Guía de Referencia Rápida - Java

Contenido

Tipos de datos, en Java

Tipos de datos primitivos		
Tipo de dato	**Valor, por defecto**	**Tamaño (en bytes) 1 byte = 8 bits**
boolean	FALSE	1 bit
char	" "(espacio)	2 byte
byte	0	1 byte
short	0	2 byte
int	0	4 byte
long	0	8 byte
float	0.0f	4 byte
double	0.0d	8 byte

Tipos de datos no primitivos
String
Array
Class
Interface

Conversión de tipos (`type casting`)

Es un método de conversión de una variable de un tipo de datos a otro tipo de datos de manera que las funciones pueden procesar correctamente estas variables.

Java define dos tipos de conversiones:

- **Conversión de tipos implícitos** (Widening): Almacenar una variable de un tipo de datos más pequeño en un tipo de datos más grande.
- **Conversión de tipos explícito** (Narrowing): Almacenamiento de la variable de un tipo de datos más grande en un tipo de datos más pequeño.

Operadores, en Java

CATEGORÍA DE OPERADOR	OPERADORES
Operadores Aritmético	**+, -, /, *, %**
Operadores Relacionales	**<, >, <=, >=,==, !=**
Operadores Lógicos	**&& , ||**
Operadores Asignación	**=, +=, −=, ×=, ÷=, %=, &=, ^=, |=, <<=, >>=, >>>=**
Operadores de Incremento y Decremento	**++ , - -**
Operadores Condicionales	**?:**
Operadores manipulación de bits	**^, &, |**
Operadores Especiales	**.** (operador de punto para acceder a métodos de clase)

Java IDE y código de ejecución

Entre muchos IDE, los más recomendados son:

- NetBeans - https://netbeans.apache.org/
- Eclipse - https://www.eclipse.org/
- JetBrains IntelliJ IDEA - https://www.jetbrains.com/

El código Java también puede escribirse en cualquier editor de texto y compilarse en el terminal con los siguientes comandos:

```
$ java [nombre_del_archivo].java
$ java [nombre_del_archivo]
```

Nota: El nombre del archivo debe ser el mismo que el nombre de la clase que contiene el método `main()`, con una extensión .java.

Variables, en Java

Las variables son el nombre de la ubicación de la memoria. Es un contenedor que contiene el valor mientras se ejecuta el programa java.

Las variables son de tres tipos en Java:

Variable Local	Variable global o de instancia	Variable Estática
Declarado e inicializado dentro del cuerpo del método, bloque o constructor.	Declarado dentro de la clase pero fuera del método, bloque o constructor. Si no se inicializa, el valor predeterminado es 0.	Declarado usando una palabra clave "`static`" No puede ser local.
Se tiene acceso solo dentro del método en el que se declara y se destruye posteriormente desde el bloque o cuando se devuelve la llamada a la función.	Las variables se crean cuando se crea una instancia de la clase y se destruyen cuando se destruye la clase.	Las variables creadas y crean una única copia en la memoria que se comparte entre todos los objetos a nivel de clase.

```
class PruebaVariables {
    int dato = 20;                  // variable de instancia
    static int numero = 10;         // variable estática
    void unMetodo(){
        int num = 30;               // variable local
    }
}
```

Palabras Reservadas

Palabra	Uso
`abstract`	Utilizada para declarar una clase abstracta.
`catch`	Utilizada para detectar excepciones generadas por declaraciones `try`.
`class`	Utilizada para declarar una clase.
`enum`	Define un conjunto de constantes
`extends`	Indica que la clase es heredada
`final`	Indica que el valor no se puede cambiar
`finally`	Utilizada para ejecutar código después de la estructura `try-catch`.
`implements`	Utilizada para implementar una `interface`.
`new`	Utilizada para crear nuevos objetos.
`static`	Utilizada para indicar que una variable o un método es un método de clase.
`super`	Utilizada para referirse a la clase principal.
`this`	Utilizada para hacer referencia al objeto actual en un método o constructor.
`throw`	Utilizada para lanzar explícitamente una excepción.
`throws`	Utilizada para declarar una excepción.
`try`	Bloque de código para manejar una excepción

Métodos, en Java

La forma general de método:

1. `tipo`: tipo de retorno del método
2. `nombre`: nombre del método
3. `lista de parámetros`: secuencia de tipo y variables separadas por una coma
4. `return`: declaración para devolver valor a la rutina de llamada

```
tipo nombre (lista de parámetros)
{
  // cuerpo del método
  // return valor (solo si el tipo no es nulo)
}
```

Declaraciones Condicionales, en Java

1. IF - ELSE

Prueba la condición (**`if)`**, si la condición es verdadera si se ejecuta el bloque, de lo contrario, se ejecuta el bloque **`else`**.

```
class PruebaIfElse{
    public static void main(String args[]){
        int porcentaje = 50;
        if(porcentaje >= 50) {
            System.out.println("Aprobado ");
        }
        else {
            System.out.println("Intente de nuevo!");
        }
    }
}
```

2. SWITCH

Pruebe la condición (**`switch`**), si un caso particular es verdadero, el control se pasa a ese bloque y se ejecuta. El resto de los casos no se consideran más y el programa se sale del ciclo.

```
class PruebaSwitch{
    public static void main(String args[]){
        int tiempo = 0;
        switch(tiempo){
            case 0 :
                System.out.println("Soleado");
                break;
            case 1 :
                System.out.println("Lluvioso");
                break;
            case 2 :
                System.out.println("Frio");
                break;
            case 3 :
                System.out.println("Ventoso");
                break;
            default :
                System.out.println("Agradable");
        }
    }
}
```

3. BUCLES

Los bucles se utilizan para iterar el código un número específico de veces hasta que la condición especificada sea verdadera.

