

Diseño Descendente. Abstracción Procedimental

Contenido del Tema

3.1. Diseño Descendente

3.2. Métodos: Procedimientos y Funciones

3.2.1. Ejemplos

3.2.2. Declaración de métodos. Parámetros Formales

3.2.3. Invocación a métodos. Parámetros Reales

3.2.4. Variables locales



Diseño Descendente. Abstracción Procedimental

Contenido del Tema

3.2.5. Interfaz

3.2.6. Criterios de modularización

3.2.7. Precondiciones y Postcondiciones.

Tratamiento de situaciones excepcionales



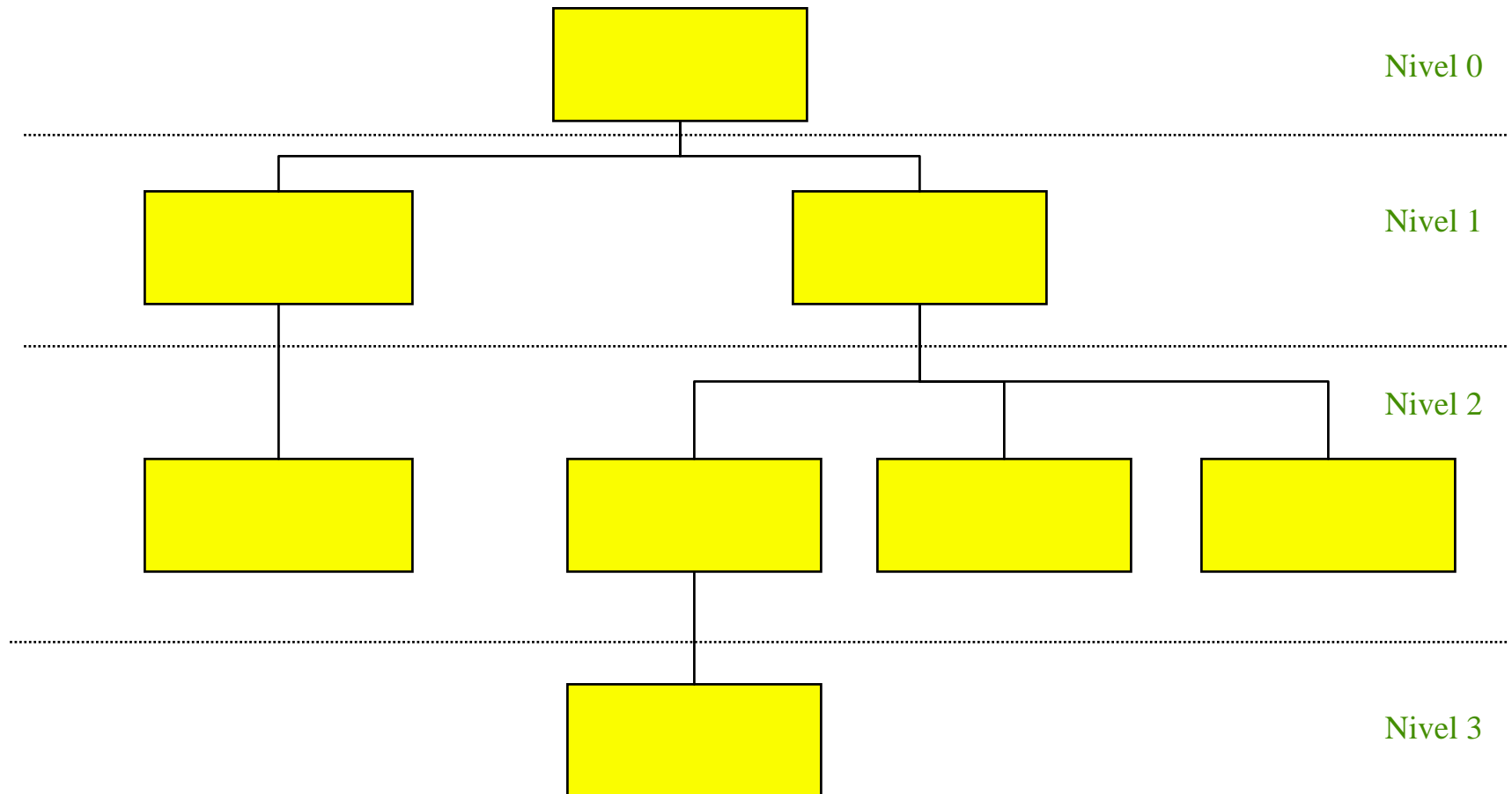
Diseño Descendente

- En la mayoría de los problemas reales, el algoritmo que los soluciona es demasiado largo y complejo para implementarlo mediante un único texto (programa).
- Desventajas de este tipo de programas:
 - Rigidez e inflexibilidad de los programas.
 - Pérdida excesiva de tiempo en corrección de errores.
 - Imposibilidad de reutilizar el programa o fragmentos suyos en proyectos futuros.

