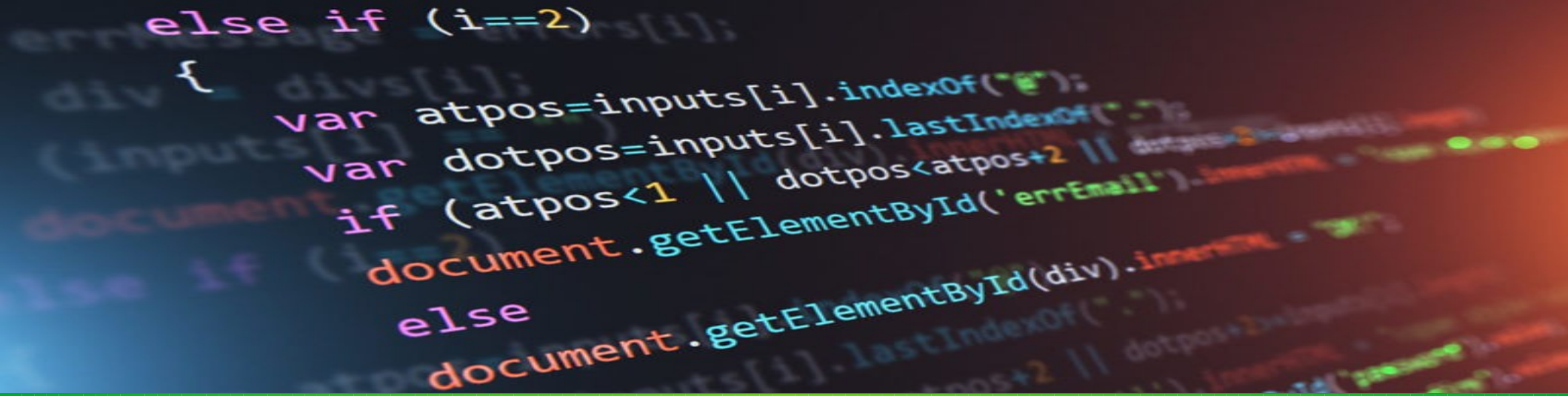


PROGRAMACIÓ

UNITAT 4: MÈTODES I FUNCIONS

CONTINGUTS

1. CONCEPTES BÀSICS

2. ÀMBIT DE LES VARIABLES

3. PAS D'INFORMACIÓ A UNA FUNCIÓ

4. VALOR RETORNAT PER UNA FUNCIÓ

5. SOBRECÀRREGA DE FUNCIONS

6. RECURSIVITAT

CONCEPTES BÀSICS

Conforme augmenta l'extensió i la complexitat d'un programa, és genera dos problemes: **duplicitat de codi** i **dificultat en el manteniment**.

