Resumen de Introducción a la Programación

Introducción al Software Científico y a la Programación

Curso 2014-15

Grupo ISCyP*

Índice

*Para errata y sugerencias: ldaniel@um.es

1. Introducción a la programación

Las fases del proceso de resolución de problemas con programas son:

- **Fase de resolución del problema**, cuyo objetivo es encontrar un **algoritmo**.

  1. ***Enunciado.*** Definición clara del problema.
  2. ***Análisis.*** ¿Qué afecta? ¿Cómo se hace? Determinar los procesos, entradas y salidas.
  3. ***Diseño.*** Construir un **algoritmo**.
  4. ***Verificación.*** Comprobar que realiza las tareas para las que ha sido diseñado. Produce el resultado correcto y esperado.

  Esta fase la realizaremos con lápiz y papel.

- **Fase de implementación/programación**, en la que se programa el **algoritmo** en el ordenador.

  1. ***Codificación.*** Escribir el **algoritmo** en un lenguaje de programación.
  2. **Compilación**, para comprobar que no haya errores en el código fuente, y **ejecución**, para ver su comportamiento.
  3. ***Prueba y verificación.*** A partir de una matriz de entrada establecidas y sus salidas esperadas se determina la validez de la solución propuesta al problema.
  4. ***Depuración***, para corregir los errores encontrados.
  5. ***Documentación.*** Referenciar desde que se dio el problema hasta la solución encontrada.

  Esta fase la realizaremos en Processing salvo el último paso que se desarrollará en LaTeX.

La **programación** es un **proceso** mediante el cual creamos **programas** que son **ejecutados** por un sistema computacional, todo ello con el fin de **resolver** una tarea. Todo programa implementa un algoritmo. Un **algoritmo** es conjunto de acciones o reglas bien **definidas**, **ordenadas** y **finitas** que permite realizar una actividad mediante pasos sucesivos que **no generen dudas** a quien deba realizar dicha actividad.

La construcción de un algoritmo (y por tanto la de creación de un programa) se puede basar en varias técnicas de diseño. Destacamos.

- Top-down (Descendente). Basado en "Divide y vencerás". Descomposición del problema en partes más simples hasta llegar a subproblemas de solución inmediata (o conocida).
- Bottom Up (Ascendente). Conocidas las soluciones de algunos problemas, integrarlas para obtener la solución de nuestro problema.
- Stepwise Refinement. Primero se describen pocos pasos y de manera incompleta. Se amplían las descripciones de manera más detallada con pasos más específicos.
- Backtracking. Donde se requiere una búsqueda exhaustiva de todas las posibilidades (como en la búsqueda de la salida de un laberinto.)
- Diseño voraz. Ir directo a un solución, no necesariamente una que sea óptima.

2. Datos, operadores y variables

2.1. Tipos de datos

Dado un dato informativo, su tipo hace referencia a qué tipo de información almacena. Destacan dos grandes grupos:

- **Primitivo (simple o sencillo)**

Tipo de dato	Valor	Rango de valores	byte
boolean	Booleano	{*true*, *false*}	
byte	Numérico entero	[−128, 127]	1
char	Alfanumérico carácter	[0, 65535]	2
double	Numérico real	4.94065645841246544e-324d hasta 1.79769313486231570e+308d (positivo o negativo)	8
float	Numérico real	1.40129846432481707e-45 hasta 3.40282346638528860e+38 (positivo o negativo)	4
int	Numérico entero	[−2147483648, 2147483647]	4
long	Numérico entero	-9223372036854775808 hasta 9223372036854775807	8

- **Estructura de datos (compuesto)**
  - Strings, arrays, árboles, ...

2.2. Variables

Una variable hace referencia a un porción de memoria del ordenador donde se almacena un dato (simple o compuesto) que se identifica con un nombre. En una declaración de variable se le indica al ordenador el tipo de dato que se guardará y el nombre con el que se identificará la variable. Ejemplos son:

Sketch 1: Ejemplos de declaración de variables

```
1 boolean encontrado; // La variable almacenará el valor true o false.
2 int contador;      // La variable contador almacenará un número entero.
3 float hipotenusa; // La variable hipotenusa almacenará un número real.
4 char tecla;       // La variable tecla almacenará un carácter imprimible
```

Una instrucción de asignación de valor a una variable es la que presenta el siguiente patrón:

`identificador = expresión;`

que significa "computa el resultado final de la `expresión`, y el resultado obtenido guárdalo en la porción de memoria que le corresponde a `identificador`". P.e.: `x = x+y;` es la asignación que calcula la suma de los valores almacenados en las variables `x` e `y`, y el resultado final se guarda en la variable `x`.

2.3. Operadores

Los datos se pueden utilizar para obtener nuevos datos mediante los denominados **operadores**. Los operadores son acordes al tipo de dato. Dos números enteros se pueden sumar, pero no así dos valores booleanos; o se puede obtener el valor complementario de `true` pero no así el de un char. Destacan los **operadores aritméticos** para números y los **operadores lógicos** para los datos booleanos. Dos datos simples se pueden comparar entre sí con los **operadores de comparación** (o relacionales). Existen varios tipos de operadores y entre ellos existe una **precedencia** (unos se realizan antes que otros en el caso de que aparezcan los dos en un instrucción). La **asignación** también se contempla como un operador y siempre es el último en realizarse. En Processing el orden de prioridad (de más preferente a menos preferente) es:

1. Artiméticos

   a) Operadores posfijos : `exp++`, `exp--`
   b) Operadores unarios : `++exp`, `--exp`, `-exp`, `!`
   c) Multiplicación y división: `*`, `/`,`%`
   d) Suma y resta: `+`, `-`

2. Operadores relacionales: `<`, `>`, `<=`, `=>`
3. Lógicos
   a) Igualdad: `==`, `!=`. **Muy importante, no confundas `==` con `=`.**
   b) Operador AND: `&`
   c) Operador XOR: `^`
   d) Operador OR: `|`
   e) Operador AND: `&&`
   f) Operador OR: `||`
4. Operadores de asignación: `=`, `+=`, `-=`, `*=`, `/=`, `%=`

P.e. `x=(2*3+5 == 5)` indica que primero se hará el producto `2*3`, después la suma `6+5`, a continuación la comparación `11 == 5` y por último se hará la asignación `x=false`.