Diseño Descendente

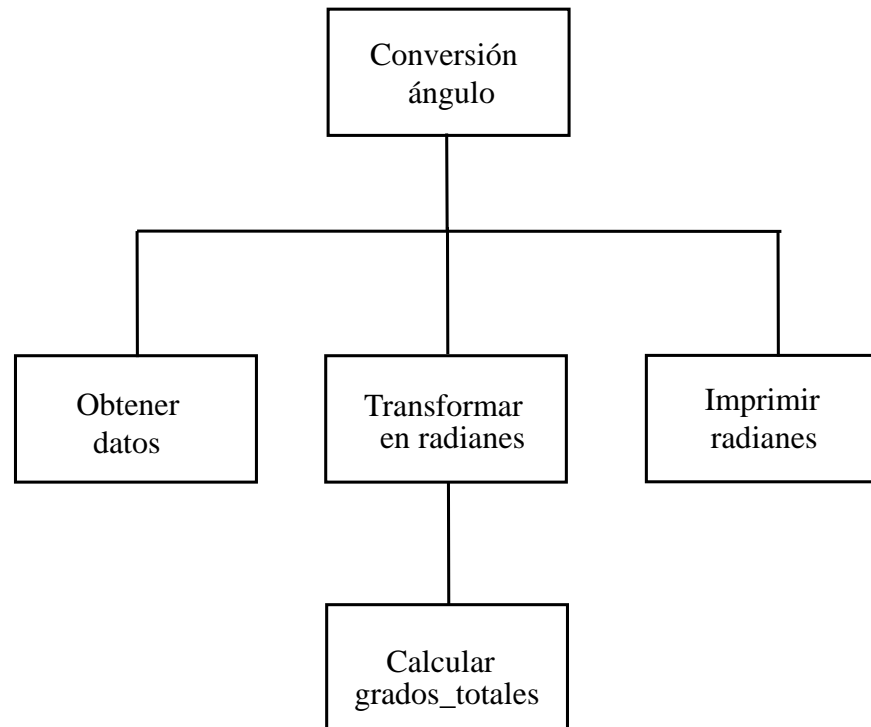
- Se hace necesaria la utilización de alguna metodología de diseño que evite estos inconvenientes
- La metodología de **Diseño Descendente** (“refinamientos sucesivos”, “Top-Down”, “divide y vencerás”) se ha mostrado como la más adecuada.
- Un **problema** se descompone en varios subproblemas y estos a su vez se descomponen en otros subproblemas hasta llegar a un nivel de **descomposición** que permita la solución sencilla de los diferentes **subproblemas**.
- La **solución** al problema inicial viene dada por la **composición** de cada una de las **soluciones** a los subproblemas

Diseño Descendente



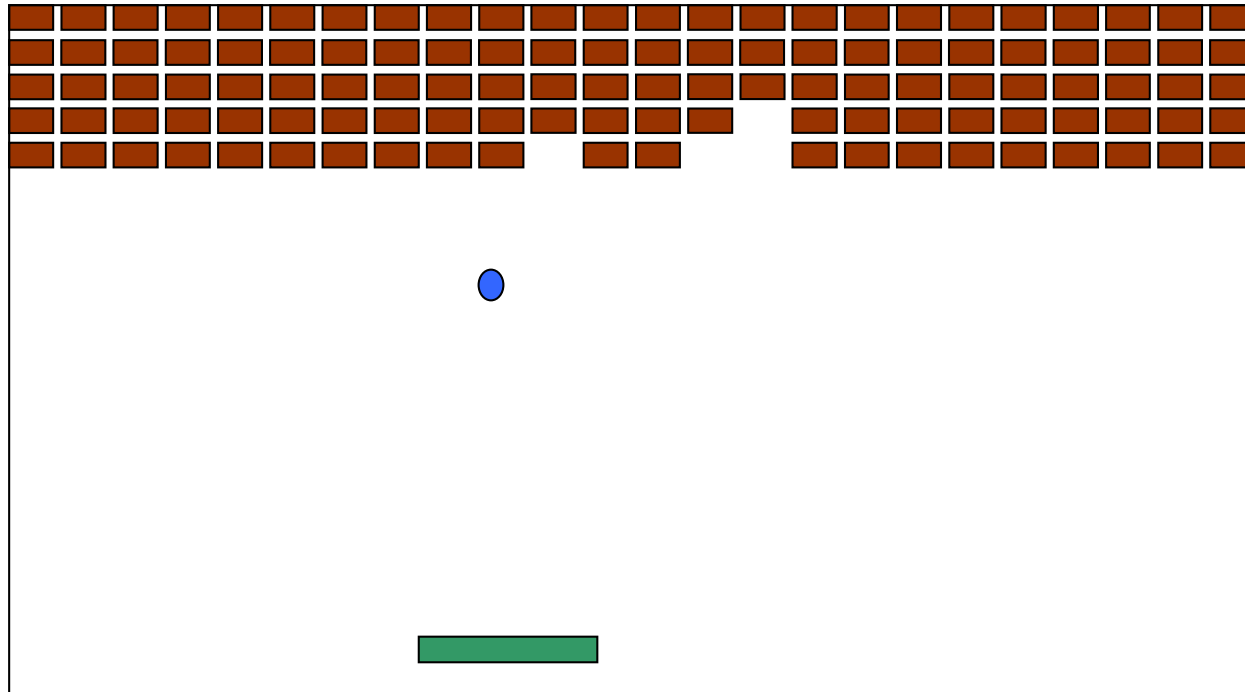
Diseño Descendente

- Ejemplo 1: Conversión a radianes de un ángulo expresado en grados, minutos y segundos.

