to control the number of births …**
  - We could increase the value of the field at every constructor (every method that can create new babies)

```
public class Baby{
        static int numBirths = 0;
        // initialized first time the Class is instantiated

        Baby () {
                numBirths += 1;
                …
        }

        Baby (String name) {
                numBirths += 1;
                …
        }
…
}
```

- **If we want to control the number of births …**
  - Or doing so explicitly every time the Class is instantiated by creating a new Baby

```
public class Baby{
        static int numBirths = 0;
        // initialized first time the Class is instantiated

        Baby () {
                numBirths += 1;
                …
        }
…
}
…
Baby myBaby = new Baby();
Baby.numBirths += 1;
```

- **Instance Vs Class Methods**
  - An instance method is always called over an instance of the Class (an object)
  - A class method can be called even without having instantiated the Class once

```
public class Baby {
        static void cry(Baby aBaby) {
                System.out.println(aBaby.name + " cries");
        }
}

public class Baby {
        void cry() {
                System.out.println(name + " cries");
        }
}
```

Static Methods

- **Class Methods limitations**
  - Can not access instance fields.
  - Can not call instance methods

```
public class Baby {
        String name = "Juan";
        static void whoAmI(){
                System.out.println(name);
        }
}
```

The name of which of the different baby objects created so far would be printed??

Main Method

```
public static void main(String[] arguments) { }
```

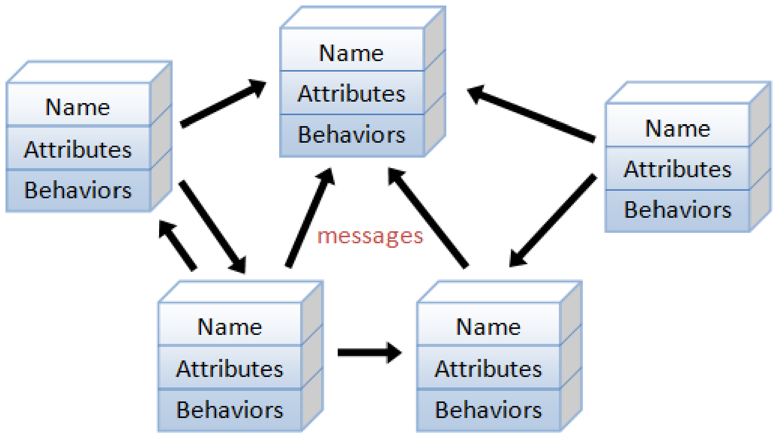
- **Application launcher**
  - Always a static method
    - When called there is no object since program execution has not started yet
  - Its duty is to start execution → void
  - String array passed as arguments when called
    - Start the program with additional arguments
      - E.g.: game for 1 or 2 players | Color spectrum …

Main method rol is not to contain program logic but create objects and call their methods

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

C:\Documents and Settings\Administrator> java Juego 2 Rojo
```

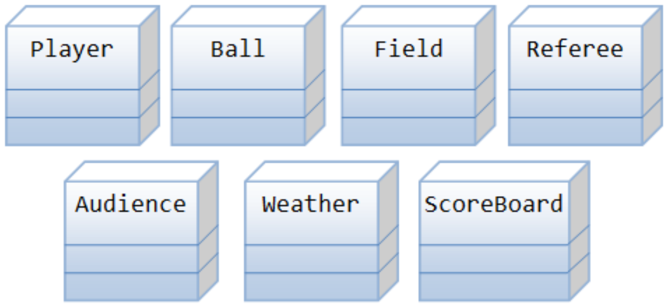
Summary (OOP)

- **Objects have state**
  - Fields
- **Objects perform actions**
  - Methods
- **Objects are build upon each other**
  - Inheritance (next chapter …)



An object-oriented program consists of many well-encapsulated objects and interacting with each other by sending messages

- **Model a computer soccer game accordingly to the "real things" that appear in the soccer games.**
  - Player: attributes include name, number, location in the field, and etc; operations include run, jump, kick-the-ball, and etc.
  - Ball
  - Referee
- **Some of these classes (such as Ball and Audience) can be reused in another application**



Classes (Entities) in a Computer Soccer Game

V: Classes and Objects

Research Group

ISe
Ingeniería de Servicios
www.ssme.es

Universidad Rey Juan Carlos

VI: Access control, Class Scope, Packages, Java API

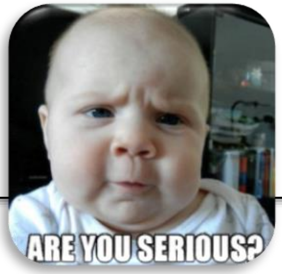
Overview

- Access control
- Class scope
- Packages
- Java API

Access Control