Las expresiones `x++`, `++x` y `x=x+1` incrementan en 1 el valor de la variable `x`. Realizan la misma operación aritmética pero tienen efectos diferentes cuando aparecen con otros operador. Según la expresión utilizada se pude generar una acción u otra. Lo mismo ocurre con las expresiones `x–`, `-x` y `x=x-1` que decrece en 1 el valor de la variable `x`; pero actúan en el programa de forma diferente.

2.4. Casting

Un **casting** es la conversión de un dato a un tipo de dato concreto. Existen dos tipos de casting:

- **implícito**, cuando lo realiza directamente el lenguaje de programación. P.e. la suma de dos datos tipo `char` es un dato de tipo `int`.
- **explícito**, cuando lo ordena el programador con la expresión `(tipo) expresión`.

Sketch 2: Ejemplos de casting explícitos para generar valores aleatorios

```
1 boolean encontrado =  ((int) random(0,1000) % 2 == 0);
2 int contador = (int) random(0,1000);
3 char  tecla= (char) ((byte) random(64,127));
```

3. Estructuras de control

3.1. Estructura secuencial

Un programa que se diseña únicamente con esta estructura de control constará generalmente de unas instrucciones de declaración de variables o constantes, instrucciones de asignación a partir de datos obtenidos por instrucciones de lectura de datos, instrucciones de impresión e invocación a funciones internas del lenguaje o funciones del programador. Suelen tener este aspecto:

Sketch 3: Estructura secuencial

```
final tipo_de_dato identificador;  // Declaración de constantes.
tipo_de_dato identificador;  // Declaración de variables.

identificador = valor de una expresión; // Inicialización de variables;
identificador = lectura_de_dato(); // resultado de un proceso de entrada de datos .

llamada a funciones y asignaciones para resolver el problema.

imprime (mensaje); // Muestra la salida del proceso
```

3.2. Estructuras Selectivas

Aquellas que seleccionan y ejecutan uno y solo uno de los procesos alternativos posibles cuando cumple con la condición establecida. Las condiciones de las estructuras selectivas responden a son expresiones cuyo valor final debe ser booleano (y el valor se obtiene utilizando operadores de comparación y booleanos,).

Condicional Las que realizan una serie de instrucciones siempre y cuando se cumpla una condición booleana cierta.

Sketch 4: Patrón del condicional simple

```
if ( condición ) {
    estructuras de control
    si la condición es verdadera
}
Instrucciones que siempre se hacen
```

Sketch 5: Dibuja un cuadrado ... a veces

```
int numero = 15;  // Dialog.inputInt("Dame un número entero");
size(100,100);
if (numero > 5 && numero < 20) {
  rect(20, 20, 80, 80); // A veces dibuja un cuadrado
}
line(0, 0, 100, 100); // Siempre dibuja una línea
```

Bicondicional Las que realizan una serie de instrucciones siempre y cuando se cumpla una condición booleana cierta; pero realiza otra serie de instrucciones siempre y cuando la misma condición sea falsa.

Sketch 6: Patrón del condicional doble

```
if ( condición ) {
  estructuras de control si condición es verdadera
} else {
  estructuras de control si condición es falsa
}

Estructuras de control independientemente de cumplirse la condición
```

Sketch 7: Dibuja una línea a una altura determinada con color variable

```
int numero = 15; // Dialog.inputInt("Dame un número");

if (numero < height/2) { // Si es verdad
  fill(255,0,0);          // dibuja un línea roja
  line(30, numero, width-30, numero);
} else {                 // pero si es falso
  fill(0,0,255);          // dibuja una línea azul
  line(0, numero, width, numero);
}
rect(width/2-10,height/2-10, 20 , 20); // Siempre dibuja un rectángulo
```

Alternativa múltiple Ejecuta uno de entre varios procesos de acuerdo al resultado de la evaluación de una expresión que se compara con valores alternativos. El tipo de dato de la expresión solo puede ser entero o carácter y, por lo tanto, las opciones deberán coincidir con el tipo de dato de la expresión: si la evaluación es un entero entonces el tipo de dato de las opciones serán un entero, y si la evaluación es un carácter entonces el tipo de dato de las opciones serán un carácter.

Sketch 8: Patrón del condicional múltiple

```
switch (expresión) {
  case A:
    estructura de control para A
    break;
  case B:
    estructura de control para B
    break;
  ....
  case N:
    estructura de control para N
    break;
  default:
    estructura de control para N+1
    break;
}

Estructura de control que se realiza independientemente del valor de expresión.
```

Sketch 9: Calcula datos de un círculo

```
// Declaramos una variable que almacenará la solución al problema.
// Damos una inicialización con un valor imposible para el problema.
float solucion = -1.0;

// Solicitamos el radio del círculo
float radio = 10.0; // Dialog.inputFloat("Dime el radio del círculo");

// Mostramos el menú
println("Selecciona una opción:");
println("a) Calcular el diámetro.");
println("b) Calcular el perímetro.");
println("c) Calcular el área.");

// Pedimos que seleccionen una opción
char opcion = 'a'; // Dialog.inputChar("Dame una opción");

switch(opcion) {
case 'a':
  print("El diámetro es: ");
  solucion = 2*radio;
  break;
case 'b':
  print("El perímetro es: ");
  solucion = 2*PI*radio;
```

```
  break;
case 'c':
  print("El área es: ");
  solucion = PI*sq(radio);
  break;
default:
  print("## Error: Opción desconocida.");
  break;
}

// Observa que si la opción es incorrecta la solución será -1

println(solucion); // Siempre muestra este mensaje
```

3.3. Estructuras Repetitivas

Aquellas que repiten una y otras vez de forma reiterativa una secuencia de instrucciones mientras que cierta condición sea cierta. Cuando la condición cambia su valor y deja de ser cierta, dejan de ejecutarse dicha secuencia de instrucciones.

Estructura `while`

Es la estructura que responde algorítmicamente al concepto de estructura repetitiva. Es la estructura esencial en cualquier lenguaje de programación. Cualquier otra que pueda existir es una leve modificación de esta estructura.

Sketch 10: Patrón general de `while`

```
Iniciar variable de control de la condición.
while (condición)  {
  instrucciones A
  actualizar variable de control
}
instrucciones B
```

Cada vez que se ejecutan las instrucciones se realiza una iteración. Una **iteración** es la repetición de un proceso dentro de un programa informático. La condición, que es un valor booleano, depende de una variable que puede ser booleana o no.

