Programación de Computadoras

Dr. Félix Calderon Solorio

25 de febrero de 2021

Índice general

Introducción 1

1.1. Propiedades deseables: Reusabilidad y extensibilidad, entre otras 1
1.2. Programación orientada a objetos 2
1.3. Ingeniería de Software. Ventajas de la Programación Orientada a Objetos . 2
1.4. El lenguaje de programación Java 3
1.5. Ciclo de desarrollo de programas en Java 4

Datos y Expresiones 7

2.1. Tipos de datos en java (primitivos, Clases, Interfaces, Enumeraciones) . . . 7
2.1.1. Ejemplo 8
2.2. Variables y asignación 9
2.3. Cadenas de caracteres 11
2.4. Expresiones 12
2.4.1. Operadores Aritméticos 12
2.4.2. Otras operaciones matemáticas menos habituales 15
2.4.3. Operadores de Asignación 16
2.4.4. Operadores Relacionales 17
2.4.5. Operadores lógicos 17
2.5. Conversión de datos 18
2.5.1. Ejemplo 19
2.6. Clases envoltorio o wrapper 19

Uso de Clases y Objetos 21

3.1. La API de Java 21
3.2. Creación de objetos 22
3.2.1. Ejemplo de manejo de Números Complejos 24
3.2.2. Ejemplo de una Cuenta Bancaria 28
3.3. La clase String 30
3.3.1. Operaciones básicas, comunes a String y StringBuffer 31
3.3.2. Métodos de la clase String 31

3.4. Paquetes 33
3.5. La clase Random 35
3.5.1. Ejemplo 1 36
3.5.2. Ejemplo 2 37
3.6. La clase Math 37
3.7. Formateo de Salida 38

Control de Flujo **41**
4.1. Sentencia if 41
4.1.1. Ejemplo Mayor Menor 42
4.1.2. Ejemplo Triángulo 43
4.1.3. Ejemplo Cuadrática 45
4.1.4. Ejemplo Caída Libre 48
4.1.5. Ejemplo Suma de dígitos 49
4.1.6. Operador Condicional 50
4.2. Comparando datos 52
4.2.1. Sobre escritura del método equal() 54
4.3. Sentencia switch 56
4.3.1. Ejemplo ultimo dígito 56
4.3.2. Ejemplo rango letras 58
4.4. Sentencia while 59
4.4.1. while 59
4.4.2. do-while 60
4.5. Sentencias do y for 60
4.5.1. Ejemplo Sumatoria 62
4.5.2. Ejemplo comparación for, while, do-while 63
4.5.3. Ejemplo Combinaciones 64
4.5.4. Ejemplo función 65
4.5.5. Ejemplo triángulo impreso 66
4.6. Iteradores 66
4.6.1. Ejemplo Agenda 68
4.7. Recursividad 72
4.7.1. Torres de Hanoi 73
4.7.2. Método de Bisecciones 74
4.8. Assertions 76
4.9. Introducción a Excepciones 77

Modelado de Objetos **81**
5.1. Modelos de especificación del Dominio: La técnica CRC 81
5.2. Diagrama de Clases (modelo conceptual) 83
5.3. Ejemplos prácticos 86

5.3.1. Documentación . . . 93
5.4. Utilizando la Técnica CRC . . . 94
5.4.1. Ejemplo Escuela . . . 94
5.4.2. Ejemplo Cajero . . . 100

Diseño de Clases en Java **105**
6.1. Repaso de Clases y Objetos . . . 105
6.2. Anatomía de una Clase (java y UML) . . . 105
6.3. Encapsulación . . . 106
6.4. Anatomía de un método . . . 108
6.5. Revisión de constructores . . . 109
6.6. Miembros estáticos . . . 114
6.7. Relaciones entre clases . . . 115
6.8. Interfaces . . . 115
6.9. Herencia . . . 117
6.9.1. Subclases . . . 118
6.9.2. Sobreescritura de métodos . . . 118
6.9.3. Visibilidad . . . 121
6.9.4. ejemplo . . . 121
6.10. Polimorfismo . . . 122
6.10.1. Polimorfismo usando herencia . . . 123
6.10.2. Polimorfismo usando Interfaces . . . 124
6.11. Revisión de Excepciones . . . 124
6.11.1. Ejemplo 1 . . . 124
6.11.2. Ejemplo 2 . . . 129
6.11.3. Ejemplo 3 . . . 131

Uso de Collections y Generics **139**
7.1. Uso de == . . . 139
7.2. Colecciones . . . 140
7.2.1. List . . . 141
7.2.2. Map . . . 146
7.2.3. Collections . . . 148
7.3. Generics . . . 149
7.4. Uso de comparadores . . . 156
7.4.1. Ejemplo de una lista de Empleados . . . 158
7.4.2. Ejemplo de Una agenda . . . 160

Concurrencia **165**
8.1. Creación de hilos . . . 165
8.2. Estados de un hilo . . . 166

8.2.1. Nuevo Thread . . . 166
8.2.2. Ejecutable . . . 167
8.2.3. Parado . . . 167
8.2.4. Muerto . . . 168
8.2.5. El método isAlive() . . . 169
8.2.6. Ejemplo Contadores . . . 169
8.2.7. Ejemplo Cánicas . . . 172
8.3. Solución de problemas de acceso concurrente . . . 175
8.4. Uso de wait . . . 177

Java I/O **181**
9.1. La clase File . . . 181
9.2. La clase RandomAccessFile . . . 183
9.2.1. Ejemplos de operaciones con ficheros de acceso aleatorio . . . 184
9.3. Comandos I/O . . . 186
9.4. Serialización . . . 186
9.5. Uso de la clase java.util.Locale . . . 186
9.6. Uso de expresiones regulares . . . 186

Tareas **187**
10.1. Tarea 1 . . . 187
10.2. Tarea 2 . . . 187
10.3. Tarea 3 . . . 187
10.4. Tarea 4 . . . 188
10.5. Tarea 5 . . . 188
10.6. Tarea 6 . . . 188
10.7. Tarea 7 . . . 188
10.8. Tarea 8 . . . 188
10.9. Tarea . . . 189

Introducción

Java es un lenguaje de programación orientado a objetos desarrollado por Sun Microsystems a principio de los años 90's.