Hay tres tipos de bucles en Java:

Bucle `For`
Repite el código un número específico de veces hasta que la condición es verdadera.

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

Bucle `While`
Si la condición en el 'while' es verdadera, el programa entra en el ciclo de iteración.

```
class PruebaBucleWhile{
    public static void main (String args[]){
        int i = 1;
        while(i <= 10){
            System.out.println(i);
            i++;
        }
    }
}
```

Bucle `Do While`
El programa entra en el ciclo de iteración al menos una vez, independiente de que la condición 'while' sea verdadera. Para más iteraciones, depende de que la condición 'while' sea verdadera.

```
class PruebaBucleDoWhile{
    public static void main (String args[]){
        int i = 1;
        do {
            System.out.println(i);
            i++;
        } while(i <= 10);
    }
}
```

Conceptos OOP[1], en Java

Un paradigma orientado a objetos ofrece los siguientes conceptos para poder simplificar el desarrollo y el mantenimiento de software.

1. Objetos y Clases

Los ***objetos*** son entidades básicas, en tiempo de ejecución, en un sistema orientado a objetos, que contienen datos y código para manipular datos.

Todo este conjunto de datos y código se puede convertir en un tipo de datos definido por el usuario utilizando el concepto de ***CLASE***.

Por tanto, una clase es una colección de objetos de un tipo de datos similar.

Ejemplo: la manzana, el mango y la naranja son miembros de la clase de frutas.

2. Encapsulación y Abstracción de datos

La envoltura o revestimiento de seguridad de los datos y los métodos en una sola unidad se conoce como encapsulación. Tomemos la cápsula médica, como ejemplo, no sabemos qué químico contiene, solo nos preocupa su efecto. Este aislamiento de datos del acceso directo por parte del programa se denomina ocultación de datos.

Por ejemplo, al usar aplicaciones, nos preocupa su funcionalidad y no por el código detrás, de ella

3. Herencia

La herencia proporciona el concepto de reutilización, es el proceso mediante el cual los objetos de una clase (clase secundaria o subclase) heredan o derivan propiedades de objetos de otra clase (clase principal).

Tipos de herencia, en Java

- **Herencia simple**: El hijo hereda propiedades de clase y el comportamiento de una sola clase padre.
- **Herencia jerárquica**: Cuando una clase principal tiene dos clases secundarias que heredan sus propiedades.
- **Herencia multinivel**: La clase secundaria hereda propiedades de su clase principal, que a su vez es una clase secundaria de otra clase principal.

Caso especial

- **Herencia múltiple**: Cuando una clase secundaria tiene dos clases principales. En Java, este concepto se logra '*simular'* mediante el uso de *interfaces*.

[1] Object-Oriented Programming

```
class ClaseA {
    int i, j;
    void muestraij() {
        System.out.println("i y j: " + i + " " + j);
    }
}

// Crea una subclase extendiendo la claseA.
class ClaseB extends ClaseA {
    int k;
    void muestrak() {
        System.out.println("k: " + k);
    }
    void sum() {
        System.out.println("i+j+k: " + (i+j+k));
    }
}

class HerenciaSimple {
    public static void main(String args[]) {
        ClaseA objA = new ClaseA();
        ClaseB objB = new ClaseB ();
        // La superclase puede usarse sola
        objA.i = 10;
        objA.j = 20;
        System.out.println("Contenido de objA: ");
        objA. muestraij();
        System.out.println();

      /* La subclase puede acceder a todos los miembros
         publicos de su superclase. */
        objB.i = 7;
        objB.j = 8;
        objB.k = 9;
        System.out.println("Contenido de objB: ");
        objB.muestraij();
        objB.muestrak();
        System.out.println();
        System.out.println("Suma de i, j y k en objB:");
        objB.sum();
    }
}
```

Algunas limitaciones en la herencia:

- Los miembros privados de la superclase no pueden derivarse de la subclase.
- La subclase no puede heredar los constructores.
- Puede haber una superclase por subclase.

4. Polimorfismo

Definido como la capacidad de tomar más de una forma. El polimorfismo permite crear código limpio y legible.

En Java, el polimorfismo se logra mediante el concepto de *reemplazo* y *sobrecarga* de métodos, que es el enfoque dinámico.

4.1. Reemplazo de métodos - *Overriding*

En la jerarquía de clases, cuando un método en una clase secundaria tiene el mismo nombre y tipo que un método en su clase principal, se dice que el método en la clase secundaria reemplaza(sobrescribe) el método en la clase principal.

En el siguiente código, si no reemplazamos el método la salida sería 4, como se calcula en la clase MathPabre; de lo contrario, sería 16.

```
class MathPabre{
    void area(){
      int a = 2;
      System.out.printf("Área del cuadrado con 2 = %d %n", a * a);
      System.out.println();
    }
}

class MathHija extends MathPabre{
    void area(){
      int a = 4;
      System.out.printf("Área del cuadrado con 4 = %d %n", a * a);
    }

    public static void main (String args[]){
        MathHija obj = new MathHija();
        obj.area();
    }
}
```

4.2. Sobrecarga de métodos - *Overloading*

La programación Java puede tener dos o más métodos en la misma clase que comparten el mismo nombre, siempre que sus declaraciones de argumentos sean diferentes. Estos métodos se denominan sobrecargados y el proceso se denomina *sobrecarga* de métodos.