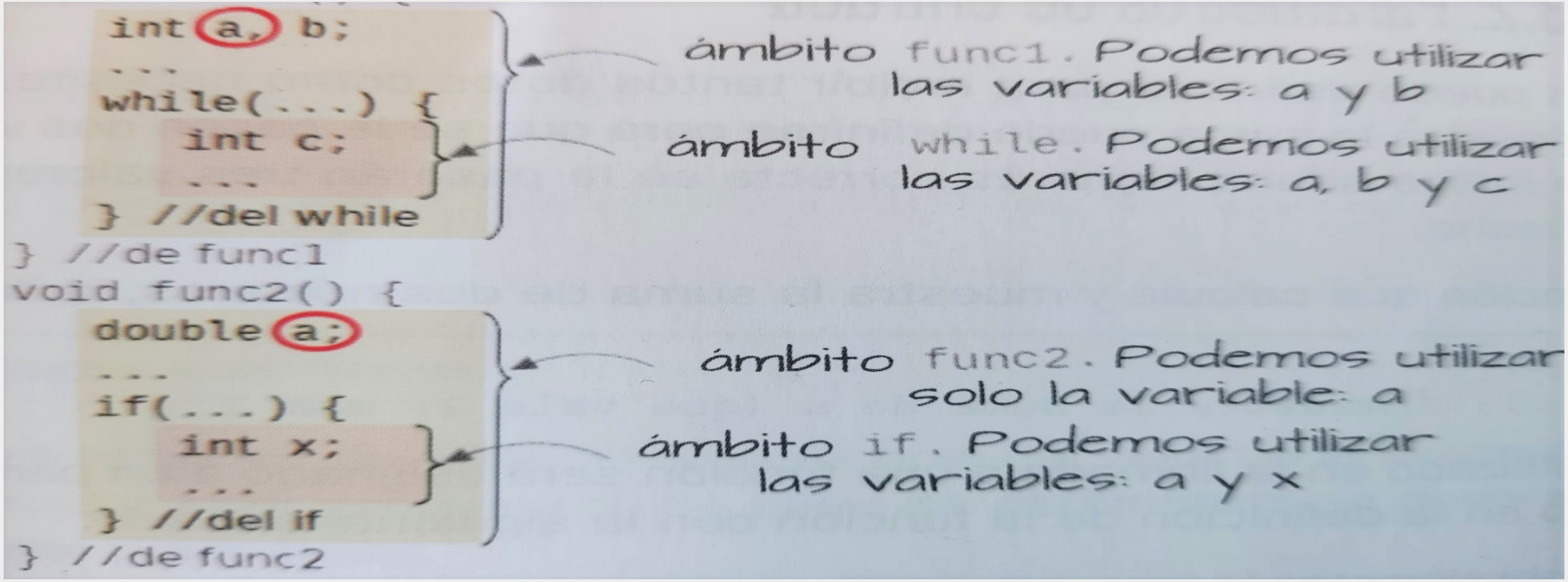
La solució per a quan necessitem les mateixa funcionalitat en diferents llocs del nostre codi no és més que etiquetar amb un nom un fragment de codi i substituir en el programa aquest fragment, en tots els llocs on aparega, pel nom que li hem assignat. Per exemple:

```
public static void main(String[] args) {
    ... //código
    sumaDosNumeros(a, b);
    ... //código
    sumaDosNumeros(c, d);
    ... //código
    sumaDosNumeros(e, f);
}
```

- **Concepte de funció:** Conjunt d'instruccions agrupades sota un nom comú, que s'executa en ser invocada.
- La definició d'una funció pot fer-se abans o després del **main()**.
- **Cridà a la funció**: És el nom de la funció, seguit de () -parèntesi-. Es converteix en una nova instrucció que podem utilitzar per a invocar-la. Comportament d'una cridà a una funció:
    1. Les instruccions del programa principal s'executen fins que troben la cridà a la funció.
    2. L'execució salta a la definició de la funció.
    3. S'executa el cos de la funció.
    4. Quan l'execució del cos acaba, retornem al punt del programa des d'on es va invocar la funció.
    5. El programa continua la seua execució.
- Una funció i un mètode són conceptualment idèntics, l'única diferència està en el nom, en la programació estructurada es diuen **funcions**, i en la programació orientada a objectes, es denominen **mètodes**.

ÀMBIT DE LES VARIABLES

- En el següent codi es defineixen dues funcions func1() i func2() i es representa gràficament l'àmbit de les diferents variables declarades.



Encara que les variables **a** de func1() i de func2() (marcades en roig) compartisquen identificador, són variables diferents.

PAS D'INFORMACIÓ A UNA FUNCIÓ

- A vegades, una funció necessita conéixer informació externa per a poder dur a terme la seua tasca. Vegem un exemple: Si desitgem implementar una funció que salude un nombre indeterminat de vegades, podem implementar la funció **variosSaludos()** a la qual se li passa el nombre de vegades que volem saludar. D'aquesta manera, si executem **variosSaludos(7)**, saludarà set vegades i si executem **variosSaludos(2)**, ho farà en dues ocasions.

```
static void variosSaludos(int vegades) {
    for(int i = 0; i < veces; i++) {
        System.out.println("Hola.");
    }
}
```

La variable **vegades** és un paràmetre d'entrada de la funció variosSaludos(). Un paràmetre d'entrada d'una funció no és més que una variable local a la qual se li assigna valors en cada crida.