- Dicha variable, llamada centinela o de control, deberá inicialilzarse a un valor acorde a su tipo de dato y antes del inicio de la estructura repetitiva.
- La condición responde a una expresión que, utilizando la variable de control, retorna un valor booleano.
- En el cuerpo deberá de existir alguna instrucción que modifique el valor de la variable de control para que a su vez modifique el valor de la condición de control a efectos de que la estructura repetitiva no entre en un ciclo sin fin.

Sketch 11: Suma de n naturales

```
int numero = 10;   // Número natural que iremos sumando
                   // Actuará de variable de control
int elfinal = 32; // Último número natural que se sumará
int suma = 0;      // Acumulador de la suma.
while ( numero <= elfinal) {    // Mientras no se llegue al último número
  suma = suma + numero;         // se suma el número actual
  numero = numero + 1;          // pasamos al siguiente número
}
print("La suma de los "); print(n);  // Se muestra el resultado
print(" primeros números naturales es: ");
println(suma);
```

Sketch 12: Suma de n naturales con un contador

```
int numero = 10;    // Número natural por el que empezamos la suma
int elfinal = 32;   // Último número natural que se sumará
int suma = 0;       // Acumulador de la suma.

int contador=0;    // Número de veces que se ha realizado la iteración
                   // Actuará de variable de control
int pasos=elfinal-numero+1; // Número de pasos que se realizará la suma.

while ( contador < pasos ) { // Mientras no hayamos recorrido todos los números
  suma = suma + numero;     // calcula la suma
  numero = numero + 1;      // numero = siguiente(numero); No es un contador.
  contador = contador + 1; // Cuento la iteración. Sí es un contador.
}

print("La suma de los números entre "); print(numero);    // Se muestra el resultado
print(" y "); print(elfinal);
print("  es: ");
println(suma);
```

Estructura `do-while`

Es una estructura `while` en la que las instrucciones del cuerpo se ejecutan al menos una vez.

Sketch 13: Patrón general de `do-while`

```
Iniciar variable de control de la condición.
do {
  instrucciones A
  actualizar variable de control de la condición
} while (condición);
instrucciones B
```

Sketch 14: Suma de números naturales hasta

```
boolean continuar;    // Variable de control que también será la condición
int suma = 0;         // Acumulador de una suma
int numero = 0;       // Número de inicio por el que se empezará la suma
do {
  suma = suma + numero;    // Se calcula la suma acumulada
  numero = numero + 1;     // Pasamos al número siguiente
  continuar = (((int) random(0,100)) %  2 == 0); // Dialog.inputBoolean("¿Continúas?);
} while (continuar);
print("La suma es ");     // Mostramos el resultado
println(suma);
```

Estructura `for-to`

Es una estructura `while` en la que las instrucciones del cuerpo se ejecutan un número determinado de veces

```
for ( expresión_1; expresión_2; expresión_3) {
   instrucciones A
}
instrucciones B
```

- `expresión_1` está formada por la declaración e inicialización de la variable de control de tipo entero. Incluye una asignación. También se llama **parte de inicialización**.
- `expresión_2` establece la condición de control (número de iteraciones a realizar). Es una expresión booleana. También se llama **parte de iteración**.
- `expresión_3` indica el incremento o disminución de la variable de control. Incluye una asignación. También se llama **parte de incremento**.

- `instrucciones A` son las instrucciones que se repetirán mientras se cumplan las condiciones de control.
- `instrucciones B` son las instrucciones que se realizarán finalizada la estructura repetitiva.

Sketch 16: Sumar números naturales entre dos valores dados.

```
int minimo = 100;     // Extremo inferior
int maximo = 174;     // Extremo superior
int suma = 0;         // La suma acumulada
int numero = minimo; // El número de inicio

for (int i=1; i<=maximo-minimo+1; i++) {   // Repetimos las instrucciones 75 veces
  suma = suma + numero;
  numero = numero + 1; // Pasamos al siguiente número
}
print("La suma es: "); println(suma);   // Mostramos el resultado
```

3.4. Instrucciones `continue` y `break`

Hay dos instrucciones que aumentan el control sobre los bucles. Sirven para parar y continuar con la siguiente iteración del bucle respectivamente.

- La instrucción `break` se usa para interrumpir (romper) la ejecución normal de un bucle allá donde se encuentre. También se usa en el `case` de un `switch` para omitir el resto del `switch` allá donde se encuentre.
- La instrucción `continue` se usa para pasar a la siguiente iteración.

  La instrucción `continue` solo aparece en las estructuras repetitivas y su función es evitar que las instrucciones restantes del cuerpo del bucle para proceder a la siguiente iteración del mismo.

Sketch 17: break y continue en una estructura while

```
int i=0;

while (i < 15) {
  if (i > 2) {
    println("'i' es más grande que 2.");
    break; // Rompe el bucle
  }
  else {
    print("i = "); println(i);
    i = i + 1;
    continue; // Empieza la siguiente iteración del bucle
    //print(" Esto nunca se imprimirá.");
  }
}
```

4. Programación modular

La **declaración de la función** (o cabecera de la función) está formada por la siguiente porción de código:

```
1        tipo_de_dato_retornado    nombreDeLaFuncion  (parámetros)
```

donde se especifica el tipo de dato que devuelve la función, el nombre de la función y sus parámetros de entrada.

- Una función puede devolver un valor que responda a un tipo de dato primitivo (`char`, `boolean`, `int`, `float`, ...) como cualquier estructura de datos (como veremos en el tema siguiente). También se puede indicar el **tipo de dato vacío o nulo** que se especifica con la palabra reservada `void` para hacer referencia al hecho de que la función no devolverá absolutamente nada.
- La siguiente componente de la declaración, el **nombre de la función** `nombreDeLaFuncion`.
- La ultima parte de la declaración consta de los **parámetros**. Representan la información de entrada con la que trabajará la función.
  - Si no utilizara ningún dato de entrada no se debe especificar nada entre los paréntesis.
  - Si utilizaran parámetros de entrada, cada item informativo se indicará mediante una secuencia separada por comas. Cada elemento de la secuencia consta de dos elementos. El segundo es el nombre del identificador del parámetro y el segundo es el tipo de dato de dicho parámetro:

```
1 tipo_de_dato identificadorVar1, tipo_de_dato identificadorVar2, ...
```

El término **parámetro** (a veces llamado parámetro formal) se utiliza para referirse a la variable que se encuentra en la declaración de la función, mientras que que el **argumento** (a veces llamado parámetro real) se refiere a la entrada real que se ha pasado en la invocación.