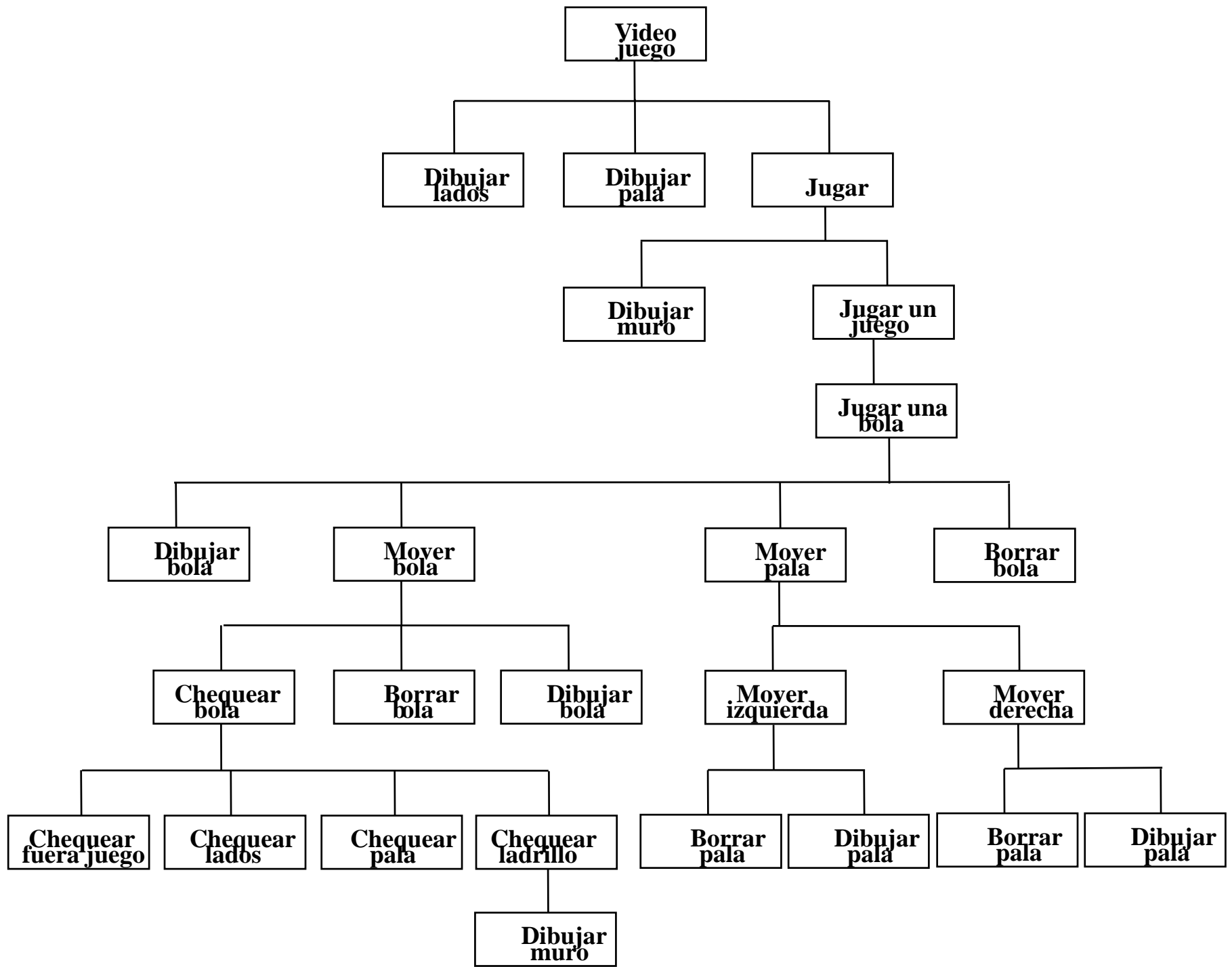


Diseño Descendente

- Ejemplo 2: Juego del Arkanoid.



Vidas: 4Puntuación: 50



Diseño Descendente

- Este enfoque proporciona indudables ventajas:
 - **Simplificación del diseño** de los programas. Un **programa** (solución al problema inicial) se estructura como una **composición de subprogramas** (soluciones a los diferentes subproblemas).
 - Mejor **comprensión y legibilidad** de los programas.
 - **Facilita la depuración** de errores.
 - **Programación aislada**.
 - Posibilidad de **reutilización** de los subprogramas.
 - **Abstracción Procedimental**

Diseño Descendente

Abstracción Procedimental

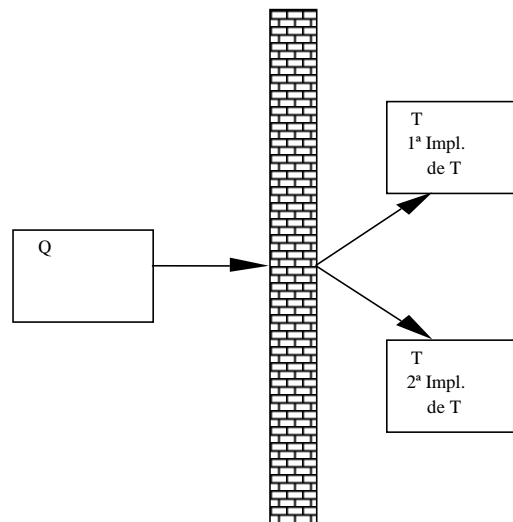
El programador, cuando diseña un subprograma:

- ❖ **NO tiene que saber cómo** otro subprograma (que necesita) resuelve su subproblema
- ❖ **SI tiene que saber qué** es lo que resuelve.