```
public class Person {
  String name;
  int age = 0;
}
```

```
/**
•   Main class of the Java program.
•   */
public class Main {
   public static void main(String[] args) {
       Person bart = new Person();
                bart.name = "Bart Simpson";
                bart.age = -56;
   }
}
```



Access Control

```
public class Person {
  private String name;
  private int age = 0;

  public void setAge(int new_age) {
            if (age >= 0 && age <= 120)
       age = new_age;
     else
       System.out.println("Wrong Age");
  }

  public int getAge() {
     return age;
  }
  public void setName (String new_name) {
     name = new_name;
  }
  public String getName() {
     return name;
  }
}
```

- Encapsulation

```
/**
•    Main class of the Java program.
•    */
public class Main {
   public static void main(String[] args) {
        Person bart = new Person();
                bart.setName("Bart Simpson”);
                bart.setAge(-56);
   }
}
```

Access Modifiers

- **Public: others can use this**
- **Private: only the class can use this**
  – public/private applies to any field or method
- **Why Access Control**
  - Protect private information (sorta)
  - Clarify how others should use your class
  - Keep implementation separate from interface

Overview

- Access control
- Class scope
- Packages
- Java API

Scope Review

```java
public class ScopeReview {
  void scopeMethod(int var1) {
    String var2;
    if (var1 > 0) {
      var2 = "above 0";
    } else {
      var2 = "less than or equal to 0";
    }
    System.out.println(var2);
  }
}
```

Scope review

```java
public class ScopeReview {
   private int var3;
   void scopeMethod(int var1) {
      var3 = var1;
      String var2;
      if (var1 > 0) {
         var2 = "above 0";
      } else {
         var2 = "less than or equal to 0";
      }
      System.out.println(var2);
   }
}
```

Class Scope

```
public class ScopeReview {
  private int var3;
  void scopeMethod(int var1) {
    var3 = var1;
    String var2;
    if (var1 > 0) {
      var2 = "above 0";
    } else {
      var2 = "less than or equal to 0";
    }
    System.out.println(var2);
  }
}
```

Scope

- **Just like methods, variables are accesible inside {}**
  - Previous lessons: method-level scope

```
void method(int arg1) {
	int arg2 = arg1 + 1;
}
```

  - This lesson: class-level scope

```
class Example {
	int memberVariable;
	void setVariable(int newVal) {
		memberVariable += newVal;
	}
}
```

Scope

```java
public class Baby {
   int servings;
   void feed(int servings) {
      servings = servings + servings;
   }
   void poop() {
      System.out.println("All better!");
      servings = 0;
   }
}
```

Only method-level 'servings' is updated

'this' keyword

- Clarifies scope
- Means 'my object'

```
class Example {
        int memberVariable;
        void setVariable(int newVal) {
                this.memberVariable += newVal;
        }
}
```

Scope

```
public class Baby {
   int servings;
   void feed(int servings) {
      servings = servings + servings;
   }
   void poop() {
      System.out.println("All better!");
      servings = 0;
   }
}
```

Only method-level 'servings' is updated

Scope – Using THIS

```java
public class Baby {
  int servings;
  void feed(int servings) {
    this.servings =
      this.servings + servings;
  }
  void poop() {
    System.out.println("All better!");
    servings = 0;
  }
}
```

Object-level 'servings' is updated

Overview

- Access control
- Class scope
- Packages
- Java API

Packages

- Each class belongs to a package
- Classes in the same package serve a similar purpose
- Packages are just directories
- Classes in other packages need to be imported

```
import java.util.Scanner

(...)
```

Defining Packages

- Defining packages

```
package path.to.package.foo;
class Foo {
    ...
}
```

- Using packages

```
import path.to.package.foo.Foo;
import path.to.package.foo.*;
```

Declaring classes inside packages