Existen tres posibles formas de sobrecargar un método:

1. **Número de parámetros**
   ejemplo: add(int, int)
            add(int, int, int)

2. **Tipo de datos de los parámetros**
   ejemplo: add(int, int)
            add(int, float)

3. **Secuencia de tipo de datos de parámetros**
   ejemplo: add(int, float)
            add(float, int)

Programa para explicar la herencia multinivel y la sobrecarga de métodos.:

```
class Figura {
    void area(){
     System.out.println("El área de las siguientes formas es: ");
    }
}
class Cuadrado extends Figura {
    void area(int lado){
        // calcular el area del cuadrado
        int area = lado * lado;
        System.out.println("Área del cuadrado: "+area);
    }
}
class Rectangulo extends Figura {
    void area(int lango,int ancho){
        //calculate el area del rectangulo
        int area = lango * ancho;
        System.out.println("Área del rectángulo : " + area);
    }
}
class Circulo extends Figura {
    void area(int radio){
        float area = 3.14f * radio * radio;
        System.out.println("Área del circulo : " + area);
    }
}
//**************************
class HerenciaSobrecarga{
    public static void main(String[] args){
        int largo = 5;
        int ancho = 7;
        Figura figura = new Figura();
        //objecto hijo class cuadrado
        Cuadrado cuadrado = new Cuadrado();
        //objecto hijo class rectangulo
        Rectangulo rectangulo = new Rectangulo();
        // objecto hijo class circulo
        Circulo circulo = new Circulo();
        //llamar a los métodos de área de todas las clases
        //secundarias para obtener el área de diferentes objetos
        figura.area();
        cuadrado.area(largo);
        rectangulo.area(largo,ancho);
        circulo.area(largo);
    }
}
```

Clases Abstractas

La clase abstracta es una superclase que solo define una forma generalizada que será compartida por todas sus subclases, dejando a cada subclase las implementaciones sus métodos.

```
abstract class ClaseA {
    abstract void llamame();
    //Los métodos concretos están permitidos en clases abstractas.
    void llamame2() {
        System.out.println("Este es un metodo concreto.");
    }
}
class ClaseB extends ClaseA {
    void llamame() {
        System.out.println("Implementación en B de 'llamame'");
    }
}
class PruebaAbstracta {
    public static void main(String args[]) {
        ClaseB b = new ClaseB();
        b.llamame();
        b.llamame2();
    }
}
```

Interfaces

Con la interfaz se puede abstraer completamente de su implementación utilizando la palabra clave "**interface**". Son similares a la clase excepto que carecen de variables de instancia y sus métodos se declaran vacíos.

- Varias clases pueden implementar una interfaz.
- Las interfaces se utilizan para implementar múltiples herencias.
- Las variables son públicas, finales y estáticas.
- Para implementar una interfaz, una clase debe crear un conjunto completo de métodos definidos por una interfaz.
- Las clases que implementan interfaces pueden definir sus propios métodos.

```
interface Area {
    final static float pi = 3.14F;
    float calcular(float x , float y);
}


class Rectangulo implements Area {
    public float calcular(float x, float y){
        return (x*y);
    }
}
```

```
class Circulo implements Area {
    public float calcular(float x, float y) {
        return (pi * x * x);
    }
}

class PruebaInterface {
    public static void main(String args[]){
        float x = 2.0F;
        float y = 6.0F;
        Rectangulo rect = new Rectangulo(); //creando objeto
        Circulo cir = new Circulo();
        float result1 = rect.calcular(x,y);
        System.out.println("Area de Rectangulo = "+ result1);
        float result2 = cir.calcular(x,y);
        System.out.println("Area del Circulo = "+ result2);
    }
}
```

Constructores, en Java

- Un constructor inicializa un objeto.
- El constructor tiene el mismo nombre que la clase.
- No tienen ningún tipo de `return`, ni siquiera *null*.
- El constructor no puede ser *estático*, *abstracto* o *final*.

Los constructores pueden ser:

1. **CONSTRUCTOR NO PARAMETRIZADO O PREDETERMINADO:** Se invoca automáticamente incluso si no se declara

```
class Caja {
    double anchura;
    double altura;
    double profundidad;
    // Este es el constructor de Caja.
    Caja() {
        System.out.println("Constructor de CAJA");
        anchura = 10;
        altura = 10;
        profundidad = 10;
    }
    // calcular y devolver el volumen
    double volumen() {
        return anchura * altura * profundidad;
    }
}
```

```
class PruebaVolumenCaja {
    public static void main(String args[]) {
        // declarar, asignar e inicializar objetos Caja
        Caja caja1 = new Caja();
        Caja caja2 = new Caja();
        double vol;
        vol = caja1.volumen();
        System.out.println("El volumen de caja1 es " + vol);
        vol = caja2.volumen();
        System.out.println("El volumen de caja2 es " + vol);
    }
}
```

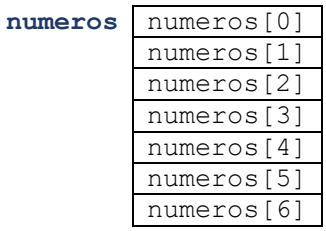
2. **PARAMETRIZADO**: Se utiliza para inicializar los atributos de la clase con valores predefinidos por parte del usuario.

```
class Caja {
    double anchura;
    double altura;
    double profundidad;

    Caja(double anc, double alt, double pro) {
        anchura = anc;
        altura = alt;
        profundidad = pro;
    }

    double volumen() {
        return anchura * altura * profundidad;
    }
}

class VolCajaP {
    public static void main(String args[]) {
        Caja caja1 = new Caja(10, 20, 15);
        Caja caja2 = new Caja(3, 6, 9);
        double vol;
        vol = caja1.volumen();
        System.out.println("El volumen de caja1 es " + vol);
        vol = caja2.volumen();
        System.out.println("El volumen de caja2 es " + vol);
    }
}
```

Arrays, en Java

Un Array(matriz) es un grupo de variables de tipo similar a las que se hace referencia por un nombre, el cual tiene memoria contigua. Se pueden almacenar en una matriz valores u objetos. Esta estructura proporciona buena optimización de código, ya que podemos ordenar los datos de manera eficiente y también acceder a ellos de forma aleatoria.