4.1. Tipos de funciones

En función de los datos recibidos "del exterior" y el resultado devuelto "del interior", se distinguen 4 tipos de funciones:

1. **Funciones que no devuelven valor.** Independientemente del número de argumentos, ejecutan una acción y nunca devuelven el valor de algún resultado, lo que se reconoce al indicar el tipo de dato vacío.

Sketch 18: Funciones que no devuelven valor

```
1 void nombreDeLaFuncion (----) {
2   // Bloque de código de la función.
3 }
```

2. **Funciones que sí devuelven un valor.** Son aquellas que tras su ejecución generan un valor como resultado del proceso y lo "entregan" a su retorno, lo que se reconoce con el tipo de dato que aparece al inicio de la declaración y la instrucción `return` en el cuerpo de la función.

Sketch 19: Funciones que sí devuelven valor

```
1 tipoDato nombreDeLaFuncion (----) {
2     tipoDato variable ;
3     // Resto del bloque de código de la función.
4     return variable ;
5 }
```

3. **Funciones sin paso de parámetros.** Son subprogramas que no requieren información adicional de su entorno. Ejecutan la acción para la que fueron diseñadas cada vez que son invocadas pudiendo o no retornar algún valor.

Sketch 20: Funciones sin paso de parámetros

```
---- nombreDeLaFuncion () {  // <- no hay nada entre los paréntesis.
  // Bloque de código de la función.
}
```

4. **Funciones con paso de parámetros.** Son subprogramas que sí requieren información adicional de su entorno y necesitará tantos items informativos como variables hayan sido indicadas entre los paréntesis '(' y ')'. Ejecutan instrucciones en las que (necesariamente) manipulan los datos de entrada.

Sketch 21: Funciones con paso de parámetros

```
---- nombreDeLaFuncion (tipo1 var1, tipo2 var2, tipo3 var3, ...) {
  // Bloque de código de la función.
}
```

En este tipo de funciones es muy importante recordar que:

- La invocación a la función se tendrá que realizar con tantos valores como parámetros se hayan indicado en la declaración y que cada valor deberá ser del mismo tipo que el del parámetro.
- Las variables que aparecen como parámetros se consideran variables declaradas en el bloque del código de la función, actuando como variables locales en el bloque y por tanto no son variables reconocidas fuera del mismo.
- Cuando se invoca a la función, los valores dados en los argumentos corresponden a la inicialización de los parámetros (inicialización de variables locales) en el orden en el que se suministren: el primer valor inicializa el primer parámetro, el segundo valor al segundo parámetros, y así sucesivamente.

Sketch 22: Un ejemplo de Processing

```
void setup() {
  // random() es una función que retorna un dato en función de los argumentos.
  int num = random(0,11);

  tablaDeMultiplicar(num); // Muestra la tabla de num
  todasLasTablas(); // Muestra todas las tablas.
}

void todasLasTablas() { // No retorna. Sin parámetros.
  for (int numero=1; numero<=10; numero++) {
    print("Tabla de multiplicar del "); println(numero);
    tablaDeMultiplicar(numero);
  }
}

void tablaDeMultiplicar(int n) { // No retorna. Con parámetros.
  for (int i=1; i<=10; i++) {
    print(n); print(" x "); print(i); print(" = "); println(n*i);
  }
}
```

4.2. Paso por referencia y paso por valor

El **paso de argumentos por valor** consiste en copiar el contenido de la variable-argumento que queremos pasar en otra dentro del ámbito local de la función. En concreto, consiste en copiar el contenido de la memoria del argumento a la dirección de memoria correspondiente al parámetro dentro del ámbito de dicha función. Se tendrán entonces dos valores duplicados e independientes, con lo que la modificación de uno no afecta al otro.

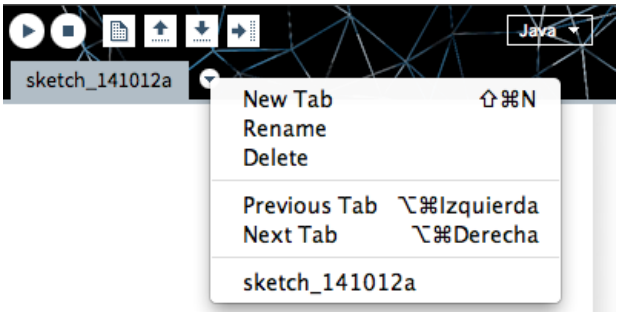
El **paso de argumentos por referencia** consiste en proporcionar a la función a la que se le quiere pasar el dato no tanto el dato en sí, sino la dirección de memoria donde está almacenado el dato. En este caso se tiene un único valor referenciado (o apuntado) desde dos puntos diferentes, el programa principal y la función a la que se le pasa el argumento, por lo que cualquier acción sobre el parámetro se realiza sobre la misma posición de memoria afectará a la variable argumento.

> *"Existe un solo modo de paso de parámetros en Java – paso por valor – y eso ayuda a mantener las cosas simples."*
>
> *"The Java Programming Language" de Ken Arnold y James Gosling*

4.3. Modularidad en Processing

Processing permite la creación de módulos de una forma francamente sencilla. Debes seguir los siguientes pasos:

1. Haz el programa.
2. Guarda tu programa.
3. Pulsa el triángulo que se encuentra a la derecha de la pestaña que contiene el nombre del fichero.
4. Selecciona "New Tab" e introduce el nombre del nuevo fichero o módulo, que se guardará en el mismo directorio donde se encuentre tu programa principal.
5. Refactoriza el programa creando las funciones adecuadas al módulo.



Como ejemplo, el siguiente programa se puede descomponer en dos módulos.

Sketch 23: Dibujar 100 bolas. Un solo programa con funciones.

```
1  // Función principal
2  void setup() {
3    size(200, 200);
4    background(255);
5    for (int i=0; i<100; i++) // Lanzamos 100 bolas
6      lanzaBola();
7  }

9  // Función que se encarga de dibujar la bola en una posición aleatoria.
10 void lanzaBola() {
11   int x = (int) random(width);
12   int y = (int) random(height);
13   dibujaBola(x, y);
14 }

16 // Función dibuja una bola en la posición dada.
```

```
17 void dibujaBola(int x, int y) {
18   int radio = (int) random(0.2*min(width, height));
19   fill(random(255), random(255), random(255));
20   ellipse(x, y, radio, radio);
21 }
```

En este caso creamos dos ficheros, uno es el programa en sí y el otro es un módulo formado por estas dos funciones.