Diseño Descendente

Abstracción Procedimental

Aisla (encapsula) los diferentes subprogramas que componen un programa

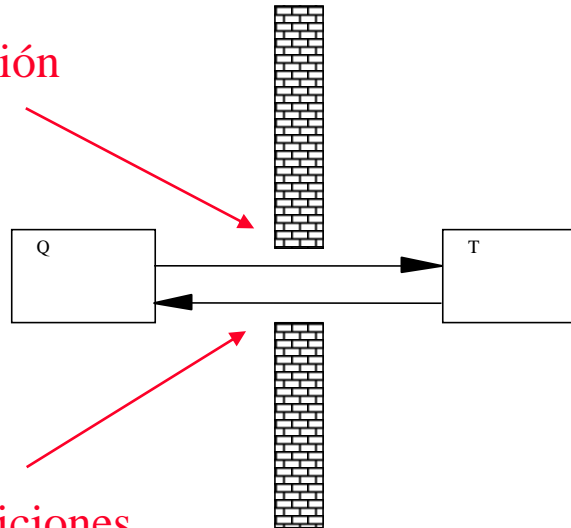


Diseño Descendente

Abstracción Procedimental

Aislamiento no puede ser total

Intercambio de información

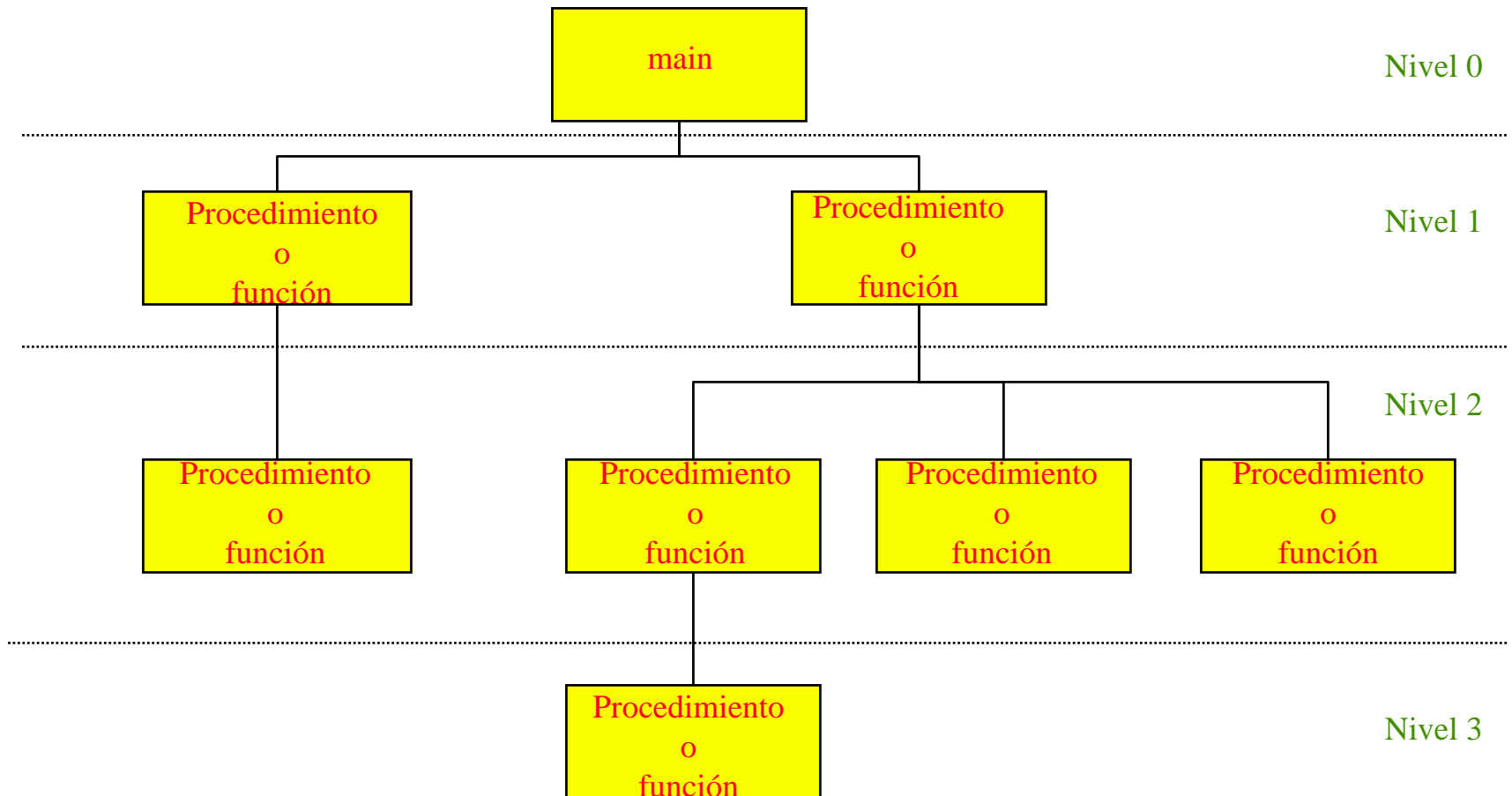


Precondiciones y Postcondiciones

Métodos: Procedimientos y Funciones

- En **Java** todos los subprogramas que conforman la solución a un problema se denominan **métodos**.
- En este Módulo 1 sólo se tratarán los denominados métodos “de clase” (**static**). En el Módulo 2 se verán otros métodos llamados “de instancia”.
- El método que iniciará la ejecución del programa es el método **main**.
- Existen dos tipos diferentes de métodos:
 - **Procedimientos**
 - **Funciones**

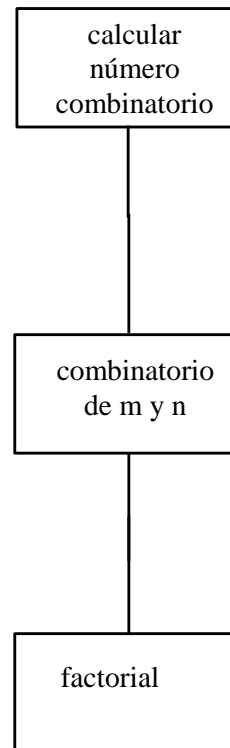
Métodos: Procedimientos y Funciones



Ejemplo 1:

Cálculo de un número combinatorio

$$\binom{m}{n} = \frac{m!}{n! (m-n)!}$$



Ejemplo 1:

Cálculo de un número combinatorio

$$\binom{m}{n} = \frac{m!}{n! (m-n)!}$$

calcular
número
combinatorio



combinatorio
de m y n

factorial

```
public static void main(String[] args) {
    Scanner teclado = new Scanner(System.in);
    int m,n,comb;
    do {