```
package parenttols;

public class BabyFood {

}
```

```
package parenttols;

public class Baby {

}
```

Using previously defined packages

```java
package adult;

import parenttools.Baby;
import parenttools.BabyFood;

public class Parent {
 public static void main(String[] args) {
    Baby baby = new Baby();
    baby.feed(new BabyFood());
 }
}
```

Why Packages?

- Combine similar functionality
  - org.boston.libraries.Library
  - org.boston.libraries.Book
- Separate similar names
  - shopping.List
  - packing.List

Special Packages

- All classes "see" classes in the same package
  - (no import needed)

- All classes "see" classes in java.lang
  - Example:
    - java.lang.String;
    - java.lang.System

Overview

- Access control
- Class scope
- Packages
- Java API

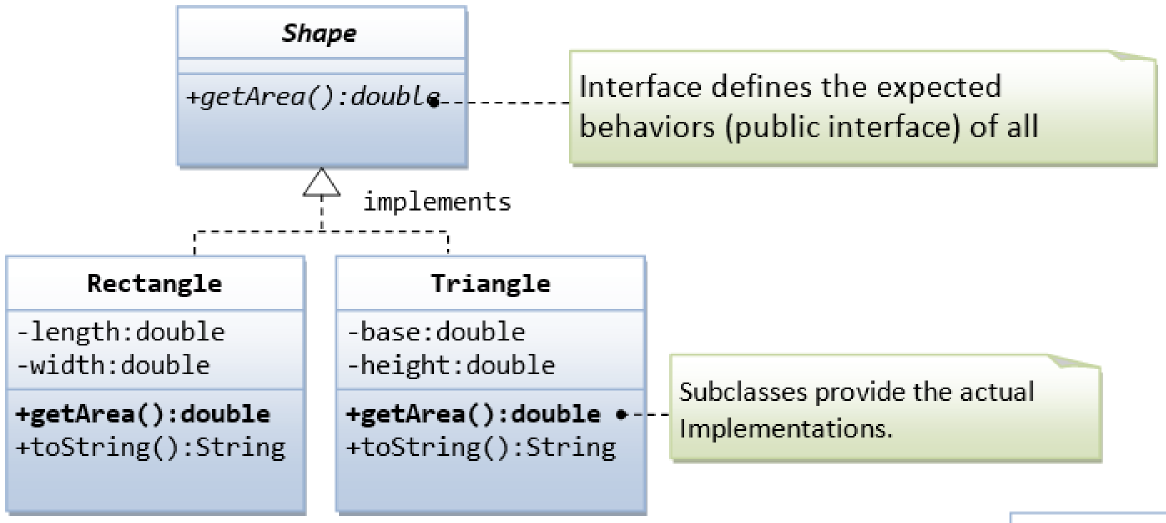