Sketch 24: Programa que dibuja 100 bolas. Usa el módulo "bolas".

```
1 // Función principal
2 void setup() {
3   size(200, 200);
4   background(255);
5   for (int i=0; i<100; i++) // Lanzamos 100 bolas
6     lanzaBola();
7 }
```

Sketch 25: Módulo "bolas".

```
1 // Función que se encarga de dibujar la bola en una posición aleatoria.
2 void lanzaBola() {
3   int x = (int) random(width);
4   int y = (int) random(height);
5   dibujaBola(x, y);
6 }

8 // Función dibuja una bola en la posición dada.
9 void dibujaBola(int x, int y) {
10   int radio = (int) random(0.2*min(width, height));
11   fill(random(255), random(255), random(255));
12   ellipse(x, y, radio, radio);
13 }
```

5. Estructuras de datos

Una **estructura de datos** es una agrupación de datos que se trata como una unidad. Puede contener solo datos elementales o bien datos estructurados o bien datos elementales y estructurado.

5.1. String

El tipo de dato `String` o cadena de caracteres almacena una secuencia de caracteres `char` que se interpretan como un dato único. El dato que almacena un `String` es una cadena alfanumérica.

La declaración de un tipo de dato String se realiza mediante la instrucción.

```
String identificador_variable;
```

Las **funciones más importantes** son:

- `str = valor`: asignación de `valor` a `str`. La asignación de un literal es `str="texto";`.
- `int strLength(String str)`, obtener la longitud de `str`.
- `char strCharAt(String str, int pos)`, obtener el carácter de la cadena `str` que se encuentra en la posición `pos`.
- `String strConcat(String str1, String str2)`, retornar la concatenación de cadenas en el orden: primero `str1` y después `str2`.
- `int strComp(String str1, String str2)`, compara la cadena `str1` con la cadena `str2`.
- `int strChar(char c, String str, boolean principio)`, retorna el lugar que ocupa `c` en la cadena `str` ya sea desde el principio (para `principio` con valor `true`) o desde el final (para `principio` con valor `false`) .
- `String strSubString(String str, int pos1, int pos2)`, retorna la subcadena de `str` que se encuentra entre las posiciones `pos1` y `pos2` (inclusiven).
- `String strCharToStr(char c)`, retorna un String formado solo por el carácter `c`. En Processing requiere del tipo de dato `array` para su definición.
- `String strReplaceChar(int pos, char c, String str)`, retorna un String igual que `str` pero el símbolo de la posición `pos` se sustituye por el del carácter `c`.

Tal y como están aquí declaradas, todas estas funciones hay que implementarlas menos la asignación. Cada lenguaje declara y define sus propias funciones. Processing proporciona estas, entre otras:

Objetivo	**Función de String** (teórica)	**Función de String en Processing**	**Ejemplo en Processing**
Asignación	=	=	`str= "cadena"`
Longitud	strLength()	length()	`str.length()`
Char en posición	strCharAt()	charAt()	`str.charAt(pos)`
Concatenar	strConcat()	+	`str1+str2`

5.2. Array

Un **array** es un conjunto de datos del mismo tipo almacenados en la memoria del ordenador en posiciones adyacentes que se agrupan bajo un nombre común y se pueden tratar como una unidad. Desde un punto de vista lógico un array se puede contemplar como un conjunto de elementos ordenados en una fila (o columna) de forma similar a un String (pero sin el byte extra).

dato0	dato1	dato2	···	datoN

dato0
dato1
dato2
⋮
datoN

La declaración de un tipo de dato array se realiza mediante la instrucción.

```
tipo_de_dato[ ] identificador_variable;
```

En **Processing** la construcción de un array de `n` datos se consigue con:

```
identificador_variable = new tipo_de_dato[ n ];
```

- `new` es la instrucción que hace la reserva,
- `n` es el número que indica el número máximo de datos que se almacenarán.

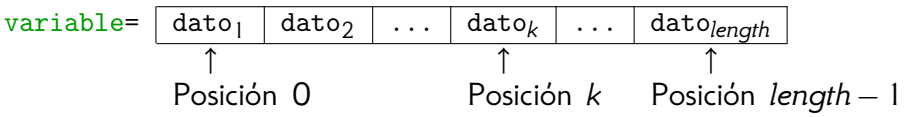
Es bastante usual que **la declaración y reserva de memoria** se hagan simultáneamente:

```
tipo_de_dato[ ] identificador_variable = new tipo_de_dato[ n ];
```

En **Processing** también podemos recuperar el tamaño de una array con la expresión `variable.length`. Hay que distinguir muy bien entre posición y elemento de la posición:

- La **posición** (o índice) es el lugar que ocupa cada "caja" (o celda) en el conjunto de array.
- El **elemento** es el dato que hay almacenado en dicha posición (o caja).

En cada posición se almacena un dato. La sintaxis para acceder a uno de tales posiciones/elementos es `variable[k]` donde `variable` es el identificador de la variable de tipo array y `k` es la posición. Los valores posibles de tales índices se llama **rango** y para un array de tamaño `n` toma valores entre `0` y `n-1` (o si lo prefeieres, entre `0` y `variable.length-1`).



Recuerda:

- `int lista` representa que la variable lista es un entero, y
- `int[] lista` representa que la variable lista es un array de entero,
  parece que lo natural es declarar
- `int[][] lista` a un array de array de enteros.
- `int[][][] lista` a un array de array de array de enteros.
  Y si quisieramos generalizarlo,
- `int[] ... [][][] lista` a un array de array de array .... de array de enteros.

Sketch 26: Ejemplo de Inicialización de un Array de Array de Array de enteros

```
1 int[][][] var = new int[4][2][3];
2
3 for (int i=0; i<4; i++)                                  // int[][]'[]'
4   for (int j=0; j<2; j++)                                // int[]'[]'[]
5     for (int k=0; k<3; k++)                              // int'[]'[][]
6       var[i][j][k] = Dialog.readInt("Dame el dato"); // 'int'[][][]
```

Si una función tiene como parámetro una array n-dimensional y en el bloque de la función se modifican los elementos del array, entonces el argumento también cambiará.

5.3. Registros

Los **registros** son estructuras de datos que almacenan datos de distinto tipo identificados con un nombre y se pueden trabajar con ellos como si fueran una unidad (un dato). El registro puede contener variables de tipo simple o de tipo eststructurado. Cada uno de los datos se denomina **campo**, y cada una de las variables que contiene el registro se llama **nombre del campo**.
En **Processing** es:

```
class NombreDelRegistro {
  tipo_de_dato1 variable1;
  tipo_de_dato2 variable2;
  ....
}
```

Una vez definido un registro, ya tienes un nuevo tipo de dato en el lenguaje de programación y como tal lo puedes incluir en cualquier estructura que consideres, ya sea en un nuevo tipo de registro o en un array. Los procesos de declaración, inicialización y acceso a los datos deberá ser acorde a todo lo expresado anteriormente.