        System.out.print("Introduzca m y n (m >= n): ");
        m = teclado.nextInt();
        n = teclado.nextInt();
    } while ((m < n) || (m < 0) || (n < 0));

    comb = combinatorio(m,n);

    System.out.println("El numero combinatorio de "
        + m + " sobre " + n + " es: " + comb);
    teclado.close();
}
```

Abstracción Procedimental

- combinatorio

Ejemplo 1:

Cálculo de un número combinatorio

$$\binom{m}{n} = \frac{m!}{n! (m-n)!}$$

calcular
número
combinatorio

combinatorio
de m y n

factorial

```
private static int combinatorio(int m, int n) {  
    return factorial(m) /  
        (factorial(n) * factorial(m-n));  
}
```

Abstracción Procedimental

- factorial

Ejemplo 1:

Cálculo de un número combinatorio

$$\binom{m}{n} = \frac{m!}{n! (m-n)!}$$

calcular
número
combinatorio

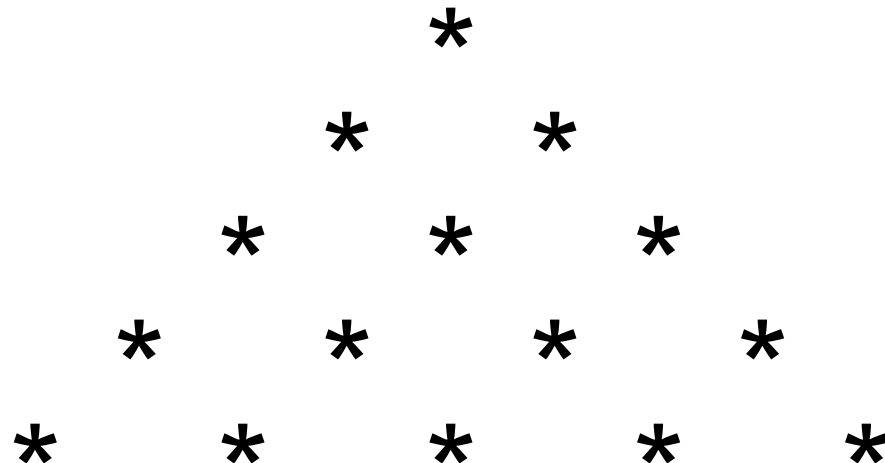
combinatorio
de m y n

factorial

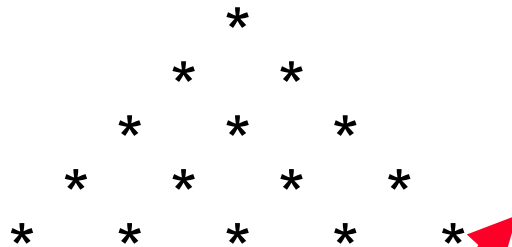
```
private static int factorial(int x) {  
    int fact = 1;  
    for (int i = 2; i <= x; i++) {  
        fact = fact * i;  
    }  
    return fact;  
}
```

Ejemplo 2: Dibujo de triángulo de asteriscos

Un programa que lea un número natural N por teclado y dibuje un triángulo de asteriscos con tantas líneas como indique N . Por ejemplo si $N=5$:



Ejemplo 2: Dibujo de triángulo de asteriscos



dibujar
triángulo
asteriscos

Dibujar
línea

Dibujar
blancos

Dibujar
asteriscos

```
public static void main(String[] args) {  
    Scanner teclado = new Scanner(System.in);  
    int n;  
  
    System.out.print("Introduzca n: ");  
    n = teclado.nextInt();  
    for (int linea = 1; linea <= n; linea++) {  
        dibujarLinea(linea, n);  
    }  
    teclado.close();  
}
```

Abstracción Procedimental

- dibujarLinea

Ejemplo 2: Dibujo de triángulo de asteriscos

```
      *
     * *
    * * *
   * * * *
  * * * * *
 * * * * *
```

dibujar
triángulo
asteriscos

Dibujar
línea

Dibujar
blancos

Dibujar
asteriscos

```
private static void dibujarLinea(int linea,
                                   int n) {
    dibujarBlancos(n-linea);
    dibujarAsteriscos(linea);
    System.out.println();
}
```

Abstracción Procedimental

- dibujarBlancos
- dibujarAsteriscos

Ejemplo 2: Dibujo de triángulo de asteriscos

```
      *
     * *
    * * *
   * * * *
  * * * * *
 * * * * *
```

dibujar
triángulo
asteriscos

Dibujar
línea

Dibujar
blancos

```
private static void dibujarBlancos(int blancos) {  
    for (int cont = 1; cont <= blancos; cont++) {  
        System.out.print(" ");  
    }  
}
```

Dibujar
asteriscos

Ejemplo 2: Dibujo de triángulo de asteriscos

```
      *
     * *
    * * *
   * * * *
  * * * * *
```

dibujar
triángulo
asteriscos

Dibujar
línea



Dibujar
blancos

Dibujar
asteriscos

```
private static void dibujarAsteriscos(int aster) {  
    for (int cont = 1; cont <= aster; cont++) {  
        System.out.print("* ");  
    }  
}
```

Declaración de métodos. Parámetros Formales

Declaración de un Procedimiento

cabecera  `private static void nombreProcedimiento(Parámetros Formales)`
`{`
`declaración de variables`
 `secuencia de sentencias`
`}`

```
private static void dibujarLinea(int linea, int n) {  
    dibujarBlancos(n-linea);  
    dibujarAsteriscos(linea);  
    System.out.println();  
}
```


Declaración de métodos. Parámetros Formales

Declaración de una Función

cabecera → **private static TipoResultado nombreFuncion** (Parámetros Formales)
{
 declaración de variables
 secuencia de sentencias (última sentencia "return"
 para devolver el resultado)
}

cuerpo →

```
private static int factorial(int x) {  
    int fact = 1;  
    for (int i = 2; i <= x; i++) {  
        fact = fact * i;  
    }  
    return fact;  
}
```

Invocación a métodos. Parámetros Reales

Invocación o llamada a un Procedimiento

```
nombreProcedimiento (Parámetros Reales) ;
```

Constituye por sí misma una sentencia

```
dibujarLinea (linea, n) ;
```

Invocación a métodos. Parámetros Reales

Invocación o llamada a una Función

nombreFuncion (Parámetros Reales)

NO constituye por sí misma una sentencia

~~combinatorio(m,n);~~

comb = combinatorio(m,n);

Reglas de uso de Parámetros

- 1) Número de parámetros formales = Número de parámetros reales.
- 2) El i-ésimo parámetro formal se corresponde con el i-ésimo parámetro real
- 3) El tipo del i-ésimo parámetro formal debe ser igual que el tipo del i-ésimo parámetro real (pueden ser distintos si al hacer una conversión implícita no se produce error).
- 4) Los parámetros de un procedimiento o una función pueden ser de cualquier tipo, al igual que cualquier variable.
- 5) Los nombres de un parámetro formal y su correspondiente real pueden o no ser iguales.
- 6) El paso de parámetros en Java siempre es “Por Valor”, esto es, se realiza una copia del valor del parámetro real en el parámetro formal correspondiente. Dentro del método, ambos elementos son independientes. Por ejemplo, una modificación del parámetro formal no afecta al valor del parámetro real. [En el Tema 4 profundizaremos.](#)

Variables Locales

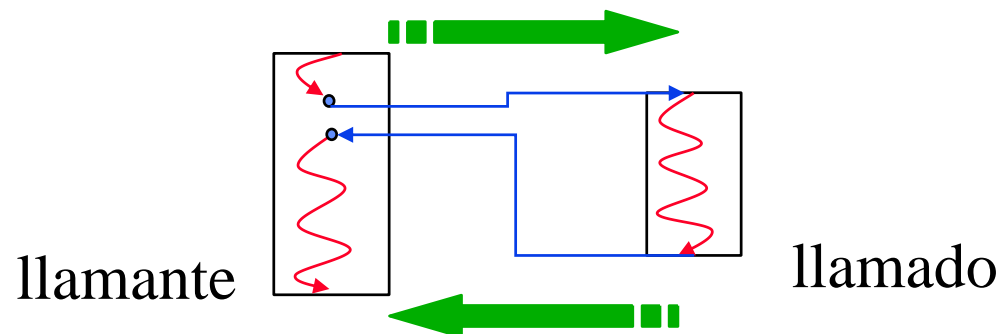
- Las variables declaradas en un método se denominan variables locales a ese método.
- Los parámetros formales de un método también son variables locales del mismo.
- También se pueden declarar variables locales en bloques específicos del código de un método:
 - En una sentencia for
 - En un bloque { ... }