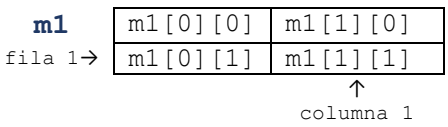
Hay dos tipos de matrices definidas en Java:



1. **Unidimensional**: Los elementos se almacenan en una sola fila.

```
class PruebaArraySimple {
    public static void main(String args[]) {
        int lon = 0;
        //declaración
        int [] numeros = {1,2,3,4,5,6,7};
        System.out.println("Los elementos de la matriz son: ");
        for(int i = 0; i < numeros.length ;i++){
            System.out.print(numeros[i] + " ");
        }
        System.out.println();
        int sum = 0;
        for(int i = 0; i < numeros.length ; i++) {
            sum = sum + numeros[i];
        }
        System.out.println("Suma de los elementos = " + sum);
    }
}
```

2. **Multidimensional**: Los elementos se almacenan en filas y columnas.




```
class PruebaArrayMultidimensional {
    public static void main(String args[]) {
        int[][] m1 = {{1, 2}, {3, 4}};
        int[][] m2 = {{5, 6}, {7, 8}};
        int[][] sum = new int[2][2];
        //Imprimir matriz 1
        System.out.println("La matriz 1 contiene : ");
        for (int i = 0; i < m1.length; i++) {
            for (int j = 0; j < m1.length; j++) {
                System.out.print("m1[" + i + " , " + j + "]:"
                        + m1[i][j] + " ");
            }
            System.out.println();
        }
        //Suma de matrices
        System.out.println("La suma de las matrices es : ");
        for (int i = 0; i < m1.length; i++) {
            for (int j = 0; j < m2.length; j++) {
                sum[i][j] = m1[i][j] + m2[i][j];
                System.out.print(sum[i][j] + " ");
            }
            System.out.println();
        }
    }
}
```

Códigos típicos de manejo de Arrays.

Crear un array con valores aleatorios	`double[] a = new double[n]`
	`for (int i = 0; i < n; i++)`
	`  a[i] = Math.random();`

Imprimir los valores del array, uno por línea	`for (int i = 0; i < n; i++)` `  System.out.println(a[i]);`

Buscar el mayor de los valores del array	`double max = Double.NEGATIVE_INFINITY`
	`for (int i = 0; i < n; i++)`
	`  if (a[i] > max) max = a[i];`

Calcular la media de los valores del array	`double suma = 0.0`
	`for (int i = 0; i < n; i++)`
	`  suma += a[i];`
	`double media = suma / n;`

Copiar los valores de un array en otro array	`double[] b = new double[n]`
	`for (int i = 0; i < n; i++)`
	`  b[i] = a[i];`

Strings, en Java

Los *Strings* (cadenas) son un tipo de datos no primitivo el cual representa una secuencia de caracteres.

- El tipo de cadena se utiliza para declarar variables de cadena.
- También se puede declarar una matriz de cadenas.
- Las cadenas de Java son inmutables, no podemos cambiarlas.
- Siempre que se crea una variable de cadena, se crea una nueva instancia.

Creación de String

Usando Literal

```
String nombre = "Juan";
```

Usando `new`

```
String s = new String();
```

Métodos sobre String

La clase `String` que implementa la interfaz `CharSequence` define una serie de métodos para tareas de manipulación de cadenas. La lista de los métodos de cadena más utilizados es.:

Método	Descripción
`toLowerCase()`	Convierte la cadena a minúsculas
`toUpperCase()`	Convierte la cadena a mayúsculas
`replace('x' , 'y')`	Reemplaza todas las apariciones de 'x' con 'y'
`trim()`	Elimina los espacios en blanco al principio y al final
`equals()`	Devuelve "true" si las cadenas son iguales
`equalsIgnoreCase()`	Devuelve "true" si las cadenas son iguales, independientemente de mayúsculas o minúsculas
`length()`	Devuelve la longitud de la cadena
`CharAt(n)`	Devuelve el enésimo carácter de la cadena

compareTo()	Devuelve: negativo si la cadena 1 < la cadena 2 positivo si la cadena 1 > la cadena 2 cero si la cadena 1 = la cadena 2
concat()	Concatena dos cadenas
substring(n)	Devuelve una subcadena desde el carácter n hasta el final
substring(n,m)	Devuelve una subcadena entre los caracteres n y m.
toString()	Crea la representación de cadena del objeto
indexOf('x')	Devuelve la posición de la primera aparición de x en la cadena.
indexOf('x',n)	Devuelve la posición aparición de x después de la posición n
ValueOf (Variable)	Convierte el valor del parámetro en una representación de cadena

Programa para mostrar la ordenación de cadenas:

```
class PruebaOrdenarStrings {
    static String cadenas[] = {
            "En", "un", "lugar", "de", "la", "Mancha", "cuyo",
            "nombre", "no", "quiero", "acordarme"};
    public static void main(String args[]) {
        for(int j = 0; j < cadenas.length; j++) {
            for(int i = j + 1; i < cadenas.length; i++) {
                if(cadenas [i].compareTo(cadenas[j]) < 0) {
                    String temp = cadenas[j];
                    cadenas[j] = cadenas[i];
                    cadenas[i] = temp;
                }
            }
            System.out.println(cadenas[j]);
        }
    }
}
```

String Buffer y String Builder

- Para cadenas mutables, podemos usar las clases `StringBuilder` y `StringBuffer` que también implementan la interfaz `CharSequence`.
- Estas clases representan una interfaz de caracteres que se puede hacer crecer y escribir. Crecen automáticamente para dejar espacio para las adiciones y, a menudo, tienen más caracteres preasignados de los que realmente se necesitan, para permitir el crecimiento.