En **Processing** la construcción de un registro se realiza con:

```
identificador_variable = new Tipo_de_registro;
```

- `new` es la instrucción que hace la reserva,
- `Tipo_de_registro` es el nombre de un registro previamente definido.

A continuación realiza los procesos de construcción, asignación y modificación que correspondan a cada campo del registro de acuerdo al tipo de dato que hayas declarado para él.

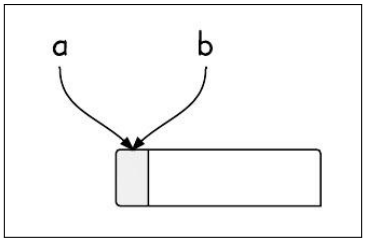
Sketch 27: Definición de un registro en Processing y su inicialización

```
class  Persona {
  String nombre;
  int edad;
  String[] hermanos;
}

// Reservamos espacio para el registro y para cada uno de los campos que también lo requieran.
Persona registro = new Persona();
registro.hermanos = new String[2];

// Asignamos valores a los campos
registro.nombre = "mi nombre personal";
registro.edad = 27;
registro.hermanos[0] = "Mi primer hermano";
registro.hermanos[1] = "Mi segundo hermano";
```

5.4. Gestión de Memoria

Con este breve apartado queremos hacer mucho hincapié en la gestión de memoria con los tipos de datos estructurados, que almacenan punteros o referencias.

Si tienes dos variables `A` y `B` del mismo tipo y una de ellas, digamos `A`, ya tienen guardada una referencia a un zona de memoria que contiene información, la asignación `B=A` almacena en `B` la referencia que contiene `A`. El resultado de la asignación es que las dos variables apuntan a la misma zona de memoria.



Si tienes dos variables que apuntan a la misma dirección de memoria, modificar a través de **una** de las variables uno de sus elementos informativos, modificará simultáneamente la información de las **dos variables**. Este es el motivo por el que si tienes como parámetro un dato estructuado, los cambios que realices en él en el cuerpo de una función serán "visibles" fuera de la función.

En consecuencia, copiar los datos a los que apuntan dos variables del mismo tipo (array o registro) `A` en `B` no se resuelve con la asignación `B=A`. La asignación, solo hace que las dos variables tengan la misma referencia a la información (no que contengan la información en sí). Si se parte de `A`, copiar su información en `B` conlleva:

1. reservar el espacio de memoria adecuado para `B` (conocido el espacio que está ocupando `A`), y
2. copiar el contenido de cada uno de los items informativos de `A` en `B`.

Ten en cuenta que si alguno de esos items informativos fuese a su vez otro tipo de dato estructurado tendrás que repetir el proceso indicado en los dos pasos anteriores.

El siguiente código crea un vector a partir de un literal. A continuación se realiza la copia de dicho valor en un **nuevo lugar de la memoria** para evitar que las dos variables apunten a la misma referencia.

Sketch 28: Copia de vectores

```
1 // Colección de datos original
2 int[] vectorA = {1 , 11, 111};

4 // Reservamos la misma cantidad de memoria que número de datos tenga la colección de datos original.
5 int[] vectorB = new int[vectorA.length];

7 // Copiamos los datos
8 for (int indice=0; indice < vectorA.length; indice++)
9    vectorB[indice] = vectorA[indice];
```

6. Algoritmos de ordenación y búsqueda

6.1. Algoritmos de búsqueda

Búsqueda secuencial

Datos de entrada:

- `vec`: vector en el que se desea buscar el dato.
- `dato`: elemento que se quiere buscar.

Variables intermedias:

- `tam`: tamaño del vector. Los subíndices válidos van desde 0 hasta tam-1 inclusive.
- `pos`: posición actual en el vector.

```
1  int busquedaSecuencial(vec, dato)
2    tam = length(vec)
3    pos = 0
4
5    Mientras pos < tam:
6       Si vec[pos] == dato return pos,
7       de lo contrario, pos = pos + 1
8    Fin (Mientras)
9
10   return -1
```

Observa que se asume que `tam` viene implícito por el vector `vec`. En Processing se determina por `vec.length`

Búsqueda Binaria

Datos de entrada:

- `vec`: vector ORDENADO en el que se desea buscar el dato.
- `dato`: elemento que se quiere buscar.

Variables intermedias:

- `tam`: tamaño del vector. Los subíndices válidos van desde 0 hasta tam-1 inclusive.
- `pos`: posición actual en el vector.
- `inf`: límite inferior del intervalo.
- `sup`: límite superior del intervalo.

```
1  int busquedaBinaria(vec, dato)
2    tam = length(vec)
3    inf = 0
4    sup = tam-1
5
6    Mientras inf <= sup:
7      pos = ((sup - inf) / 2) + inf // División entera: se trunca la fracción
8      Si vec[pos] == dato return pos,
```

```
    de lo contrario:
      Si dato < vec[pos] entonces sup = pos - 1
      en caso contrario inf = pos + 1
  Fin (Mientras)
  return -1
```

Observa que se asume que `tam` viene implícito por el vector `vec`. En Processing se determina por `vec.length`.

6.2. Algoritmos de ordenación

Ordenación de Burbuja

Datos de entrada:

- `vec`: vector que se desea ordenar.

Variables intermedias:

- `swapped`: variable que indica si se he producido un intercambio de valores.
- `tam`: tamaño del vector. Los subíndices válidos van desde 0 hasta tam-1 inclusive.

Funciones:

- `intercambiar(a, b)`: intercambia los valores de las variables a y b.

```
ordenaciónBurbuja(vec)
  tam = length(vec)

  hacer
    swapeed = false
    Desde pos=1 hasta tam-1 inclusive hacer
      Si vec[pos-1] > vec[pos] entonces
        intercambiar(vec[pos-1], vec[pos])
        swapped = true
      Fin (Si)
    Fin (For)
 mientras swapped
```

Se puede optimizar si se tiene en cuenta que en cada iteración se puede haber colocado más de un elemento en su posición final.

Datos de entrada:

- `vec`: vector que se desea ordenar.