Java API

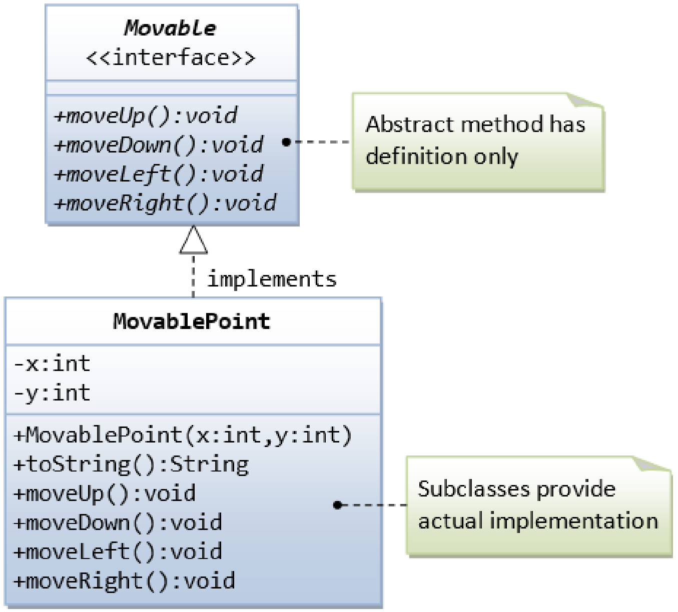
- Java includes lots of packages/classes
- Reuse classes to avoid extra work
- Java API versión 8:
  - http://docs.oracle.com/javase/8/docs/api/

Interfaces

- Set of classes that share methods
- Declare an *interface* with the common methods
- Can use the interface, without knowing an object's specific type
- Only have methods (mostly true)
- Do not provide code, only the definition (called *signatures*)
- A class can implement any number of interfaces

Interfaces





http://www.ntu.edu.sg/home/ehchua/programming/java/j3b_oopinheritancepolymorphism.html#zz-5.

Implementing Interfaces

- Implementations provide complete methods

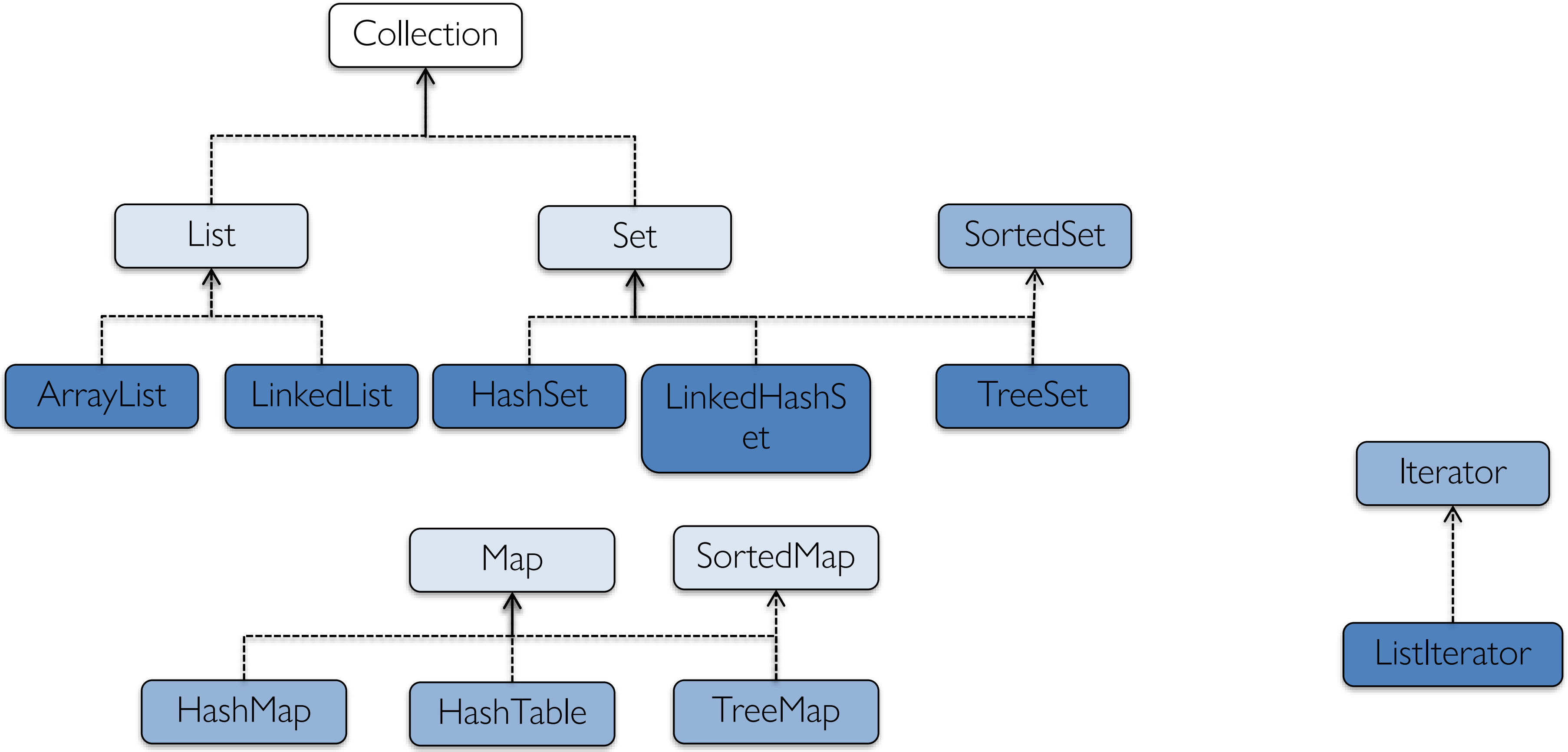
```
interface Shape {
   int area();
   int perimetre();
   void setColor(Color color);
}
```