Variables Locales

- Una variable local sólo puede ser accedida dentro del método/bloque que la declara y a partir del punto donde se declara (regla de ámbito).
- Una variable local se crea cuando se declara y se destruye cuando termina el método/bloque donde se declara.
- El compilador producirá un error si se intenta utilizar una variable local no inicializada en algún punto donde se requiere su valor

Flujo de Control

- Cuando se produce una invocación o llamada a un método:
 - Se establecen las vías de comunicación entre llamante y llamado a través de los parámetros. Es decir, se realiza la copia correspondiente.
 - Se crean las variables declaradas en el llamado
 - El flujo de control pasa a la primera instrucción del llamado
- Cuando acaba el método llamado:
 - Se destruyen las variables declaradas en el llamado (y los parámetros formales)
 - El flujo de control continúa por la instrucción que sigue a la de llamada en el llamante



Genéricamente. Define la interacción entre dos entidades independientes

En programación. Define la forma en que se comunican y cooperan dos subprogramas.

Diseño del interfaz de un subprograma (método):

- Qué información necesita para resolver el problema
- Qué información produce como resultado
- Bajo qué condiciones se realiza el intercambio de información

Criterios de Modularización

- Los subprogramas también se denominan **módulos**.
- No existen mecanismos formales para determinar **cómo descomponer un problema en módulos**, es una **labor subjetiva**.
- Podemos guiarnos por algunos criterios generales:
 - Acoplamiento
 - Cohesión

Criterios de Modularización

ACOPLAMIENTO

- Un objetivo en el diseño descendente es crear módulos aislados e independientes.
- Debe haber alguna interacción entre módulos para formar un sistema coherente.
- Dicha interacción se conoce como acoplamiento.
- Maximizar la independencia será **minimizar el acoplamiento**.

Criterios de Modularización

Ej. Suma de la serie $1, x, x^2/2!, x^3/3!, \dots$

```
static final double UMBRAL = 0.01;
public static void main(String[] args){
    Scanner tec = new Scanner(System.in);
    double x, suma;

    System.out.print("Introduzca X: ");
    x = tec.nextDouble();
    suma = 0;
    System.out.println("La suma de la
        serie es: " + sumaSerie(suma,x));
    tec.close();
}
```

```
private static double sumaSerie(double s,
                                double x) {
    double term, res;
    int i;

    i = 0;
    res = s;
    do {
        term = calcTerm(i,x);
        res = res + term;
        i++;
    } while (term >= UMBRAL);
    return res;
}
```

ALTO ACOPLAMIENTO

Criterios de Modularización

Ej. Suma de la serie $1, x, x^2/2!, x^3/3!, \dots$

```
static final double UMBRAL = 0.01;
public static void main(String[] args){
    Scanner tec = new Scanner(System.in);
    double x, suma,

    System.out.print("Introduzca X: ");
    x = tec.nextDouble();
    suma = 0;
    System.out.println("La suma de la
        serie es: " + sumaSerie(suma, x));
    tec.close();
}
```

```
private static double sumaSerie(double s,
                                double x) {
    double term, res;
    int i;

    i = 0;
    res = s;
    do {
        term = calcTerm(i, x);
        res = res + term;
        i++;
    } while (term >= UMBRAL);
    return res;
}
```

ALTO ACOPLAMIENTO

Criterios de Modularización

Ej. Suma de la serie $1, x, x^2/2!, x^3/3!, \dots$

```
static final double UMBRAL = 0.01;
public static void main(String[] args){
    Scanner tec = new Scanner(System.in);
    double x;

    System.out.print("Introduzca X: ");
    x = tec.nextDouble();
    System.out.println("La suma de la
        serie es: " + sumaSerie(x));
    tec.close();
}
```

```
private static double sumaSerie(double x)
{
    double term, res;
    int i;

    i = 0;
    res = 0;
    do {
        term = calcTerm(i, x);
        res = res + term;
        i++;
    } while (term >= UMBRAL);
    return res;
}
```

BAJO ACOPLAMIENTO

COHESIÓN

- Hace referencia al grado de relación entre las diferentes partes internas a un módulo.
- Si la cohesión es muy débil, la diversidad entre las distintas tareas realizadas dentro de un módulo es tal que posteriores modificaciones podrán resultar complicadas.
- Se busca **maximizar la cohesión** dentro de cada módulo.