Variables intermedias (se cambia por `swapeed`):

- `nuevoTam`: variable que almacena el último elemento intercambiado.
- `tam`: tamaño del vector. Los subíndices válidos van desde 0 hasta tam-1 inclusive.

Funciones:

- `intercambiar(a, b)`: intercambia los valores de las variables a y b.

```
ordenaciónBurbuja(vec)
  tam = length(vec)

  hacer
    nuevoTam = 0
    Desde pos=1 hasta tam-1 inclusive hacer
      Si vec[pos-1] > vec[pos] entonces
        intercambiar(vec[pos-1], vec[pos])
        nuevoTam = pos   <<< Optimización
      Fin (Si)
    Fin (For)
    tam = nuevoTam   <<< Optimización
 mientras tam != 0   <<< Optimización
```

Datos de entrada:

- `vec`: vector que se desea ordenar.

Variables intermedias:

- `tam`: tamaño del vector. Los subíndices válidos van desde 0 hasta tam-1 inclusive.
- `pos`: posición del elemento a comparar.
- `siguiente`: variable que recorre todos las posiciones siguientes a pos (hasta la última)
- `minimo`: posición donde se encuentra el elemento más pequeño encontrado.

Funciones:

- `intercambiar(a, b)`: intercambia los valores de las variables a y b.

Ordenación por selección

```
ordenaciónSelección(vec)
  tam = length(vec)

  Desde pos=0 hasta tam-1 inclusive hacer

    minimo = pos;
    Desde siguiente=pos+1 hasta tam-1 inclusive hacer
        Si vec[minimo] > vec[siguiente] entonces
            minimo = siguiente;
        Fin (Si)
    Fin (For-siguiente)

    intercambiar(vec[pos], vec[minimo])
  Fin (For-pos)
```

Datos de entrada:

- `vec`: vector que se desea ordenar.
- `tam`: tamaño del vector. Los subíndices válidos van desde 0 hasta tam-1 inclusive.

Variables intermedias:

- `pos`: posición del elemento a comparar.
- `anterior`: variable que recorre todos las posiciones siguientes a pos (hasta la última)

Funciones:

- `intercambiar(a, b)`: intercambia los valores de las variables a y b.

Ordenación por inserción

```
1  ordenaciónInserción(vec)
2    tam = length(vec)
3
4    Desde pos=1 hasta tam-1 inclusive hacer
5
6      valor = vec[pos]
7      anterior = pos - 1
8
9      Mientras anterior >= 0  AND vec[anterior] > valor hacer
10         intercambiar(vec[anterior+1], vec[anterior])
11         anterior = anterior - 1
12     Fin (Mientras)
13
14     vec[anterior + 1] = valor
15
16   Fin (For-pos)
```

7. Recursividad

Toda definición recursiva establece un proceso que debe venir dado por 3 reglas.

- **Regla Base:** es la que indica cuáles son los ejemplos o casos particulares que cumplen la definición. Es imprescindible para hacer una definición recursiva.
- **Regla Recursiva:** realmente es un conjunto de reglas que aplicándolas establece cuándo un objeto responde a la definición, reglas en las que deben de aparecer de nuevo el concepto a definir. . Es una parte imprescindible.

  También recibe el nombre de pasos recursivos.
- **Regla de Exclusión:** es un conjunto de reglas que indica cuándo un objeto no se ajusta al concepto en términos de la regla base y la regla recursiva. Normalmente esta parte va implícita y es opcional.

Todo concepto recursivo se construye a partir de unas reglas para las que **se requiere de una clara ordenación** que permita el crecimiento gradual de objetos en el dominio que estamos definiendo. Para dicho orden y crecimiento es fundamental disponer de algún procedimiento que garantice cómo se construye un objeto nuevo a partir de uno o varios de los ya construido. Si el procedimiento construye siempre objetos nuevos y diferentes de los ya construidos tendrás una buena definición recurrente; si no es así tendrás un conflicto en la definición. Los procedimientos de construcción de objetos es lo que formalmente se conoce como Recursividad Estructural, que son aquellos en los que partiendo de una serie de objetos $\{y_1, y_2, \ldots, y_n\}$ se construye un nuevo objeto x.

La programación como técnica para la resolución de problemas también contempla la recursividad. Una primera definición al concepto de recursividad desde la perspectiva de la programación es: "una función es recursiva si en su cuerpo se invoca a sí misma". Al igual que en el planteamiento teórico, no se puede entrar en un ciclo sin fin. Solo se puede construir una función recursiva si sus acciones responden a una recursividad estructural (porque hay una definición recursiva que las justifica) y, por tanto, toda función recursiva deberá contemplar una regla base, una regla recursiva y, si fuese necesario, una regla de exclusión.

La recursividad se puede presentar de muy distintos modos. Existen la recursión indirecta o cruzada, directa, simple o lineal, lineal final, múltiple y anidada - ver los apuntes para los detalles. También está la **Recursión infinita** que es la que hay que evitar a toda costa: algorítmicamente hablando, una recursión infinita suele ser equivalente a un programa mal hecho donde la regla base no ha sido correctamente establecida o donde la regla recursiva no establece correctamente el orden de creación de objetos.

La recursion es una alternativa a la iteración en la solución de problemas, especialmente si estos tienen naturaleza recursiva. En estos casos, el esquema suele ser el de una Recursividad Lineal:

```
1  tipo_dato  funcionRecursiva  (parametros) {
2    if (parametros no posibles) return error;  //  Regla de exclusión.
3
4    if (parámetros son caso base)  { // Regla base.
5      Calcular solución inmediata.
6      return solución
7    }
8    else {  // Regla recursiva.
9       solucion parcial =  funcionRecursiva ( parametros para problema más pequeño )
10      Calcular solución a partir de la solución parcial.
11      return solución
12   }
13 }
```

7.1. Ejemplo: Factorial de un número

Como ejemplo tenemos el producto de los primeros n números naturales.

Sketch 29: Implementación del Factorial en Processing

```
1  void setup() {
2    println(factorial(inputInt("Dame un número natural")));
3  }
4
5  int factorial (int n) {
6    if (n<1) return -1;  // Regla de exclusión. Un -1 es un error. No existe el factorial.
7
8    if (n == 1) return 1; // Regla base. El producto del primer número natural.
9    else return n *  factorial(n-1);  // Regla recursiva. El producto de n-números naturales.
10 }
```

Para la invocación `factorial(5)` se tienen las siguientes llamadas.