- En la **crida a una funció** es poden passar valors que provenen de literals, expressions o variables. Per exemple, a la funció variosSaludos() se li passa un enter (nombre de vegades que desitgem saludar).

```
variosSaludos(2); // crida amb un literal
int n = 3;
variosSaludos(2*n) // crida amb una expressió
```

- Una funció pot definir-se per a rebre tantes dades com necessite. Per a una funció que calcula i mostra la **suma** de dos números, se li passen dos valors que sumar, la crida seria:

```
int a = 3;
suma(a, 2);  //mostra la suma de a (que val 3) més 2
```

i a una altra que indica si la **data** és correcta se li passaran tres valors: l'any, el mes i el dia de la data.

- El nombre de paràmetres definits en la funció determina el nombre de valors que cal utilitzar en cada crida i el seu tipus.

PAS D'INFORMACIÓ A UNA FUNCIÓ

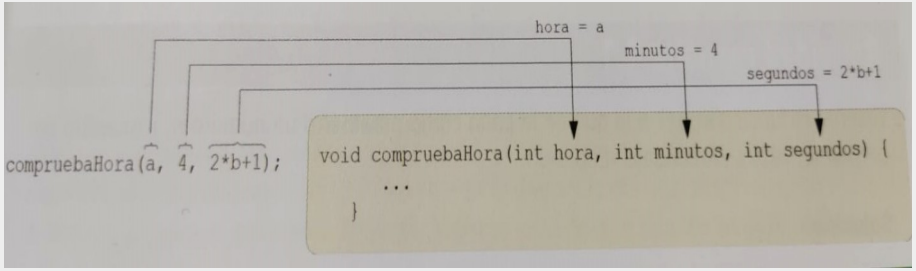
- Suposant que haguérem implementat la funció **comprobaHora()**, a la qual se li passa l'hora, minuts i segons d'un instant, i mostra en pantalla si l'hora és correcta i incorrecta, el prototip de la funció seria:

  static void compobaHora(int hora, int minuts, int segons)

- Un exemple de la crida a la funció es:

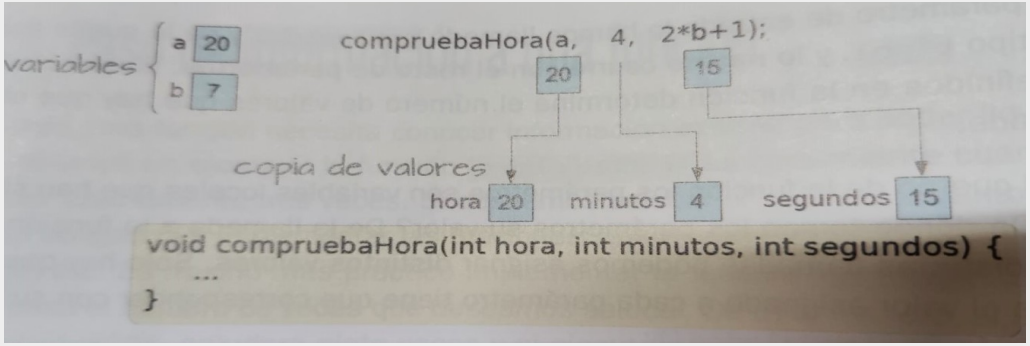
  comprobaHora(a, 4, 2*b+1);

- El mecanisme de pas de paràmetre es representa en la següent figura.



- A Java els paràmetres prenen el seu valor com una còpia del valor de l'expressió o variable utilitzada en la crida; aquest mecanisme de pas de paràmetres de denomina pas de **paràmetres per valor** o **per còpia**.

- C**òpia dels valors** de les variables utilitzades en la crida **a les variables emprades** com a paràmetres.