Diferencia entre `length()` y `capacity()`

- `length()`: Para encontrar la longitud de `StringBuffer`
- `capacity()`: Para encontrar la capacidad total asignada

```
/* StringBuffer length() vs. capacity() */
class PruebaStringBuffer {
    public static void main(String args[]) {
        StringBuffer sb = new StringBuffer("Hola");
        System.out.println("buffer = " + sb);
        System.out.println("length() = " + sb.length());
        System.out.println("capacity() = " + sb.capacity());
    }
}
```

String Builder	String Buffer
No sincronizado, por lo tanto eficiente.	Sincronizado
Los subprocesos utilizan multiproceso.	Hilo seguro

Manejo de Excepciones, en Java

La excepción es una condición de anomalía o error que se debe a un error en tiempo de ejecución en el programa, si este objeto excepción lanzada por condición de error no se detecta y gestiona adecuadamente, el intérprete mostrará un mensaje de error. Si queremos evitar esto y queremos que el programa continúe, deberíamos intentar detectar las excepciones. Esta tarea se conoce como manejo de excepciones.

Excepciones más comunes de Java :

Exception Type	Causa de excepción
ArithmeticException	Causado por errores matemáticos
ArrayIndexOutOfBoundException	Causado por índices de matriz incorrectos
ArrayStoreException	Causado cuando un programa intenta almacenar tipos de datos incorrectos en un array
FileNotFoundException	Causado por el intento de acceder a un archivo inexistente
IOException	Causado por fallas generales de E/S.
NullPointerException	Causado por hacer referencia a un objeto *Null*.
NumberFormatException	Causado cuando falla una conversión entre cadenas y número.
OutOfMemoryException	Causado cuando no hay suficiente memoria para asignar
StringIndexOutOfBoundException	Causado cuando un programa intenta acceder a una posición de carácter inexistente en una cadena.

Las excepciones en Java pueden ser de dos tipos:

Checked Exceptions

- Manejado explícitamente en el código mismo con la ayuda del bloque `try-catch`.
- Extendido de la clase `java.lang.Exception`

Unchecked Exceptions

- No se maneja esencialmente en el código del programa, sino que JVM maneja tales excepciones.
- Extendido de la clase `java.lang.RuntimeException`

Try - Catch

La palabra clave **`try`** se usa para prefijar un bloque de código que probablemente cause una condición de error y **`throw`** (lanza) una excepción. Un bloque **`catch`**(captura), definido por la palabra clave `catch`, captura la excepción lanzada dentro del bloque `try` y la maneja de manera apropiada.

Un código puede tener más de una declaración de `catch` en el bloque de captura, cuando se genera una excepción en el bloque de `try`.

Uso de `Try - Catch` para el manejo de excepciones

```
class Error {
    public static void main(String args[]) {
        int a [] = {5, 5, 0};
        int b = 5;
        try {
            int x = a[2] / (b - a[1]);
        }
        catch(ArithmeticException e) {
            System.out.println("División por cero");
        }
        catch(ArrayIndexOutOfBoundsException e) {
            System.out.println("Error de índice de Array");
        }
        catch(ArrayStoreException e) {
            System.out.println("Tipo de datos incorrecto ");
        }
        int y = a[1] / a[0];
        System.out.println("y = " + y);
    }
}
```

Finally

La declaración **`finally`** se utiliza para manejar excepciones que no son detectadas por ninguna de las declaraciones de `catch` anteriores. Un bloque `finally` garantiza ejecutarse, independientemente de si se lanza o no una excepción.

```
class Error {
    public static void main(String args[]) {
        int a [] = {5, 5, 0};
        int b = 5;
        try {
            int x = a[2] / (b - a[1]);
        }
        catch(ArithmeticException e) {
            System.out.println("División por cero");
        }
        catch(ArrayIndexOutOfBoundsException e) {
            System.out.println("Error de índice de Array");
        }
        catch(ArrayStoreException e) {
            System.out.println("Tipo de datos incorrecto ");
        }
        finally
        {
            int y = a[1] / a[0];
            System.out.println("y = " + y);
        }
    }
}
```

Crear su propia excepción

Las propias excepciones se pueden definir usando la palabra **throw**.

```
import java.lang.Exception;
class MiExcepcion extends Exception {
    MiExcepcion(String message) {
        super(message);
    }
}

class PruebaMiExcepcion {
    public static void main(String args[]) {
        int x = 5 , y = 1000;
        try {
            float z = (float) x / (float) y ;
            if( z < 0.01)
            {
                throw new MiExcepcion("Número muy pequeño");
            }
        }
        catch (MiExcepcion e) {
            System.out.println("Captura mi excepción ");
            System.out.println(e.getMessage());
        }
        finally {
            System.out.println("Siempre llego a finally");
        }
    }
}
```

Gestión de archivos, en Java

El almacenamiento de datos en variables y *arrays* plantea los siguientes problemas:

- **Almacenamiento temporal**: los datos se pierden cuando la variable se sale de alcance o cuando se termina el programa.
- **Gran cantidad de datos**: difíciles de manejar

Estos problemas se pueden resolver almacenando datos en dispositivos secundarios utilizando el concepto de archivos.

Un archivo es una colección de registros relacionados almacenados en un área particular del disco denominada archivo o fichero. Los archivos almacenan y administran datos mediante el concepto de manejo de archivos.

El procesamiento de archivos incluye:

- Creando de archivos
- Actualización de archivos
- Manipulación de datos

Java proporciona muchas funciones en la gestión de archivos como:

- La lectura/escritura de datos se puede realizar a nivel de bytes o en caracteres o campos, según el requisito.
- También proporciona la capacidad de leer/escribir objetos directamente.

Streams

Java utiliza el concepto de flujos(`stream`) para representar una secuencia ordenada de datos, que es una ruta por la que fluyen los datos. Tiene un origen y un destino.