En Diciembre de 1950 Patrick Naughton, ingeniero de Sun Microsystems, reclutó a varios colegas entre ellos James Gosling y Mike Sheridan para trabajar sobre un nuevo proyecto conocido como El proyecto verde. Con la ayuda de otros ingenieros, empezaron a trabajar en una pequeña oficina en Sand Hill Road en Menlo Park, California. Y así interrumpió todas las comunicaciones regulares con Sun y trabajó sin descanso durante 18 meses.

Intentaban desarrollar una nueva tecnología para programar la siguiente generación de dispositivos inteligentes, en los que Sun veía un campo nuevo a explorar. Crear un lenguaje de programación fácil de aprender y de usar. En un principio se consideraba C++ como lenguaje a utilizar, pero tanto Gosling como Bill Joy lo encontraron inadecuado. Gosling intentó primero extender y modificar C++ resultando el lenguaje C++ ++ - (++ - porque se añadían y eliminaban características a C++), pero lo abandonó para crear un nuevo lenguaje desde cero al que llamo Oak (roble en inglés, según la versión mas aceptada, por el roble que veía a través de la ventana de su despacho).

El resultado fue un lenguaje que tenía similitudes con C, C++ y Objetive C y que no estaba ligado a un tipo de CPU concreta. Mas tarde, se cambiaría el nombre de Oak a Java, por cuestiones de propiedad intelectual, al existir ya un lenguaje con el nombre de Oak. Se supone que le pusieron ese nombre mientras tomaban café (Java es nombre de un tipo de café, originario de Asia), aunque otros afirman que el nombre deriva de las siglas de James Gosling, Arthur Van Hoff, y Andy Bechtolsheim.

1.1. Propiedades deseables: Reusabilidad y extensibilidad, entre otras

Reusabilidad. Cuando hemos diseñado adecuadamente las clases, se pueden usar en distintas partes del programa y en numerosos proyectos.

Mantenibilidad. Debido a las sencillez para abstraer el problema, los programas orientados a objetos son más sencillos de leer y comprender, pues nos permiten ocultar detalles de implementación dejando visibles sólo aquellos detalles más relevantes.

Modificabilidad. La facilidad de añadir, suprimir o modificar nuevos objetos nos permite hacer modificaciones de una forma muy sencilla.

Fiabilidad. Al dividir el problema en partes más pequeñas podemos probarlas de manera independiente y aislar mucho más fácilmente los posibles errores que puedan surgir.

1.2. Programación orientada a objetos

La programación orientada a objetos o POO (OOP según sus siglas en inglés) es un paradigma de programación que usa los objetos en sus interacciones, para diseñar aplicaciones y programas informáticos. Está basado en varias técnicas, incluyendo herencia, cohesión, abstracción, polimorfismo, acoplamiento y encapsulamiento. Su uso se popularizó a principios de la década de los años 1990. En la actualidad, existe una gran variedad de lenguajes de programación que soportan la orientación a objetos.

Un paradigma de programación es una propuesta tecnológica que es adoptada por una comunidad de programadores cuyo núcleo central es incuestionable en cuanto a que únicamente trata de resolver uno o varios problemas claramente delimitados. Es un estilo de programación empleado. La resolución de estos problemas debe suponer consecuentemente un avance significativo en al menos un parámetro que afecte a la ingeniería de software. Tiene una estrecha relación con la formalización de determinados lenguajes en su momento de definición. Un paradigma de programación está delimitado en el tiempo en cuanto a aceptación y uso ya que nuevos paradigmas aportan nuevas o mejores soluciones que la sustituyen parcial o totalmente.

Un objeto es una unidad dentro de un programa de computadora que consta de un estado y de un comportamiento, que a su vez constan respectivamente de datos almacenados y de tareas realizables durante el tiempo de ejecución.

1.3. Ingeniería de Software. Ventajas de la Programación Orientada a Objetos

Las ventajas más importantes de la programación orientada a objetos: son Reusabilidad, Mantenibilidad, Modificabilidad y Fiabilidad.

La programación orientada a objetos presenta también algunas desventajas como pueden ser:

Cambio en la forma de pensar de la programación tradicional a la orientada a objetos.

La ejecución de programas orientados a objetos es más lenta.

La necesidad de utilizar bibliotecas de clases obliga a su aprendizaje y entrenamiento.

1.4. El lenguaje de programación Java

Las características principales que nos ofrece Java son:

1.- Lenguaje Simple:

Se lo conoce como lenguaje simple porque viene de la misma estructura de c y c++; ya que c++ fue un referente para la creación de java por eso utiliza determinadas características de c++ y se han eliminado otras.

2.- Orientado a Objeto:

Toda la programación en java en su mayoría está orientada a objeto, ya que al estar agrupados en estructuras en estructuras encapsuladas es más fácil su manipulación.

3.- Distribuido:

Permite abrir sockets, establecer y aceptar conexiones con los servidores o clientes remotos; facilita la creación de aplicaciones distribuidas ya que proporciona una colección de clases para aplicaciones en red.

4.- Robusto:

Es altamente fiable en comparación con c, se han eliminado muchas características con la aritmética de punteros, proporciona numerosas comprobaciones en compilación y en tiempo de ejecución.

5.- Seguro:

La seguridad es una característica muy importante en java ya que se han implementado barreras des eguridad en el lenguaje y en el sistema de ejecución de tiempo real.

6.- Indiferente a la arquitectura:

Java es compatible con los más variados entornos de red, cualquiera sean estos desde Windows 95, Unix aWindows Nt y Mac, para poder trabajar con diferentes sistemas operativos.

Java es muy versátil ya que utiliza byte-codes que es un formato intermedio que sirve para transportar el código eficientemente o de diferentes plataformas (Hardware - Software).

7.- Portable:

Por ser indiferente a la arquitectura sobre la cual está trabajando, esto hace que su portabilidad sea muy eficiente, sus programas son iguales en cualquiera de las plataformas, yaque java especifica tamaños básicos, esto se conoce como la máquina virtual de java.

8.- Interpretado y compilado a la vez:

Java puede ser compilado e interpretado en tiempo real, ya que cuando se construye el código fuente este se transforma en una especie de código de máquina.

9.- Multihebra o Multihilos:

Java tiene una facilidad de cumplir varias funciones al mismo tiempo, gracias a su función de multi-hilos ya que por cada hilo que el programa tenga se ejecutaran en tiempo real muchas funciones al mismo tiempo.