```
class Square implements Shape{
   private side = 0;

   public int area() {
         return side * side;
   }
   public int perimetre() {
         return side * 4;
   }
}
```

Collections

- An object that groups multiple items into a single unit.
- Aggregate data items that are normally considered together as a whole
  - A poker hand (collection of cards)
  - A folder (collection of files)
  - A team (collection of players)

- Java Collections Framework



Collection Interface

- Group together common methods to handle groups of objects
  - add (Object x)
  - remove (Object x)
  - contains (Object x)
  - size (Object x)
  - toArray (Object x)
  - Iterator (Object x)

List

- **Redimensionable ordered list of elements**
- **Repatead values are allowed**
  - ArrayList
  - LinkedList
    - Keep insert order
    - Hampers performance

- add(Object o)
- add(int indice, Object o)
- get(int indice)
- remove(int indice)
- clear()
- indexOf(Object o)
- lastIndexOf(Object o)
- size()
- contains(Object o)

- List

```
import java.util.ArrayList;
class ArrayListExample {
   public static void main(String[] arguments) {
         ArrayList<String> cadenas = new ArrayList<String>();
         cadenas.add("Ignacio");
         cadenas.add("José");
         cadenas.add("Clara");
         System.out.println(cadenas.size());
         System.out.println(cadenas.get(0));
         System.out.println(cadenas.get(1));
         cadenas.set(0, "Adiós");
         cadenas.remove(1);
         for (int i=0; i < cadenas.size(); i++){
                   System.out.println(cadenas.get(i));
         }
         for (String s : cadenas) {
                  System.out.println(s);
         }
   }
}
```

Arrays with Items

- Adding books to the Books array

```
Book[] books = {}
```

  - Create the array bigger than needed

```
Book[] books = new Book[100]
```

  - Create an auxiliary copy, etc.
  - Use an ArrayList

```
ArrayList<Book> books = new ArrayList<Book>();
Book b = new Book(“El Quijote);
books.add(b);
```

Sets

- **Like an ArrayList, but**
  - Only one copy of each object
    - equals() | hashcode()
  - No array index
- **Features**
  - Add objects to the set
  - Remove objects from the set
  - Is an object in the set?

- add(Object o)
- remove(Object o)
- clear()
- isEmpty()
- iterator()
- size()

Sets

- **HashSet**
  - Best performance
- **LinkedInHashSet**
  - Performance: worse than HashSet
  - Keep insertion order
- **TreeSet**
  - Sorted (lowest to higest) set of Comparable items
  - Worst performance

Sets

```
import java.util.TreeSet;
  class TreeSetExample {
   public static void main(String[] arguments) {
   TreeSet<String> cadenas = new TreeSet<String>();
   cadenas.add("Cristian");
   cadenas.add("Andrés");
   cadenas.add("Tania");
   System.out.println(cadenas.first());
   System.out.println(cadenas.last());
   System.out.println(cadenas.size());
   cadenas.remove("Tania");
   for (String s : cadenas) {
       System.out.println(s);
   }
  }
}
```

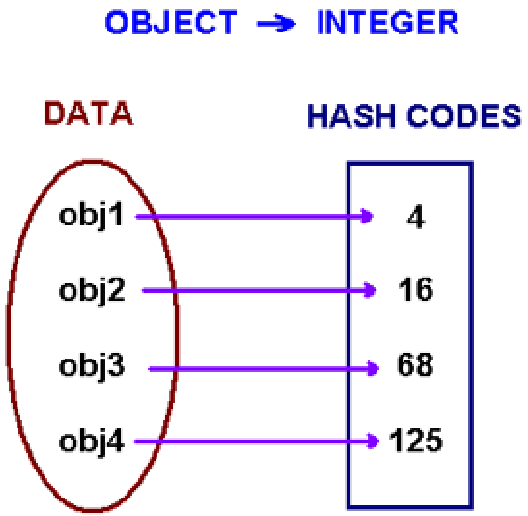
Maps

- **Stores a (key, value) pair of objects**
  - AKA 2 cols table
  - Look up the key, get back the value
  - No duplicate keys
  - One value per key
- **Examples: Address Book**
  - Map from names to email addresses

- clear()
- containsKey(Object o)
- containsValue(Object o)
- get (Object key)
- isEmpty()
- remove(Object key)
- size()

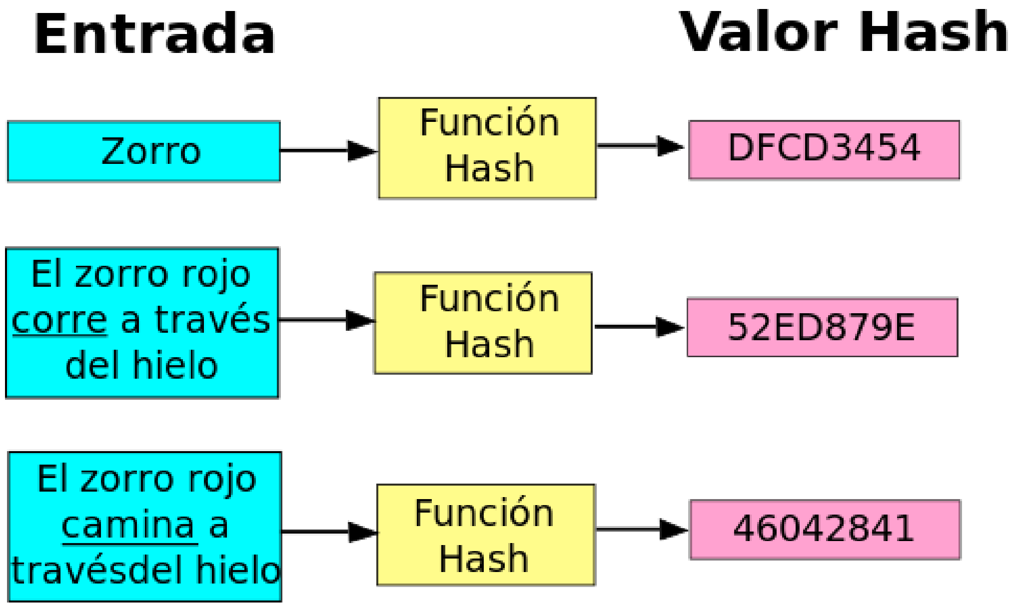
Hashing

- A way to speed up searching
  - Instead of traversing the entire list
  - Use a *magic* function which yields the element index
- Hash function
  - Given a key, generates the address in the table



Hash function

- From a given input, a Hash función produce a value from a finite ouput range (maps the key into an index)
  - Usually a fixed length text character



- **HashMap / Hashtable**
  - Unordered (pseudo-random)
  - Best performance
  - [Hashtable: no null values]
- **LinkedHashMap**
  - Keep insert order
  - Performance: worse than HashMap
- **TreeMap**
  - Sorted (lowest to highest value)
  - Worst performance

Maps