Los flujos se clasifican en dos tipos básicos:

- **Input Stream:** Que extrae, es decir, lee datos del archivo de origen y los envía al programa.
- **Output Stream:** Que toma los datos del programa y los envía, es decir, escribe al destino.

Clases Stream

Están contenidos en el paquete java.lang.io.

Están categorizados en dos grupos

- **Byte Stream:** Proporciona soporte para manejar la operación de E/S en bytes.
- **Character Stream:** Proporciona soporte para administrar operaciones de E/S de caracteres.

Clase Bytes Stream

Diseñada para proporcionar funcionalidad para crear y manipular flujos y archivos para lectura / escritura de bytes.

Dado que los flujos son unidireccionales, existen dos tipos de clases de flujo de bytes:

- `Input Stream`
- `Output Stream`

Clase Input Stream

Se utilizan para leer bytes de 8 bits que incluyen una superclase conocida como `InputStream`. InputStream es una clase abstracta y define los métodos para funciones de entrada como:

Métodos	**Descripción**
`read( )`	Lee un byte del flujo de entrada
`read(byte b[ ])`	Lee un array de bytes en b
`read(byte b[ ], int n, int m)`	Lee m bytes en b a partir del enésimo byte de b
`available( )`	Indica el número de bytes disponibles en la entrada.
`skip(n)`	Omite n bytes del flujo de entrada
`reset ( )`	Vuelve al principio de la secuencia.
`close ( )`	Cierra el flujo de entrada

Clase Output Stream

Estas clases se derivan de la clase base `OutputStream.` OutputStream es una clase abstracta y define los métodos para funciones de salida como :

Métodos	**Descripción**
`write( )`	Escribe un byte en el flujo de salida.
`write(byte b[])`	Escribe todos los bytes del array b en el flujo de salida.
`write(byte b[], int n, int m)`	Escribe m bytes del array b a partir del enésimo byte
`close( )`	Cierra el flujo de salida
`flush( )`	Limpia el flujo de salida

Leer/Escribir Bytes

Dos subclases comunes que se utilizan son `FileInputStream` y `FileOutputStream` que manejan bytes de 8 bits.

1. `FileOutputStream` se utiliza para escribir bytes en un archivo como se muestra a continuación:

Escribiendo bytes en un archivo

```
import java.io.*;
class escribirBytes {
    public static void main(String args[])
    {
        byte ciudades [] = {
                'M','A','D','R','I','D', '\n',
                'T','O','L','E','D','O','\n',
                'J','A','E','N','\n'};
        //Crear flujo de archivo de salida
        FileOutputStream outfile = null;
        try {
        //conectar el flujo de archivos a "ciudades.txt"
            outfile = new FileOutputStream("ciudades.txt");
            //Escribir datos en el stream
            outfile.write(ciudades);
            outfile.close();
        }
        catch(IOException ioe) {
            System.out.println(ioe);
            System.exit(-1);
        }
    }
}
```

2. `FileIntputStream` se utiliza para leer bytes de un archivo como se muestra a continuación:

Leyendo bytes de un archivo

```
import java.io.*;
class LeerBytes {
    public static void main(String args[]) {
        //Crear un flujo de archivo de entrada
        FileInputStream infile = null;
        int b;
        try {
            //conecte el flujo de datos al archivo requerido
            infile = new FileInputStream(args [ 0 ]);
            //Leer y mostrar
            while( (b = infile.read ( ) ) !=-1)
            {
                System.out.print((char) b );
            }
            infile.close();
        }
        catch(IOException ioe) {
            System.out.println(ioe);
            System.exit(-1);
        }
    }
}
```

Clase Character Stream

1. **Clases Reader Stream**
   - Diseñada para leer caracteres de los archivos.
   - La clase `Reader` es la clase base para todas las demás clases.
   - Estas clases son similares a las clases input stream excepto en su unidad fundamental de información, mientras que el reader stream usa caracteres.

2. **Clases Writer Stream**
   - Realiza todas las operaciones de salida en archivos.
   - Escribe caracteres
   - La clase `Writer` es una clase abstracta, es la clase base, y tiene métodos idénticos a los de `OutputStream`.

Lectura / Escritura de caracteres

Las dos subclases de las clases `Reader` y `Writer` para manejar caracteres en archivos son `FileReader` y `FileWriter`.

Copiar caracteres de un archivo a otro

```
import java.io.*;
class CopiarCharacteres {
    public static void main (String args[]) {
        //Declarar y crear archivos de entrada y salida
        File inFile = new File("entrada.dat");
        File outFile = new File("salida.dat");
        //crea entradas de flujo de archivos
        FileReader entrada = null;
        //crea salidas de flujo de archivos
        FileWriter salida = null;
        try {
            //se abre la entrada
            entrada = new FileReader(inFile);
            //se abre la salida
            salida = new FileWriter(outFile);
            //Lee y escribe
            int caracter;
            while((caracter = entrada.read( )) != -1)
            {
                salida.write(caracter);
            }
        }
        catch(IOException e) {
            System.out.println(e);
            System.exit(-1);
        }
        finally {
            try {
                salida.close();
                salida.close();
            }
            catch (IOException e)
            {}
        }
    }
}
```

El tiempo, en Java

Desde Java 8 se ha introducido una nueva API de gestión de fechas y horas en el paquete `java.time`.

LocalDate

La clase Java `LocalDate` es una clase inmutable que representa la fecha con un formato predeterminado de `aaaa-MM-dd`.