10.- Dinámico:

El lenguaje java es muy dinámico en la fase de enlazado, sus clases solamente actuaran en medida en que sean requeridas o necesitadas con esto permitirá que los enlaces se puedan incluir incluso desde fuentes muy variadas o desde la red.

11.- Produce Applets:

En java se pueden crear aplicaciones independientes y applets.

Independientes porque se pueden comportar como cualquier programa escrito en cualquier lenguaje.

Por otra parte los applets considerados pequeños programas, tienen la capacidad de ejecutar funciones muy complejas.

12.- Alto rendimiento

Java es considerado de alto rendimiento por ser tan veloz en el momento de correr los programas y por ahorrase muchas líneas de código.

1.5. Ciclo de desarrollo de programas en Java

Planificación

La importante tarea a la hora de crear un producto de software es obtener los requisitos o el análisis de los requisitos. Los clientes suelen tener una idea más bien abstracta del resultado final, pero no sobre las funciones que debería cumplir el software.

Una vez que se hayan recopilado los requisitos del cliente, se debe realizar un análisis del ámbito del desarrollo. Este documento se conoce como especificación funcional.

Implementación, pruebas y documentación

La implementación es parte del proceso en el que los ingenieros de software programan el código para el proyecto.

Las pruebas de software son parte esencial del proceso de desarrollo del software. Esta parte del proceso tiene la función de detectar los errores de software lo antes posible.

La documentación del diseño interno del software con el objetivo de facilitar su mejora y su mantenimiento se realiza a lo largo del proyecto. Esto puede incluir la documentación de un API, tanto interior como exterior.

Despliegue y mantenimiento

El despliegue comienza cuando el código ha sido suficientemente probado, ha sido aprobado para su liberación y ha sido distribuido en el entorno de producción.

Entrenamiento y soporte para el software es de suma importancia y algo que muchos desarrolladores de software descuidan. Los usuarios, por naturaleza, se oponen al cambio porque conlleva una cierta inseguridad, es por ello que es fundamental instruir de forma adecuada a los futuros usuarios del software.

El mantenimiento y mejora del software de un software con problemas recientemente desplegado puede requerir más tiempo que el desarrollo inicial del software. Es posible que haya que incorporar código que no se ajusta al diseño original con el objetivo de solucionar un problema o ampliar la funcionalidad para un cliente. Si los costes de mantenimiento son muy elevados puede que sea oportuno rediseñar el sistema para poder contener los costes de mantenimiento.

Datos y Expresiones

2.1. Tipos de datos en java (primitivos, Clases, Interfaces, Enumeraciones)

En nuestro primer ejemplo escribíamos un texto en pantalla, pero este texto estaba prefijado dentro de nuestro programa.

Esto no es lo habitual: normalmente los datos que maneje nuestro programa serán el resultado de alguna operación matemática o de cualquier otro tipo. Por eso, necesitaremos un espacio en el que ir almacenando valores temporales y resultados.

En casi cualquier lenguaje de programación podremos reservar esos espacios, y asignarles un nombre con el que acceder a ellos. Esto es lo que se conoce como variables.

Por ejemplo, si queremos que el usuario introduzca dos números y el ordenador calcule la suma de ambos números y la muestre en pantalla, necesitaríamos el espacio para almacenar al menos esos dos números iniciales (en principio, para la suma no sería imprescindible, porque podemos mostrarla en pantalla nada más calcularla, sin almacenarla previamente en ningún sitio). Los pasos a dar serían los siguientes:

Pedir al usuario que introduzca un número. Almacenar ese valor que introduzca el usuario (por ejemplo, en un espacio de memoria al que podríamos asignar el nombre primerNumero). Pedir al usuario que introduzca otro número. Almacenar el nuevo valor (por ejemplo, en otro espacio de memoria llamado segundoNumero). Mostrar en pantalla el resultado de sumar primerNumero y segundoNumero.

Pues bien, en este programa estaríamos empleando dos variables llamadas primerNumero y segundoNumero. Cada una de ellas sería un espacio de memoria capaz de almacenar un número.

El espacio de memoria que hace falta reservar será distinto según la precisión que necesitemos para ese número, o según lo grande que pueda llegar a ser dicho número. Por eso, tenemos disponibles diferentes tipos de variables.

Por ejemplo, si vamos a manejar números sin decimales (números enteros) de como máximo 9 cifras, nos interesaría el tipo llamado int (abreviatura de integer, que es entero en inglés). Este tipo de datos consume un espacio de memoria de 4 bytes (1 Mb es cerca de 1 millón de bytes). Si no necesitamos guardar datos tan grandes (por ejemplo, si nuestros datos va a ser números inferiores a 1.000), podemos emplear el tipo de datos llamado short (entero corto).

Los tipos de datos numéricos disponibles en Java son los siguientes:

Nombre	Valor min.	Valor max.	Precisión	Ocupa
byte	-128	127	-	1 byte
short	-32,768	32,767	-	2 bytes
int	-2,147,483,648	2,147,483.647	-	4 bytes
long	-9,223,372,03,.854,775.808	9,223,372,036,854,775.807	-	8 bytes
float	1.401298E-45	3.402823E38	6-7 cifras	4 bytes
double	4.94065645841247E-324	1.79769313486232E308	14-15 cifras	8 bytes

2.1.1. Ejemplo

El siguiente ejemplo muestra el tamaño de los tipos de datos numéricos de Java. Note como se realiza una pequeña operación y el resultado como se ve afectado.

```
public class tipos {
    static public void main(String args[]) {
        byte a = 127;
        short b = 32767;
        int c =  2147483647;
        long d = 9223372036854775807l;
        float e = 3.402823E38f;
        double f = 1.7976931348623E308;

        System.out.println("byte   " + (byte)(a +1));
        System.out.println("short  " + (short)(b +1));
        System.out.println("int    " + (int)(c +1));
        System.out.println("long   " + (long)(d +1));

        System.out.println("float  " + (float)(e*1.1));
        System.out.println("double " + (double)(f*1.1));

    }
}
```

Tenemos otros dos tipos básicos de variables, que no son para datos numéricos:

char. Será una letra del alfabeto, o un dígito numérico, o un símbolo de puntuación. Ocupa 2 bytes. Sigue un estándar llamado Unicode (que a su vez engloba a otro estándar anterior llamado ASCII). boolean. se usa para evaluar condiciones, y puede tener el valor verdadero (true) o false (false). Ocupa 1 byte.