- Després d'executar la crida a la funció, se salta al cos de la funció, copiant el valor del primer paràmetre (el valor de **a**, que és **20**) a la primera variable utilitzada com a paràmetre (**hora**), el segon valor utilitzat en la crida (**4**), al segon paràmetre d'entrada (**minuts**), etc. El procés es repeteix per a cada parella valor-paràmetre.
- Qualsevol canvi en un paràmetre d'entrada que s'efectue dins del cos de la funció no repercuteix en la variable o expressió utilitzada en la crida, ja que el que es modifica és una còpia i no la dada original.

VALOR RETORNAT PER UNA FUNCIÓ

- El tipus **void** d'una funció indica que la funció no retorna res, dit d'una altra forma: la crida a la funció no se substitueix per cap valor.
- És possible utilitzar qualsevol tipus (**int**, **char**, **double**, etc.) per a especificar que la crida a la funció se substituirà per un valor del tipus indicat.

```
tipus nomFunció(paràmetres) {
    ....
    return (valor);
}
```

- La instrucció **return** finalitza la execució de la funció y tipus de **valor** ha de coincidir amb **tipus**.
- La instrucció **return** s'utilitza en funcions amb un tipus retornat diferent a **void**.

- Exemple d'una **funció** que realitza la **suma** de dos nombres enters:

```
static int suma (int n1, int n2) {
//cada crida retorna un int
        int resultat;
        resultat = n1 + n2;
        return (resultat);
 //substitueix la crida pel valor del resultat
}
```

SOBRECÀRREGA DE FUNCIONS

- Java permet que dos o més funcions compartisquen el mateix identificador en un mateix programa. Això es coneix com a **sobrecàrrega de funcions**.
- La manera de distingir entre les diferents funcions sobrecarregades és mitjançant les seues **llistes de paràmetres**, que han de ser **diferents**, ja siguen en número o en tipus.
- Suposem que volem dissenyar una funció per a calcular la **suma de dos enters**, però també és útil fer una **suma ponderada**, on cada sumand tinga un pes diferent.

- Exemple funcions sobrecarregades:

```
// funció sobrecarregada
static int suma(int a, int b) {
    int suma;
    suma = a+ b;
    return (suma);
}
// funció sobrecarregada
static doble suma(int a, double pesoA, int b, double pesoB) {
    double suma;
    suma = a * pesoA / (pesoA + pesoB) + b * pesoB / (pesoA + pesoB);
    return (suma);
}
```

- Si invoquem a la funció **suma()** de forma: **suma(2, 3**), s'executarà la primera versió, i retornarà 5. En canvi, si es crida amb: **suma(2, 0.25, 3, 0.75)**, s'executarà la segona versió, i retornarà 3,25.

RECURSIVITAT

- Una funció pot ser invocada des de qualsevol lloc: des del programa principal, des d'una altra funció i fins i tot des de dins del seu propi cos d'instruccions. Quan una funció s'invoca a si mateixa, direm que és una **funció recursiva**.

```
Static int funcioRecursiva() {
    ....
    funcioRecursiva(); //crida recursiva
    ....
}
```

- Dins de **funcioRecursiva()** s'invoca a **funcioRecursiva()**. Això ens porta a un **cicle infinit** de crides a la funció. Per a evitar-ho, hem d'habilitar un mecanisme que detinga, en algun moment, la sèrie cridaes recursives: una sentència **if** que, utilitzant una condició anomenada "**cas base o trivial**", impedisca que es continue amb una nova crida.

```
int funcioRecursiva(dades) {
    int resultat;
    if (cas base) {
        resultat = valorBase;
    } else {
        resultat = funcioRecursiva(novesDades); // crida recursiva
        ....
    }
    return (resultat);
}
```