Los principales métodos son:

Método	Descripción
`LocalDateTime atTime(int hour, int minute)`	Combine esta fecha con una hora para crear un `LocalDateTime`.
`int compareTo(ChronoLocalDate other)`	Compara esta fecha con otra fecha.
`boolean equals(Object obj)`	compruebe si esta fecha es igual a otra fecha.
`String format(DateTimeFormatter formatter)`	Formatea esta fecha utilizando el formateador especificado.
`int get(TemporalField field)`	obtiene el valor del campo especificado a partir de esta fecha como `int`.
`boolean isLeapYear()`	Compruebe si el año es bisiesto, de acuerdo con las reglas del sistema de calendario ISO.
`LocalDate minusDays(long daysToSubtract)`	Devuelve una copia de este `LocalDate` con la cantidad de días especificada restada.
`LocalDate minusMonths(long monthsToSubtract)`	Devuelve una copia de este `LocalDate` con el número especificado de meses restado.
`static LocalDate now()`	Obtiene la fecha actual del reloj del sistema en la zona horaria predeterminada.
`LocalDate plusDays(long daysToAdd)`	Devuelve una copia de este `LocalDate` con el número especificado de días agregados.
`LocalDate plusMonths(long monthsToAdd)`	Devuelve una copia de este `LocalDate` con el número especificado de meses añadidos.

LocalDateTime

La clase `LocalDateTime` es un objeto de fecha y hora inmutable que representa una fecha y hora, con el formato predeterminado `aaaa-MM-dd-HH-mm-ss.zzz`.

Los principales métodos son:

Método	Descripción
`String format(DateTimeFormatter formatter)`	Formatear esta fecha y hora utilizando el formateador especificado.
`int get(TemporalField field)`	Obtener el valor del campo especificado de esta fecha y hora como un `int`.

`LocalDateTime minusDays(long days)`	devuelve una copia de este `LocalDateTime` con la cantidad de días especificada restada.
`static LocalDateTime now()`	Obtener la fecha y hora actual del reloj del sistema.
`static LocalDateTime of(LocalDate date, LocalTime time)`	Obtener una instancia de `LocalDateTime` a partir de una fecha y hora.
`LocalDateTime plusDays(long days)`	devolver una copia de este `LocalDateTime` con el número especificado de días sumados.
`boolean equals(Object obj)`	Compruebe si esta fecha-hora es igual a otra fecha-hora.

Colecciones, en Java

El framework `Collection` contenido en el paquete `java.util` define un conjunto de interfaces y sus implementaciones para manipular colecciones, que sirven como contenedor para un grupo de objetos.

Interfaces

El framework `Collection` contiene muchas interfaces como Colección, Mapa e Iterador.

Las interfaces y su descripción se mencionan a continuación:

Interface	**Descripción**
`Collection`	Colección de elementos
`List (extends Collection)`	Secuencia de elementos
`Queue (extends Collection)`	Tipo especial de lista
`Set (extends Collection)`	Colección de elementos únicos
`SortedSet (extends Set)`	Colección ordenada de elementos únicos
`Map`	Colección de pares de claves y valores, que deben ser únicos
`SortedMap (extends Map)`	Colección ordenada de pares clave-valor
`Iterator`	Objeto utilizado para recorrer una colección
`List (extends Iterator)`	Objeto utilizado para recorrer una secuencia

Clases

Las clases disponibles en el framework `Collection` implementan la interfaz y las subinterfaces de la colección. También implementan interfaces `Map` e `Iterator`.

Las clases y sus interfaces correspondientes se enumeran a continuación:

Clase	**Interface**
`AbstractCollection`	Collection
`AbstarctList`	List
`Abstract`	Queue
`AbstractSequentialList`	List
`LinkedList`	List

`ArrayList`	List, Cloneable y Serializable
`AbstractSet`	Set
`EnumSet`	Set
`HashSet`	Set
`PriorityQueue`	Queue
`TreeSet`	Set
`Vector`	List, Cloneable y Serializable
`Stack`	List, Cloneable y Serializable
`Hashtable`	Map, Cloneable y Serializable

Implementación de *ArrayList*

Algunos métodos, más usados, que proporciona *`ArrayList`* son:

Método	Descripción
`size()`	Devuelve el número de elementos (`int`)
`add(X)`	Añade el objeto X al final. Devuelve `true`.
`add(posición, X)`	Inserta el objeto X en la posición indicada.
`get(posicion)`	Devuelve el elemento que está en la posición indicada.
`remove(posicion)`	Elimina el elemento que se encuentra en la posición indicada. Devuelve el elemento eliminado.
`remove(X)`	Elimina la primera ocurrencia del objeto X. Devuelve `true` si el elemento está en la lista.
`clear()`	Elimina todos los elementos.
`set(posición, X)`	Sustituye el elemento que se encuentra en la posición indicada por el objeto X. Devuelve el elemento sustituido.
`contains(X)`	Comprueba si la colección contiene al objeto X. Devuelve `true` o `false`.
`indexOf(X)`	Devuelve la posición del objeto X. Si no existe devuelve -1

Uso de algunos métodos de la clase *ArrayList*

```
import java.util.*;
class Num {
    public static void main(String args[])      {
        ArrayList num = new ArrayList ();
        num.add(9);
        num.add(12);
        num.add(10);
        num.add(16);
        num.add(6);
        num.add(8);
        num.add(56);
        // Impresión del array list
        System.out.println("Elementos : ");
        num.forEach((s) -> System.out.println(s));
        // Devuelve el tamaño
        System.out.println("El tamaño es: ");
        num.size();
```

```
            // Devuelve un elemento específico
            int n = (Integer) num.get(2);
            System.out.println(n);
            // eliminar un elemento específico
            num.remove(4);
            // Impresión del array list
            System.out.println("Elementos : ");
            num.forEach((s) -> System.out.println(s));
        }
    }
```

Implementación *LinkedList* - lista enlazada

```
    import java.util.*;
    class PruebaLL {
        public static void main(String args[]) {
            // Creacion del objeto
            LinkedList<String> ll = new LinkedList<String>();
            // Añadir elementos a la lista enlazada
            ll.add("A");
            ll.add("B");
            ll.addLast("C");
            ll.addFirst("D");
            ll.add(2, "E");
            System.out.println(ll);
            ll.remove("B");
            ll.remove(3);
            ll.removeFirst();
            ll.removeLast();
            System.out.println(ll);
        }
    }
```