```java
import java.util.Hashtable;
import java.util.Map.Entry;
  class HashMapExample {
    public static void main(String[] arguments) {
    Hashtable<String, String> cadenas = new Hashtable<String, String>();
    cadenas.put("Álvaro", "alv@urjc.es");
    cadenas.put("Carolina", "caro@urjc.es");
    cadenas.put("Saúl", "saul@urjc.es");
    System.out.println(cadenas.size());
    cadenas.remove("Álvaro");
    System.out.println(cadenas.get("Carolina"));
    for (String s : cadenas.keySet()) {
         System.out.println(s);
    }
    for (String s : cadenas.values()) {
         System.out.println(s); }
    for (Entry<String, String> pairs : cadenas.entrySet()) {
         System.out.println(pairs);
    }
  }
}
```

Hashtable

- A Java Hashtable is a data structure that uses a hash function to identify data through a key
  – Each key corresponds to a given value
  – No index

Clave	Valor
Marvel-234	Spiderman
DCComics-567	Batman
Marvel-768	Hulk
DCComics-987	Superman

Don't need to know the insides of a hash function

- Declaration

```
Hashtable<String,String> heroes =
                    new Hashtable<String,String>();
```

- `put(key, value)`
  - Insert value and assigns the provided key to it

```
heroes.put("Marvel-234", "Spiderman");
heroes.put("DCComics-567", "Batman");
heroes.put("Marvel-768", "Hulk");
heroes.put("DCComics-987", "Superman");
// 1st element if the key
// 2nd element is the value
```

Clave	Valor
Marvel-234	Spiderman
DCComics-567	Batman
Marvel-768	Hulk
DCComics-987	Superman

Hashtable

- `get(key)`
  - returns the value corresponding to the key provided

```
String heroe1 = heroes.get("Marvel-234");
// heroe1 variable contains "Spiderman"

String heroe2 = heroes.get("Marvel-768");
// heroe2 variable contains "Hulk"

System.out.println("Marvel heroes are: " +
                       heroe1 + " & " + heroe2);

// prints:
// Marvel heroes are Spiderman & Hulk
```

Clave	Valor
Marvel-234	Spiderman
DCComics-567	Batman
Marvel-768	Hulk
DCComics-987	Superman

Hashtable

- The values in the hashtable are to be traversed by means of an `Enumeration` object