El siguiente ejemplo muestra como se declaran estos tipos de datos

```
public class otros_tipos {
    static public void main(String args[]) {
        char a = 'a';
        boolean b = true;

        System.out.println(a + " " + b);

    }
}
```

Estos ocho tipos de datos son lo que se conoce como tipos de datos primitivos (porque forman parte del lenguaje estándar, y a partir de ellos podremos crear otros más complejos).

Nota para C y C++: estos tipos de datos son muy similares a los que se emplean en C y en C++, apenas con alguna pequeña diferencia, como es el hecho de que el tamaño de cada tipo de datos (y los valores que puede almacenar) son independientes en Java del sistema que empleemos, cosa que no ocurre en C y C++, lenguajes en los que un valor de 40.000 puede ser válido para un int en unos sistemas, pero desbordar el máximo valor permitido en otros. Otra diferencia es que en Java no existen enteros sin signo (unsigned). Y otra más es que el tipo boolean ya está incorporado al lenguaje, en vez de corresponder a un entero disinto de cero

2.2. Variables y asignación

La forma de declarar variables de estos tipos es la siguiente:

```
int numeroEntero;     // La variable numeroEntero será un número de tipo int
short distancia;      // La variable distancia guardará números short
long gastos;          // La variable gastos es de tipo long
byte edad;            // Un entero de valores pequeños
float porcentaje;     // Con decimales, unas 6 cifras de precisión
double numPrecision; // Con decimales y precisión de unas 14 cifras
```

(es decir, primero se indica el tipo de datos que guardará la variable y después el nombre que queremos dar a la variable).

Se pueden declarar varias variables del mismo tipo a la vez:

```
int primerNumero, segundoNumero;    // Dos enteros
```

Y también se puede dar un valor a las variables a la vez que se declaran:

```
int a = 5;            // a es un entero, e inicialmente vale 5
short b=-1, c, d=4;   // b vale -1, c vale 0, d vale 4
```

Los nombres de variables pueden contener letras y números (y algún otro símbolo, como el de subrayado, pero no podrán contener otros muchos símbolos, como los de las distintas operaciones matemáticas posibles, ni podrán tener vocales acentuadas o eñes).

Vamos a aplicarlo con un ejemplo sencillo en Java que sume dos números:

```
    /* ------------------------------*
     *  Introducción a Java - Ejemplo *
     * ------------------------------*
     *  Fichero: suma.java           *
     *  (Suma dos números enteros)    *
     * ------------------------------*/


class Suma2 {

  public static void main( String args[] ) {

     int primerNumero = 56;    // Dos enteros con valores prefijados
     int segundoNumero = 23;

     System.out.println( La suma es: );  // Muestro un mensaje de aviso
     System.out.println( primerNumero+segundoNumero );

       // y el resultado de la operación
   }
}
```

El mismo código pero utilizando la representación como objetos

```
public class Sumados {
    int primernumero, segundonumero, suma;
```

```
    Sumados(int uno, int dos) {
        primernumero = uno;
        segundonumero = dos;
        suma = primernumero + segundonumero;
    }

    @Override
    public String toString(){
       return "La suma es " + suma;
    }
}
```

Para ejecutar el código hacemos

```
 */
public class prueba {
    static public void main(String args[]) {

        Sumados s = new Sumados(1,2);
        System.out.println(s);
    }
}
```

2.3. Cadenas de caracteres

En Java las cadenas se tratan de forma diferente a C/C++, donde las cadenas son matrices de caracteres en las que se indica el final de la cadena con un carácter de valor '\0'.

En Java las cadenas son objetos de las clases predefinida String o StringBuffer, que están incluidas en el paquete java.lang.*.

Siempre que aparecen conjuntos de caracteres entre comillas dobles, el compilador de Java crea automáticamente un objeto String.

Si sólo existieran cadenas de sólo lectura (String), durante una serie de manipulaciones sobre un objeto String habría que crear un nuevo objeto para cada uno de los resultados intermedios.

El compilador es más eficiente y usa un objeto StringBuffer para construir cadenas a partir de las expresiones, creando el String final sólo cuando es necesario. Los objetos StringBuffer se pueden modificar, de forma que no son necesarios nuevos objetos para albergar los resultados intermedios.

Los caracteres de las cadenas tienen un índice que indica su posición. El primer carácter de una cadena tiene el índice 0, el segundo el 1, el tercero el 2 y así sucesivamente. Esto puede sonar familiar a los programadores de C/C++, pero resultar chocante para aquellos programadores que provengan de otros lenguajes.

Un ejemplo de uso de las cadenas de caracteres es:

```
public class Cadenas {
    static public void main(String args[]) {
        String a = "Hola Mundo";
        System.out.println(a);
    }
}
```

2.4. Expresiones

Hay varias expresiones matemáticas que son frecuentes. Veremos cómo expresarlas en Java, y también veremos otras expresiones menos frecuentes.

Las que más encontramos normalmente son:

2.4.1. Operadores Aritméticos

Los operadores aritméticos más comunes son:

Operador	Símbolo
Suma	+
Resta	-
Multiplicación	*
División	/

Hemos visto un ejemplo de cómo calcular la suma de dos números; las otras operaciones se emplearían de forma similar.

Ejemplo

Dada la siguiente información, hacer un programa que permita calcular el área de un triángulo, de un cuadrado, y de un rectángulo, así como los volúmenes de un cilindro y de un cono.