Implementación de *HashSet*

```
  import java.util.*;
  class HashSetExample {
    public static void main(String args[]) {
        HashSet hs = new HashSet();
        hs.add("D");
        hs.add("W");
        hs.add("G");
        hs.add("L");
        hs.add("Y");
        System.out.println("Los elementos del conjunto son :" + hs);
    }
  }
```

Implementación de *TreeSet*

```
import java.util.*;
class EjemploTreeSet {
   public static void main(String args[]) {
     TreeSet ts = new TreeSet();
     ts.add("D");
     ts.add("W");
     ts.add("G");
     ts.add("L");
     ts.add("Y");
     System.out.println("Los elementos del árbol son :" + ts);
  }
}
```

Implementación de la clase *Vector*

```
import java.util.*;
class EjemploVector
{
    public static void main(String args[])
    {
        Vector frutas = new Vector ();
        frutas.add("Manzana");
        frutas.add("Naranja");
        frutas.add("Uvas");
        frutas.add("Peras");
        Iterator it = frutas.iterator();
        while (it.hasNext())
        {
            System.out.println(it.next());
        }
    }
}
```

Implementación de la clase *Stack*

```
import java.util.*;
class EjemploStack {
    public static void main (String args[]) {
        Stack st = new Stack ();
        st.push("Java");
        st.push("Clases");
        st.push("Objectos");
        st.push("Multithreading");
        st.push("Programacion");
        System.out.println("Elementos en el Stack : " + st);
        System.out.println("Elementos en la parte superior del
                            Stack : " + st.peek());
        System.out.println("Elementos salieron del Stack : " +
                            st.pop());
        System.out.println("Elementos en el Stack despues de pop
                             : " + st);
        System.out.println("El resultado de la búsqueda  : " +
                            st.search ("r e"));
    }
}
```

Implementación de la clase *HashTable*

```
import java.util.*;
class HashTableExample
{
    public static void main (String args[])
    {
        Hashtable ht = new Hashtable();
        ht.put("Item 1","Manzana");
        ht.put("Item 2","Naranja");
        ht.put("Item 3","Uvas");
        ht.put("Item 4","Piña");
        ht.put("Item 5","Kiwi");
        Enumeration e = ht.keys();
        while(e.hasMoreElements())
        {
            String str = (String) e.nextElement();
            System.out.println(ht.get(str));
        }
    }
}
```

Algunas ejercicios comunes sobre codificación, en Java

Introduzca radio y el diámetro e impresión, perímetro y área

```
import java.util.Scanner;
class Circulo {
    public static void main (String args []) {
        double rad, dia, peri, area ;
        System.out.println("Introduzca el radio del círculo:
");
        Scanner s = new Scanner (System.in);
        rad = s.nextDouble();
        dia = 2 * rad;
        peri = 2 * Math.PI * rad;
        area = Math.PI * rad * rad;
        System.out.printf("El diámetro es    : %.2f \n", dia);
        System.out.printf("El perímetro es  : %.2f \n", peri);
        System.out.printf("El área es       : %.2f \n", area);
    }
}
```

Imprime todos los números pares entre x e y.

```
import java.util.Scanner;
class Pares {
    public static void main (String args[]) {
        int x,y;
        Scanner s = new Scanner (System.in);
        System.out.println("Ingrese los valores x , y : ");
        x = s.nextInt();
        y = s.nextInt();
        System.out.println("NÚMEROS PARES EN EL RANGO DADO >>");
        int cont = x;
        while(cont <= y) {
            if(cont % 2 == 0){
                System.out.println(cont);
            }
            cont ++;
        }
    }
}
```

Comprobar si el número dado es primo

```
import java.util.Scanner;
class Primo {
    public static void main (String args[]) {
        double numero;
        int n;
        boolean esPrimo = true;
        Scanner s = new Scanner(System.in);
        System.out.println("Ingrese el número a verificar  :");
        numero = s.nextDouble();
        n = (int) Math.sqrt(numero);
        for(int i=2; i <= n ;i++) {
            if(numero % i == 0)
            {
                esPrimo = false;
            }
            else
            {
                esPrimo = true;
            }
        }
        if(esPrimo) {
           System.out.println("* EL NUMBERO ES PRIMO !! *");
        }
        else {
           System.out.println("* EL NUMBERO NO ES PRIMO !! *");
        }
    }
}
```

Compruebe si el número introducido es Palíndromo

```
import java.util.Scanner;
class Palindromo
{
    public static void main(String args[])
    {
        int num, reverso = 0, modulo;
        Scanner s = new Scanner(System.in);
        System.out.println("Ingrese un número para verificar
                            Palindromo : ");
        num = s.nextInt();
        int numero = num;
        while(num!=0)
        {
            //System.out.println(" número de entrada  = "+num);
            modulo = num % 10;
            //System.out.println(" modulo = "+modulo);
            reverso =(reverso * 10 ) + modulo;
            //System.out.println(" reverso = "+reverso);
            num = num / 10;
            //System.out.println(" nuevo num = "+num);
        }
        //System.out.println(" reverso = "+reverso);
        if(reverso == numero)
        {
            System.out.println(" ** PALINDROMO !!! ** ");
        }
        else
        {
            System.out.println(" ** NO ES PALINDROME !!! ** ");
        }
    }
}
```

Impresión el patrón

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

```
class Estrellas {
    public static void main(String args[]) {
        for(int i=0;i<=5;i++) {
            for(int j=0;j<i;j++) {
                System.out.print(" * ");
            }
            System.out.println();
        }
        System.out.println();
    }
}
```