```
Enumeration<String> miEnumHeroes = heroes.elements();

while (miEnumHeroes.hasMoreElements()){
  System.out.println(”Hero: " + miEnumHeroes.nextElement());
}
// prints the four values
```

  - elements()
    - Returns the values in the hashtable
  - hasMoreElements()
    - Yields true if more elements remain in the Enumeration object
  - nextElement()
    - Returns the next element in the Enumeration object

Clave	Valor
Marvel-234	Spiderman
DCComics-567	Batman
Marvel-768	Hulk
DCComics-987	Superman

- keys()
  - Returns the keys of the hashtable

```
// Must import the follwing classess
import java.util.Enumeration;
import java.util.Hashtable;


Enumeration<String> miEnumClaves = heroes.keys();

while (miEnumClaves.hasMoreElements()){
  System.out.println("Clave: " +
miEnumClaves.nextElement());
}
// prints the four keys:
// Clave: Marvel-234
// Clave: DCComics-567
// Clave: Marvel-768
// Clave: DCComics-987
```

Clave	Valor
Marvel-234	Spiderman
DCComics-567	Batman
Marvel-768	Hulk
DCComics-987	Superman

Hashtable

- **Hashtable methods (summary)**
  - put(key, value)
    - Adds a new <key,value> pair
  - remove(key)
    - Deletes the given <key,value> pair
  - get(key)
    - Returns the value corresponding to the given key
  - containsKey(key)
    - Yields true if key is in the Hashtable
  - contains(value)
    - Yields true if value is in the Hashtable
  - size()
    - Returns the number of <key,value> pairs in the Hashtable

- Ejemplo

```
heroes.remove("DCComics-567");
// deletes the <DCComics-567,Batman> pair

System.out.println("Amount of heroes: " + heroes.size());
// prints 3

String searchKey = "DCComics-987";

System.out.print ("Hero " + heroes.get(searchKey)

if (heroes.containsKey(searchKey)) {
   System.out.println(" is in the table");
} else {
   System.out.println("is NOT in the table");
}
```

Clave	Valor
Marvel-234	Spiderman
~~DCComics-567~~	~~Batman~~
Marvel-768	Hulk
DCComics-987	Superman

Hashtable as Objects container

- Hashtable objects can be used as well to store any given type of objects

Hashtable as Objects container

```
import java.time.LocalDateTime;
import java.time.temporal.ChronoUnit;

public class Persona {
  String nombre;
  String apellido;
  String DNI;
  LocalDateTime nacimiento;

  Persona(String nom, String ape, String dni, LocalDateTime naci){
    this.nombre = nom;
    this.apellido = ape;
    this.DNI = dni;
    this.nacimiento = naci;
  }

  long obtenerEdad(){
    LocalDateTime fechaActual = LocalDateTime.now(); // devuelve la fecha y hora actual
    long anyos = nacimiento.until(fechaActual, ChronoUnit.YEARS);
    return anyos;
  }
}
```

java.time: available from Java 8 on (improvements for dates handling)

- LocalDateTime = date & time
- LocalDate = date (wout time)

DATE - until()
Two dates diff
(MINUTES, DAYS, MONTHS, YEARS, ETC.)

Hashtable as Objects container

```
import java.time.LocalDateTime;

public class DemoPersona {
  public static void main(String args[]){

    LocalDateTime naci = LocalDateTime.of(1980, 6, 30, 13, 00); // 30/6/1980 13:00
    Persona p1 = new Persona("Pepe", "Perez", “123654A", naci);
    System.out.println("La edad de " + p1.nombre + " es: " + p1.obtenerEdad());
    // prints: La edad de Pepe es: 35
  }
}
```

Hashtable as Objects container

```
import java.time.LocalDateTime;
import java.util.Enumeration;
import java.util.Hashtable;

public class DemoPersonasHash {

 public static void main(String args[]){
  Hashtable<String,Persona> personas = new Hashtable<String,Persona>();
  Persona pepe = new Persona("Pepe", "Perez", "123654A", LocalDateTime.of(1980, 6, 30, 13, 00));
  Persona maria = new Persona("Maria", "Gomez", "789369D", LocalDateTime.of(1975, 8, 28, 15, 30));
  Persona andres = new Persona("Andres", "Gonzalez", "741852R", LocalDateTime.of(1988, 10, 30, 19, 15));
  Persona angel = new Persona("Angel", "Urbino", "357159P", LocalDateTime.of(1993, 3, 25, 8, 45));
  // Persona objects are added using their DNI as key to identify them
  personas.put("123654A", pepe);
  personas.put("789369D", maria);
  personas.put("741852R", andres);
  personas.put("357159P", angel);

  Enumeration<Persona> miEnumPersonas = personas.elements(); // ready to traverse the Persona table

  while (miEnumPersonas.hasMoreElements()){
   Persona p1 = miEnumPersonas.nextElement();
   System.out.println(p1.nombre + " tiene " + p1.obtenerEdad() + " años");
  }
 }
}
```

Hashtable <DNI,Persona>

Prints

```
Pepe tiene 35 años
Maria tiene 40 años
Andres tiene 27 años
Angel tiene 22 años
```

Hashtable as Objects container

- What if we want to check whether a DNI is in the table?