El triángulo es un polígono formado por tres lados y tres ángulos. La suma de todos sus ángulos siempre es 180 grados. Para calcular el área se emplea la siguiente fórmula:

Área del triángulo = (base * altura) / 2

El cuadrado es un polígono de cuatro lados, con la particularidad de que todos ellos son iguales. Además sus cuatro ángulos son de 90 grados cada uno. El área de esta figura se calcula mediante la fórmula:

Área del cuadrado = lado al cuadrado

El rectángulo es un polígono de cuatro lados, iguales dos a dos. Sus cuatro ángulos son de 90 grados cada uno. El área de esta figura se calcula mediante la fórmula:

Área del rectángulo = base * altura

El cilindro es el sólido engendrado por un rectángulo al girar en torno a uno de sus lados. Para calcular su área lateral, su área total así como para ver su desarrollo pulsar sobre la figura anterior Para calcular su volumen se emplea la siguiente fórmula:

Volumen del cilindro = área de la base * altura

El cono es el sólido engendrado por un triángulo rectángulo al girar en torno a uno de sus catetos. Para calcular su área lateral, su área total así como para ver su desarrollo pulsar sobre la figura anterior Para calcular su volumen se emplea la siguiente fórmula:

Volumen del cono = (área de la base * altura) / 3

La implementación en Java es:

```
public class Areas_Volumenes {

  public static void main(String[] args)
  {
    // datos

    double base = 10, altura = 5, radio = 2;

    // areas

    double Atriangulo  = (base * altura) / 2.0;
    double Acuadrado   = base*base;
    double Arectangulo = base * altura;

    // volumenes
```

```
    double Abase       = 3.1416*radio*radio;
    double Vcilindro   = Abase * altura ;
    double Vcono       = Abase * altura / 3.0;

    System.out.println("Area de triangulo  = " + Atriangulo);
    System.out.println("Area de cuadrado   = " + Acuadrado);
    System.out.println("Area de rectangulo = " + Arectangulo);

    System.out.println("Volumen cilindro   = " + Vcilindro);
    System.out.println("Volumen cono       = " + Vcono);

  }
}
```

El código del objeto cuadro puede ser escrito como

```
public class Cuadro {
    float base;

    Cuadro(float unabase){
        base = unabase;
    }

    float Area() {
        return base*base;
    }

    float Perimetro() {
        return 4.0f*base;
    }

    @Override
    public String toString(){
       return "El cuadro de base  " + base + " tiene un perimetro " +
               Perimetro() + " y area = " + Area();
    }
}
```

y para el triangulo como

```
public class Triangulo {
    float base, altura;
```

```
    Triangulo(float unaBase, float unaAltura) {
        base = unaBase;
        altura = unaAltura;
    }

    float Area() {
        return base*altura;
    }

    float Perimetro() {
        float lado;

        lado = (float)Math.atan(2.0f*altura/base);

        return (2.0f*lado + base);
    }

    @Override
    public String toString(){
      return "El triangulo de base  " + base + " y altura " + altura + " tiene un perimetro " +
                Perimetro() + " y area = " + Area();
    }
}
```

El código para probar estas clase queda de la siguiente manera

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

        Cuadro a = new Cuadro(2.5f);
        System.out.println(a);

        Triangulo t = new Triangulo(2.0f, 2.5f);
        System.out.println(t);
    }
}
```

2.4.2. Otras operaciones matemáticas menos habituales

En la subvención anterior describimos esas son las operaciones que todo el mundo conoce, aunque no haya manejado computadoras. Pero hay otras operaciones que son frecuentes

en informática y no tanto para quien no ha manejado computadoras, pero que aun así deberíamos ver:

Operación	Símbolo
Resto de la división	%
Desplazamiento de bits a la derecha	>>
Desplazamiento de bits a la izquierda	<<
Producto lógico (and)	&
Suma lógica (or)	|
Suma exclusiva (xor)	^
Complemento	~

El resto de la división también es una operación conocida. Por ejemplo, si dividimos 14 entre 3 obtenemos 4 como cociente y 2 como resto, de modo que el resultado de 10 %3 sería 2.

Las operaciones a nivel de bits deberían ser habituales para los estudiantes de informática, pero quizás no tanto para quien no haya trabajado con el sistema binario. No entraré por ahora en más detalles, salvo poner un par de ejemplos para quien necesite emplear estas operaciones: para desplazar los bits de a dos posiciones hacia la izquierda, usaríamos *b* << 2; para invertir los bits de c (complemento) usaríamos tilde c; para hacer una suma lógica de los bits de d y f sería d | f. Incremento y asignaciones abreviadas.

2.4.3. Operadores de Asignación

Hay varias operaciones muy habituales , que tienen una sintaxis abreviada en Java. Por ejemplo, para sumar 2 a una variable a, la forma normal de conseguirlo sería:

a = a + 2;

pero existe una forma abreviada en Java:

a += 2;

Al igual que tenemos el operador += para aumentar el valor de una variable, tenemos -= para disminuirlo, /= para dividirla entre un cierto número, *= para multiplicarla por un número, y así sucesivamente. Por ejemplo, para multiplicar por 10 el valor de la variable b haríamos

b *= 10;

También podemos aumentar o disminuir en una unidad el valor de una variable, empleando los operadores de incremento (++) y de decremento (–). Así, para sumar 1 al valor de a, podemos emplear cualquiera de estas tres formas:

a = a +1; a += 1; a++;

Los operadores de incremento y de decremento se pueden escribir antes o después de la variable. Así, es lo mismo escribir estas dos líneas:

a++; ++a;

Pero hay una diferencia si ese valor se asigna a otra variable al mismo tiempo que se incrementa/decrementa:

int c = 5; int b = c++;

da como resultado c = 6 y b = 5 (se asigna el valor a b antes de incrementar c) mientras que

int c = 5; int b = ++c;

da como resultado c = 6 y b = 6 (se asigna el valor a b después de incrementar c).

2.4.4. Operadores Relacionales

También podremos comprobar si entre dos números (o entre dos variables) existe una cierta relación del tipo ¿es a mayor que b? o ¿tiene a el mismo valor que b?. Los operadores que utilizaremos para ello son:

Operación	Símbolo
Mayor que	>
Mayor o igual que	>=
Menor que	<
Menor o igual que	<=
Igual que	==
Distinto de	! =

Así, por ejemplo, para ver si el valor de una variable b es distinto de 5, escribiríamos algo parecido (veremos la sintaxis correcta un poco más adelante) a

SI b != 5 ENTONCES ...

o para ver si la variable a vale 70, sería algo como (nuevamente, veremos la sintaxis correcta un poco más adelante)

SI a == 70 ENTONCES ...

Es muy importante recordar esta diferencia: para asignar un valor a una variable se emplea un único signo de igual, mientras que para comparar si una variable es igual a otra (o a un cierto valor) se emplean dos signos de igual.

2.4.5. Operadores lógicos

Podremos enlazar varias condiciones, para indicar qué hacer cuando se cumplan ambas, o sólo una de ellas, o cuando una no se cumpla. Los operadores que nos permitirán enlazar condiciones son:

Operación	Símbolo
Y	&&
O	||
No	!

Por ejemplo, la forma de decir si a vale 3 y b es mayor que 5, o bien a vale 7 y b no es menor que 4 en una sintaxis parecida a la de Java (aunque todavía no es la correcta) sería:

```
SI (a==3 && bb>5) || (a==7  && ! (b<4))
```

2.5. Conversión de datos

En Java es posible transformar el tipo de una variable u objeto en otro diferente al original con el que fue declarado. Este proceso se denomina conversión, moldeado o tipado y es algo que debemos manejar con cuidado pues un mal uso de la conversión de tipos es frecuente que dé lugar a errores.

Una forma de realizar conversiones consiste en colocar el tipo destino entre paréntesis, a la izquierda del valor que queremos convertir de la forma siguiente: Tipo VariableNueva = (NuevoTipo) VariableAntigua;

Por ejemplo:

```
int miNumero = (int) ObjetoInteger;
char c = (char)System.in.read();
```

En el primer ejemplo, extraemos como tipo primitivo int el valor entero contenido en un campo del objeto Integer. En el segundo caso, la función read devuelve un valor int, que se convierte en un char debido a la conversión (char), y el valor resultante se almacena en la variable de tipo carácter c.

El tamaño de los tipos que queremos convertir es muy importante. No todos los tipos se convertirán de forma segura. Por ejemplo, al convertir un long en un int, el compilador corta los 32 bits superiores del long (de 64 bits), de forma que encajen en los 32 bits del int, con lo que si contienen información útil, ésta se perderá. Este tipo de conversiones que suponen pérdida de información se denominan “conversiones no seguras” y en general se tratan de evitar, aunque de forma controlada pueden usarse puntualmente.

De forma general trataremos de atenernos a la norma de que .^{en} las conversiones debe evitarse la pérdida de información". En la siguiente tabla vemos conversiones que son seguras por no suponer pérdida de información.

TIPO ORIGEN	TIPO DESTINO
byte	double, float, long, int, char, short
short	double, float, long, int
char	double, float, long, int
int	double, float, long
long	double, float
float	Double

No todos los tipos se pueden convertir de esta manera. Como alternativa, existen otras formas para realizar conversiones.

2.5.1. Ejemplo

Escribir un código en Java que permita redondear un número flotante a dos cifras significativas

```
public class redondea {
    static public void main(String args[]) {
        float pi = 3.1415926f;
        int pi_nvo = (int) (pi*100.0f + 0.5f);
        System.out.println(pi_nvo);
        pi = (float )pi_nvo/100.0f;
        System.out.println(pi);
    }
}
```

2.6. Clases envoltorio o wrapper

En ocasiones es muy conveniente poder tratar los datos primitivos (int, boolean, etc.) como objetos. Por ejemplo, los contenedores definidos por el API en el package java.util (Arrays dinámicos, listas enlazadas, colecciones, conjuntos, etc.) utilizan como unidad de almacenamiento la clase Object. Dado que Object es la raíz de toda la jerarquía de objetos en Java, estos contenedores pueden almacenar cualquier tipo de objetos. Pero los datos primitivos no son objetos, con lo que quedan en principio excluidos de estas posibilidades.

Para resolver esta situación el API de Java incorpora las clases envoltorio (wrapper class), que no son más que dotar a los datos primitivos con un envoltorio que permita tratarlos

como objetos. Por ejemplo podríamos definir una clase envoltorio para los enteros, de forma bastante sencilla, con:

```
public class Entero {
    private int valor;

    Entero(int valor) {
        this.valor = valor;
    }

    int intValue() {
        return valor;
    }
}
```

La API de Java hace innecesario esta tarea al proporcionar un conjunto completo de clases envoltorio para todos los tipos primitivos. Adicionalmente a la funcionalidad básica que se muestra en el ejemplo las clases envoltorio proporcionan métodos de utilidad para la manipulación de datos primitivos (conversiones de / hacia datos primitivos, conversiones a String, etc.)

Las clases envoltorio existentes son:

Byte para byte. Short para short. Integer para int. Long para long. Boolean para boolean Float para float. Double para double y Character para char.

Observe que las clases envoltorio tienen siempre la primera letra en mayúsculas.

Las clases envoltura se usan como cualquier otra:

```
Integer i = new Integer(5);
int x = i.intValue();
```

Hay que tener en cuenta que las operaciones aritméticas habituales (suma, resta, multiplicación ...) están definidas solo para los datos primitivos por lo que las clases envoltura no sirven para este fin.

Las variables primitivas tienen mecanismos de reserva y liberación de memoria más eficaces y rápidos que los objetos por lo que deben usarse datos primitivos en lugar de sus correspondientes envolturas siempre que se pueda.

Uso de Clases y Objetos

3.1. La API de Java

El API Java es una Interfaz de Programación de Aplicaciones (API: por sus siglas en inglés) provista por los creadores del lenguaje Java, y que da a los programadores los medios para desarrollar aplicaciones Java. Como el lenguaje Java es un Lenguaje Orientado a Objetos, la API de Java provee de un conjunto de clases utilitarias para efectuar toda clase de tareas necesarias dentro de un programa. La API Java está organizada en paquetes lógicos, donde cada paquete contiene un conjunto de clases relacionadas semánticamente.

La dirección donde puedes consultar la Documentación del API de Java Versión 6 es está:

https://docs.oracle.com/javase/8/docs/api/

En la parte superior Izquierda, vienen los paquetes. En la inferior Izquierda vienen las clases, ahí por ejemplo buscas String, le das click, luego en la derecha te aparece toda la descripción de la clase.

1.- El paquete en el que se encuentra

2.- De que clase hereda

3.- Las interfaces que implementa

4.- Una Breve Descripción

5.- Las constantes declaradas en la clase

6.- Los tipos de constructores

7.- Los métodos de la clase, con sus parámetros y lo que regresan

3.2. Creación de objetos

Comenzaremos con una analogía simple, para ayudarle a comprender el concepto de las clases y su contenido. Suponga que desea conducir un automóvil y, para hacer que aumente su velocidad, debe presionar el pedal del acelerador. ¿Qué debe ocurrir antes de que pueda hacer esto? Bueno, antes de poder conducir un automóvil, alguien tiene que diseñarlo. Por lo general, un automóvil empieza en forma de dibujos de ingeniería, similares a los planos de construcción que se utilizan para diseñar una casa. Estos dibujos de ingeniería incluyen el diseño del pedal del acelerador, para que el automóvil aumente su velocidad. El pedal oculta los complejos mecanismos que se encargan de que el automóvil aumente su velocidad, de igual forma que el pedal del freno oculta los mecanismos que disminuyen la velocidad del automóvil y por otro lado, el volante oculta los mecanismos que hacen que el automóvil de vuelta. Esto permite que las personas con poco o nada de conocimiento acerca de cómo funcionan los motores puedan conducir un automóvil con facilidad. Desafortunadamente, no puede conducir los dibujos de ingeniería de un auto. Antes de poder conducir un automóvil, éste debe construirse a partir de los dibujos de ingeniería que lo describen. Un automóvil completo tendrá un pedal acelerador verdadero para hacer que aumente su velocidad, pero aún así no es suficiente; el auto- móvil no acelerará por su propia cuenta, así que el conductor debe oprimir el pedal del acelerador.

Ahora utilizaremos nuestro ejemplo del automóvil para introducir los conceptos clave de programación. Para realizar una tarea en una aplicación se requiere de un método. El método describe los mecanismos que se encargan de realizar sus tareas y oculta la tareas complejas que realiza, de la misma forma que el pedal del acelerador de un automóvil oculta al conductor los complejos mecanismos para hacer que el automóvil vaya más rápido. En Java, empezamos por crear una unidad de aplicación llamada clase para alojar a un método, así como los dibujos de ingeniería de un automóvil alojan el diseño del pedal del acelerador. En una clase se proporcionan uno o más métodos, los cuales están diseñados para realizar las tareas de esa clase. Por ejemplo, una clase que representa a una cuenta bancaria podría contener un método para depositar dinero en una cuenta, otro para retirar dinero de una cuenta y un tercero para solicitar el saldo actual de la cuenta.

Una manera de especificar esto es utilizando los diagramas UML (Unified Modeling Language), el cual permite especificar los métodos y variables para un objeto. Como ejemplo consideremos crear una objeto LibroCalificaciones, el diagrama para esta clase se muestra en la Figura 3.1.

En la Figura 3.1 podemos ver que el diagrama esta dividido en tres partes. La primer parte es la correspondiente al nombre del objeto, la segunda son las variables relacionadas con el objeto y finalmente los métodos. Para cada uno de los métodos se especifica un signo, menos para una variable o método privado y más para publico. Una variable y/o método será pública cuando el programador desea que sea manipulada por el usuario final y en caso

LibroCalificaciones

– nombreDelCurso : String

«constructor» LibroCalificaciones(nombre : String)
+ establecerNombreDelCurso(nombre : String)
+ obtenerNombreDelCurso( ) : String
+ mostrarMensaje( )

Figura 3.1: Diagrama UML para el objeto LibroCalificaciones

contrario la declara privada. En particular existe un método que se utiliza para inicializar las variables del objeto y se conoce como constructor. Un método constructor siempre tiene el nombre del objeto. Adicionalmente podemos ver el tipo de variable con dos puntos y los parámetros que recibe el método (en parentesis) y el tipo que regresa con dos puntos.

La implementación en Java para este objeto es

```
public class LibroCalificaciones {
    private String nombreDelCurso; // nombre del curso para este LibroCalificaciones
   // el constructor inicializa nombreDelCurso con el objeto
   //String que se provee como argumento
   public LibroCalificaciones( String nombre )
   {
      nombreDelCurso = nombre; // inicializa nombreDelCurso
   } // fin del constructor
   // método para establecer el nombre del curso

   public void establecerNombreDelCurso( String nombre )
   {
      nombreDelCurso = nombre; // almacena el nombre del curso
   } // fin del método establecerNombreDelCurso
   // método para obtener el nombre del curso

   public String obtenerNombreDelCurso()
   {
      return nombreDelCurso;
   } // fin del método obtenerNombreDelCurso
   // muestra un mensaje de bienvenida al usuario de LibroCalificaciones
```

```
   public void mostrarMensaje()
   {
      // esta instrucción llama a obtenerNombreDelCurso para obtener el
      // nombre del curso que este LibroCalificaciones representa
      System.out.printf("Bienvenido al Libro de calificaciones para\n\%s!\n",
         obtenerNombreDelCurso() );
   } // fin del método mostrarMensaje
} // fin de la clase LibroCalificaciones
```

La manera de utilizar esta clase es escribiendo otra que tenga el método main, así por ejemplo tenemos

```
public class PruebaLibroCalificaciones {
    // el método main empieza la ejecución del programa
    public static void main( String args[] )
    {
      // crea objeto LibroCalificaciones
      LibroCalificaciones libroCalificaciones1 = new LibroCalificaciones(
         "CS101 Introduccion a la programacion en Java" );
      LibroCalificaciones libroCalificaciones2 = new LibroCalificaciones(
         "CS102 Estructuras de datos en Java" );
      // muestra el valor inicial de nombreDelCurso para cada LibroCalificaciones
      System.out.printf("El nombre del curso de libroCalificaciones1 es: \%s\n",
         libroCalificaciones1.obtenerNombreDelCurso() );
      System.out.printf("El nombre del curso de libroCalificaciones2 es: \%s\n",
         libroCalificaciones2.obtenerNombreDelCurso() );
   } // fin de main
} // fin de la clase PruebaLibroCalificaciones
```

3.2.1. Ejemplo de manejo de Números Complejos

En este ejemplo se pretende escribir el código correspondiente para hacer la implementación de las operaciones básicas para el manejo de números complejos. La descripción de la clase se da en el diagrama UML de la figura 3.2

Para este caso el código implementado queda como

```
package Capitulo_3.NumerosComplejos;