- Només quan la condició del cas base siga **false**, es farà una nova crida **recursiva**. Quan el cas base siga **true** es trencarà la cadena de crides. La idea principal de la recursivitat és solucionar un problema reduint la seua grandària. Aquest procés continua fins que tinga una grandària tan xicoteta que la seua solució siga trivial.
- Per a aconseguir problemes cada vegada més xicotets, les dades d'entrada han de tendir cap al cas base. Conceptualment **novesDades** han de ser de menor grandària que **dades**; així garantim que en algun moment les dades utilitzades en la funció aconsegueixen el cas base, tallant la sèrie de crides recursives.
- Vegem l'exemple del factorial d'un número **n**:

  n! = n*(n-1)*(n-2)* ... 2*1 → 5! = 5*4*3*2*1
- La definició de factorial es pot escriure també de la següent manera:

  n! = n*(n-1)*(n-2)* ... 2*1 → n! = n*(n-1)!
- El **cas base** de factorial és: 0! = 1
- En cada crida, les dades d'entrada van sent menors i tendeixen cap al cas base: Per a calcular el factorial de **n**, utilitzem el factorial de **(n-1)** que, al seu torn, usarà el factorial de **(n-2)**, i així successivament, fins a arribar a **0**, el factorial del qual val 1. Aquest serà el cas base.

RECURSIVITAT

- Amb tota la informació de la qual disposes escriu una **funció** que calcule el **factorial** d'un número de forma recursiva:

```
long factorial (int n) {
    long resultat;
    if (n == 0) { //si n és 0
        resultat = 1; //cas base
    } else {
        resultat = n * factorial (n -1); //crida recursiva
    }
    return resultat;
}
```

- Fem una **traça** (execució pas a pas de les instruccions d'un programa) de la funció **factorial (3)**:

```
long factorial (3) {
    long resultat;
    if (3 == 0) { // fals
        ...
    } else {
        resultat = 3 * factorial (2);
```

- L'execució de **factorial(3)** queda a l'espera que s'execute **factorial(2)**.
- En aquest instant existeixen dues funcions **factorial()** en memòria. Vegem que ocorre en la crida a factorial(2):

```
long factorial (2) {
    long resultat;
    if (2 == 0) { // fals
        ...
    } else {
        resultat = 2 * factorial (1);
```

- Ara també **factorial(2)** es queda esperant a l'execució de **factorial(1)**. En aquest moment existeixen en memòria les funcions **factorial(3)** i **factorial(2)** esperant que acabe l'execució de **factorial(1)**. La nova crida s'executa de la següent manera:

```
long factorial (1) {
    long resultat;
    if (1 == 0) { // fals
        ...
    } else {
        resultat = 1 * factorial (0);
```

- De manera anàloga a les anteriors, es deté l'execució de la crida a la funció **factorial(1)** perquè comence a executar-se una nova instància de la funció; és l'últim cas, **factorial(0)**.

RECURSIVITAT

- En aquest moment de l'execució, estan a l'espera que finalitzen les respectives crides recursives diverses instàncies, o còpies, de la funció **factorial()**. Vegem com s'executa **factorial(0)**:

```
long factorial (0) {
    long resultat;
    if (1 == 0) { // cert
        resultat = 1;
    } else {
        ...
    }
    return (1);
}
```

- L'última instància de la funció acaba d'executar-se, retornant el valor 1, i permetent que la crida anterior (**factorial(1)**) prosseguisca la seua execució.

```
long factorial (1) {
        resultat = 1 * 1; //1
    }
    return (1);
}
```

- De nou la funció que s'executa actualment, **factorial(1)**, acaba retornant el valor 1 i permetent que la instància de la funció que esperava la seua finalització continue.

- Des del punt que es va quedar esperant, el factorial(2) prossegueix així:

```
long factorial (2) {
        resultat = 2 * 1; //2
    }
    return (2);
}
```

- Acaba retornant el control perquè seguisca la seua execució **factorial(3)**:

```
long factorial (3) {
        resultat = 3 * 2; //6
    }
    return (6);
}
```

- Finalitza la primera instància de la funció que es va invocar, retornant el control al programa principal o on es cridara. Una possible crida per a la traça anterior seria:

```
long solución = factorial(3);
System.out.println(solucion); //mostra 6
```

RECURSIVITAT

1. Amb tota la informació de la qual disposes escriu una **funció** que calcule el **factorial** d'un número de forma recursiva.
2. Dissenyar una **funció** que calcule $\mathbf{a^n}$, on a és real i n és sencer no negatiu. Realitzar una versió iterativa i una altra recursiva.