```
String DNIbuscado = "741852R";

if (personas.containsKey(DNIbuscado)) {
   System.out.println(personas.get(DNIbuscado).nombre + ", con DNI: " +
         DNIbuscado + " está en la tabla");
} else {
   System.out.println("La persona buscada NO está en la tabla");
}
// Prints: Andres, con DNI: 741852R está en la tabla
```

Don't forget to use an Scanner object to get input data from the user
(this time ask him about the DNI to look for)

Hashtable as Objects container

- Remove a person and insert a new one

```
personas.remove("357159P");

personas.put("258654T", new Persona("Laura", "Serrano", "258654T",
                                 LocalDateTime.of(1995, 5, 30, 19, 55)));
```

- Restrict the number of Persona objects to 10

```
Persona laura = new Persona("Laura", "Serrano", "258654T",
                                 LocalDateTime.of(1995, 5, 30, 19, 55));
if (personas.size()<10){
  personas.put(laura.DNI, laura);
}
else{
  System.out.println("La tabla para almacenar personas está al completo!");
}
```

Collections - Sum Up

- **Set**
  - Unordered collection (unguaranted insertion order)
  - Does not allow for duplicates
  - Just one null ellement
- **List**
  - Ordered collection
  - Allows for duplicates
  - Multiple null elements
- **Map**
  - Unordered collection
  - Unique keys but duplicate values
  - One null key and multiple null values

Warnings

- Using TreeSet/TreeMap?
  - Read about Comparable interface
- Using HashSet/HashMap?
  - Read about equals, hashCode methods

- Note
  - This only matters for classes you build, not for java built-in types

Research Group

Ingeniería de Servicios
www.ssme.es

Universidad Rey Juan Carlos

VI: Access control, Class Scope, Packages, Java API

Ingeniería de Servicios

Universidad Rey Juan Carlos

Introduccion a la programación

Java Input - Using Java Scanner

Java Scanner

- A Java Scanner is the fastest, easiest way to get input from a user in Java.
- A class providing a number of methods to read data from the screen (or file).
- The input is considered to be a set of tokens, each one delimited by whitespaces.

```
> Hello world, this is my first Java program
```

TOKEN

Java Scanner

– nextLine(): advances to the next line and returns the content skipped (a line of text)

– next: reads the following token from the input and returns it as a String

```
System.out.println("What is your name: ");
Scanner scanner = new Scanner(System.in);
String name = scanner.nextLine();
// User enters Juan Manuel
System.out.println("Hello " + name);
// Program output: Hello Juan Manuel
```

```
System.out.println("What is your name: ");
Scanner scanner = new Scanner(System.in);
String name = scanner.next();
// User enters Juan Manuel
System.out.println("Hello " + name);
// Program output: Hello Juan
```

Java Scanner

– nextInt(): reads from the input the next token and retuns it as an integer

```
System.out.println("How old are you?: ");
Scanner scanner = new Scanner(System.in);
int age = scanner.nextInt();
// User enters 28
System.out.println("You are " + age);
// Program output: You are 28
```

– nextFloat(): reads from the input the next token and retuns it as a Float

```
System.out.println("How tall are you?: ");
Scanner scanner = new Scanner(System.in);
float height = scanner.nextFloat();
// User enters 1.82
System.out.println("You are " + height + " tall");
// Program output: You are 1.82 tall
```

Java Scanner

– nextBoolean(): reads from the input the next token and retuns it as a Boolean

```
System.out.println(“Are you married?: “);
Scanner scanner = new Scanner(System.in);
Boolean married = scanner.nextBoolean();
// User enters true
System.out.println(“Married: “ + married);
// Program output: Married: true
```

Introduccion a la programación

Java Input - Using Java Scanner

Profesores:

Juan M. Vara
juanmanuel.vara@urjc.es

David Granada
david.granada@urjc.es