`factorial(5)`

Para poder retornar el valor se requiere calcular antes:

`factorial(4)`

Para poder retornar el valor se requiere calcular antes:

`factorial(3)`

Para poder retornar el valor se requiere calcular antes:

`factorial(2)`

Para poder retornar el valor se requiere calcular antes:

`factorial(1)`

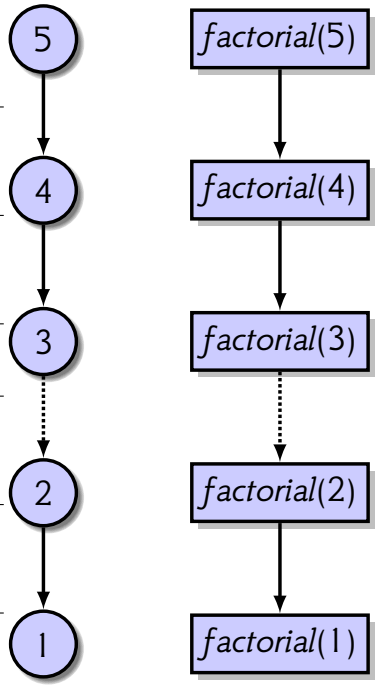
Corresponde al caso base. Su valor es conocido.
`return 1`

Conocido el valor de factorial(1), se puede calcular el producto.
`return 2 * 1`

Conocido el valor de factorial(2), se puede calcular el producto.
`return 3 * 2`

Conocido el valor de factorial(3), se puede calcular el producto.
`return 4 * 3`

Conocido el valor de factorial(4), se puede calcular el producto.
`return 5 * 4`



El árbol de la derecha se llama **árbol de llamadas**. La figura del centro es el árbol de llamadas sin indicar el nombre d la función (por ser lineal).

8. Resumen de algunas funciones y variables de Processing

8.1. Entorno

- `size(w,h)`, tamaño de la ventana gráfica.
- `weight`, variable que almacena la resolución del ancho de la ventana gráfica.
- `height`, variable que almacena la resolución del alto de la ventana gráfica.
- `displayWeight`, variable que almacena la resolución del ancho del monitor.
- `displayHeight`, variable que almacena la resolución del alto del monitor.

8.2. Consola de texto

- `print(expresión)`, imprime una cadena alfanumérica o el resultado de una expresión matemática.
- `println(expresión)`, imprime una cadena alfanumérica o el resultado de una expresión matemática y a continuación imprime un retorno de carro (crea una línea nueva).

8.3. Figuras geométricas

- `point(x,y)`, dibuja un punto.
- `line(x1, y1, x2, y2)`, dibuja un línea.
- `triangle(x1, y1, x2, y2, x3, y3)`, dibuja un triángulo dados sus vértices.
- `rect(a, b, c, d)`, dibuja un rectángulo dado su vértice superior izquierdo, el largo y el ancho.
- `quad(x1, y1, x2, y2, x3, y3, x4, y4)`, dibuja un cuadrilátero dados sus vértices.
- `ellipse(a, b, c, d)`, dibuja una elipse inscrita en un rectángulo dado su centro, largo y ancho.
- `arc(a, b, c, d, inicio, final)`, dibuja un arco.
- `text(expresión, x, y)`, muestra una expresión (igual que en `print()`) en el punto (*x*,*y*) de la ventana gráfica.

8.4. "Efectos" en la pantalla gráfica

- `background(r,g,b)`, establece el color del fondo de la ventana gráfica.
- `fill (r, g, b, alpha)`, establece el color del fondo de las siguientes figuras geométricas.
- `noFill()`, deja de colorear el fondo de las siguientes figuras geométricas.
- `stroke(r, g, b, alpha)`, establece el color del borde de las siguientes figuras geométricas.
- `strokeWeight(grosor)`, establece el grosor del borde de las siguientes figuras geométricas.
- `noStroke()`, elimina el borde de las siguientes figuras geométricas.

Cuidado: Aplicar `noFill()` y `noStroke()` no dibujará nada.

8.5. Animación

- `draw()` es la función que se repite, por defecto, 60 veces por segundo.
- `noLoop()` inhabilita la ejecución de `draw`. Deja de ejecutarse el ciclo de animación.
- `loop()` habilita la ejecución de `draw`. Ejecuta el ciclo de animación.
- `redraw()` ejecutará el código de `draw()` donde se invoque, que debe ser dentro de funciones de eventos de ratón o de teclado o de aquellas funciones del programador que estén en ejecución.
- `frameRate(fps)` indica el número de frames que se mostrarán por segundo.

8.6. Funciones y variables relacionadas con el teclado

- `keyPressed()` es un función que se llamará cada vez que se presione un tecla. La tecla presionada se guardará en la variable `key`.
- `keyReleased()` es un función que se llamará cada vez que se suelte un tecla. La tecla presionada se guardará en la variable `key`.
- Las variables que informan sobre el teclados son:
  - `key` es un variable global de tipo `char` que almacena la tecla presionada. Para teclas no-ASCII (cuando `key == CODED`) hay que usar la variable `keyCode`. `key` reconoce las teclas BACKSPACE, TAB, ENTER, RETURN, ESC, y DELETE
  - `keyCode` almacena el valor de teclas especiales como UP, DOWN, LEFT, RIGHT, teclas de flechas, y ALT, CONTROL, SHIFT.
  - `keyPressed` es una variable booleana que vale `true` cuando se presiona una tecla y es `false` si no se presiona ninguna tecla.

8.7. Funciones y variables relacionadas con el ratón

- `mouseMoved()` se llamará cada vez que se mueva el ratón pero sin ningún botón del ratón presionado.
- `mousePressed()` es un función que se llamará cada vez que se presione un botón del ratón.
- `mouseReleased()` se llamará cada vez que se suelte un botón del ratón.
- `mouseClicked()` es un función que se llamará cada vez que se presione y suelte el botón de ratón.
- `mouseDragged()` es un función que se llamará cada vez que se mueva el ratón con un botón del ratón presionado.
- Las variables que informan sobre el ratón son:
  - `mouseButton` es un variable global que guarda el botón que ha sido presionado del ratón. Sus valores son: LEFT, RIGHT, o CENTER.
  - `mousePressed` variable booleana que vale `true` si un botón está presionado o `false` si ninguno lo está.
  - `mouseX` indica la coordenada horizontal actual del ratón.
  - `mouseY` indica la coordenada vertical actual del ratón.
  - `pmouseX` indica la coordenada horizontal del ratón en el frame previo al frame actual.
  - `pmouseY` indica la coordenada vertical del ratón en el frame previo al frame actual.