Criterios de Modularización

Ej. Escribir si un número es primo

Opción 1

```
private static void escribirSiPrimo(int num) {  
    int divisor, tope;  
    if (num < 2) {  
        System.out.println("NO es primo");  
    } else {  
        tope = num - 1;  
        divisor = 2;  
        while ((divisor <= tope) && (num % divisor != 0)) {  
            divisor++;  
        }  
        if (divisor > tope) {  
            System.out.println("SI es primo");  
        } else {  
            System.out.println("NO es primo");  
        }  
    }  
}
```

BAJA COHESIÓN

Criterios de Modularización

Ej. Escribir si un número es primo

Opción 2

```
private static boolean esPrimo(int num) {  
    boolean res = false;  
    int divisor, tope;  
    if (num >= 2) {  
        tope = num-1;  
        divisor = 2;  
        while ((divisor <= tope) && (num % divisor != 0)){  
            divisor++;  
        }  
        res = divisor > tope;  
    }  
    return res;  
}
```

```
private static void  
    escribirSiPrimo(int num) {  
    if (esPrimo(num)) {  
        System.out.println("SI es primo");  
    } else {  
        System.out.println("NO es primo");  
    }  
}
```

ALTA COHESIÓN

Criterios de Modularización

Ej. Escribir si un número es primo

Opción 2

```
private static boolean esPrimo(int num) {  
    boolean res = false;  
    int divisor, tope;  
    if (num >= 2) {  
        tope = (int) Math.sqrt(num);  
        divisor = 2;  
        while ((divisor <= tope) && (num % divisor != 0)){  
            divisor++;  
        }  
        res = divisor > tope;  
    }  
    return res;  
}
```

```
private static void  
    escribirSiPrimo(int num) {  
    if (esPrimo(num)) {  
        System.out.println("SI es primo");  
    } else {  
        System.out.println("NO es primo");  
    }  
}
```

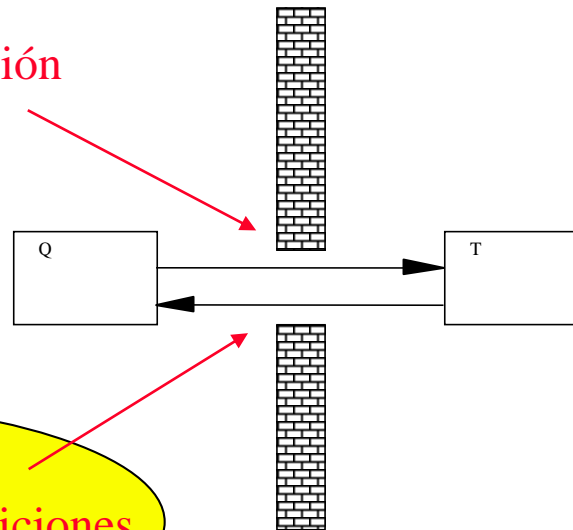
ALTA COHESIÓN

Precondiciones y Postcondiciones. Tratamiento de Situaciones Excepcionales

Abstracción Procedimental

Aislamiento no puede ser total

Intercambio de información



Precondiciones y Postcondiciones

Precondiciones y Postcondiciones. Tratamiento de Situaciones Excepcionales

- *Precondiciones*: condiciones que deben cumplirse antes de que el subprograma se ejecute, con objeto de garantizar que se puede realizar la tarea.
- *Postcondiciones*: condiciones que el subprograma garantiza tras finalizar su ejecución, suponiendo que las precondiciones se cumplieron cuando el subprograma fue llamado.

Precondiciones y Postcondiciones. Tratamiento de Situaciones Excepcionales

- Cuando se codifica un subprograma es buena práctica anteponer unos comentarios especificando claramente las precondiciones y postcondiciones del mismo.

```
// precondition: m >= n
// postcondicion: devuelve combinatorio de m sobre n
private static int combinatorio(int m, int n) {
    return factorial(m) / (factorial(n) * factorial(m-n));
}
```

Precondiciones y Postcondiciones. Tratamiento de Situaciones Excepcionales

- De cualquier forma esto no es suficiente, pues el procedimiento o función que utilice (llame) el subprograma diseñado, puede o no tener en cuenta las precondiciones de uso del mismo.
- Por ejemplo, un subprograma podría llamar a combinatorio con dos valores de m y n tales que $m < n$, saltándose la precondición de uso de combinatorio que establece que se debe cumplir $m \geq n$. ¿Qué ocurriría?
- En algún momento el sistema terminará el programa con un error de ejecución inesperado o bien terminará dándonos un resultado incorrecto.

Precondiciones y Postcondiciones. Tratamiento de Situaciones Excepcionales

- Cuando se diseñan procedimientos y funciones es fundamental tener en cuenta este tipo de situaciones, ya que estamos diseñando de forma separada un subprograma que será utilizado (llamado) por otro.
- Para ello podemos hacer uso del lanzamiento de **excepciones**.

```
// precondicion: m >= n
// postcondicion: devuelve combinatorio de m sobre n
private static int combinatorio(int m, int n) {
    if (m < n) {
        throw new RuntimeException("Error: (m<n) en función combinatorio");
    }
    return factorial(m) / (factorial(n) * factorial(m-n));
}
```

Precondiciones y Postcondiciones. Tratamiento de Situaciones Excepcionales

- Cuando se diseñan procedimientos y funciones es fundamental tener en cuenta este tipo de situaciones, ya que estamos diseñando de forma separada un subprograma que será utilizado (llamado) por otro.
- Para ello podemos hacer uso del lanzamiento de **excepciones**.

```
// precondicion: m >= n
// postcondicion: devuelve
private static int combinatorio
{
    if (m < n) {
        throw new RuntimeException("Error: (m<n) en función combinatorio");
    }
    return factorial(m) / (factorial(n) * factorial(m-n));
}
```

En el Módulo 2 se profundizará sobre Excepciones