/**
 *
```

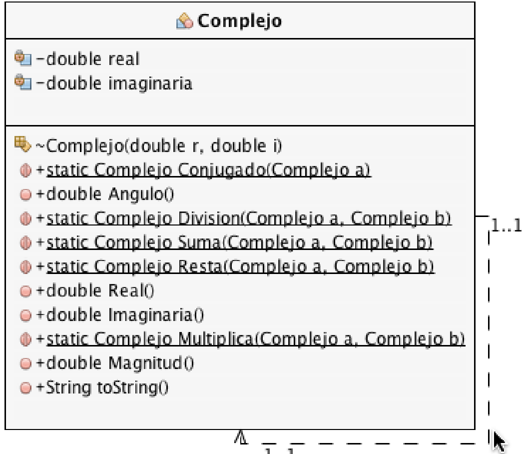

Figura 3.2: Diagrama UML para manejar números complejos

```
 * @author felix
 */
public class Complejo {
    private double real;
    private double imaginaria;

    Complejo(double r, double i) {
        real = r;
        imaginaria = i;
    }

    static public Complejo Conjugado(Complejo a) {
        return new Complejo(a.real, -a.imaginaria);
    }

    public double Angulo() {
        return Math.atan2(imaginaria, real);
    }

    static public Complejo Division(Complejo a, Complejo b) {
        double r, i, mag;

        mag = b.real*b.real + b.imaginaria*b.imaginaria;
```

```
        r = a.real*b.real + a.imaginaria*b.imaginaria;
        i = -a.real*b.imaginaria + a.imaginaria*b.real;

        return new Complejo(r/mag, i/mag);
    }

    static public Complejo Suma(Complejo a, Complejo b) {
        double r, i;

        r = a.real + b.real;
        i = a.imaginaria + b.imaginaria;

        return new Complejo(r, i);
    }

    static public Complejo Resta(Complejo a, Complejo b) {
        double r, i;

        r = a.real - b.real;
        i = a.imaginaria - b.imaginaria;

        return new Complejo(r, i);
    }

    public double Real() {
        return real;
    }

    public double Imaginaria() {
        return imaginaria;
    }

    static public Complejo Multiplica(Complejo a, Complejo b) {
        double r, i;

        r = a.real*b.real - a.imaginaria*b.imaginaria;
        i = a.real*b.imaginaria + a.imaginaria*b.real;

        return new Complejo(r, i);
    }
```

```
    public double Magnitud() {
        return Math.sqrt(real*real + imaginaria*imaginaria);
    }

    @Override
    public String toString() {
        String aux = "";
        aux += real;
        if (imaginaria < 0)
            aux += " - j "+(-imaginaria);
        else if(imaginaria > 0)
            aux += " + j " + imaginaria;
        return aux;
    }
}
```

Su uso dentro de otro programa se puede hacer de la siguiente manera

```
package Capitulo_3.NumerosComplejos;

import static Capitulo_3.NumerosComplejos.Complejo.Conjugado;
import static Capitulo_3.NumerosComplejos.Complejo.Division;
import static Capitulo_3.NumerosComplejos.Complejo.Multiplica;
import static Capitulo_3.NumerosComplejos.Complejo.Resta;
import static Capitulo_3.NumerosComplejos.Complejo.Suma;

/**
 *
 * @author felix
 */
public class prueba {
    static public void main(String args[]) {
        Complejo a = new Complejo(3, 4);
        Complejo b = new Complejo(3, -1);

        System.out.println("a = " + a);
        System.out.println("b = " + b);

        System.out.println("Parte real de a = " + a.Real());
        System.out.println("Parte imaginaria de a = " + a.Imaginaria());
        System.out.println("Magnitud de a = " + a.Magnitud());
        System.out.println("Angulo de a = " + a.Angulo());
```

```
        System.out.println("Conjugado de a = " + Conjugado(a));

        System.out.println("a + b = " + Suma(a, b));
        System.out.println("a - b = " + Resta(a, b));
        System.out.println("a * b = " + Multiplica(a,b));
        System.out.println("a / b = " + Division(a, b));
    }
}
```

Como resultado de la ejecución tenemos

```
a = 3.0 + j 4.0
b = 3.0 - j 1.0
Parte real de a = 3.0
Parte imaginaria de a = 4.0
Magnitud de a = 5.0
Angulo de a = 0.9272952180016122
Conjugado de a = 3.0 - j 4.0
a + b = 6.0 + j 3.0
a - b = 0.0 + j 5.0
a * b = 13.0 + j 9.0
a / b = 0.5 + j 1.5
```

3.2.2. Ejemplo de una Cuenta Bancaria

Escribir un el código Java para hacer la representación del objeto Cuenta tal como se describe en el diagrama UML de la Figura 3.3.

```
public class Cuenta {
    private double saldo; // variable de instancia que almacena el saldo

    public Cuenta( double saldoInicial )
    {
      // valida que saldoInicial sea mayor que 0.0;
      // si no lo es, saldo se inicializa con el valor predeterminado 0.0
      if ( saldoInicial > 0.0 )
         saldo = saldoInicial;
    } // fin del constructor de Cuenta

    // abona (suma) un monto a la cuenta
```

```
    public void abonar( double monto )
    {
      saldo = saldo + monto; // suma el monto al saldo
    } // fin del método abonar

    // devuelve el saldo de la cuenta

    public double obtenerSaldo()
    {
      return saldo; // proporciona el valor de saldo al método que hizo la llamada
    } // fin del método obtenerSaldo
} // fin de la clase Cuenta
```

Cuenta

– balance : Double

«constructor» Cuenta(saldoInicial : Double)
+ abonar(monto : Double)
+ obtenerSaldo( ) : Double

Figura 3.3: Diagrama UML para el objeto Cuenta

Para probar escribimos el siguiente código

```
import java.util.Scanner;


public class PruebaCuenta {
// el método main empieza la ejecución de la aplicación de Java
   public static void main( String args[] )
   {
       Cuenta cuenta1 = new Cuenta(50.00);
       Cuenta cuenta2 = new Cuenta(-7.53);

       System.out.println("Saldo de cuenta1: " + cuenta1.obtenerSaldo());
       System.out.println("Saldo de cuenta2: " + cuenta2.obtenerSaldo());

       Scanner entrada = new Scanner(System.in);
       double montoDeposito;
```

```
        System.out.print("Escriba el monto a depositar en cuenta1: ");
        montoDeposito = entrada.nextDouble();
        System.out.println("Sumando saldo de cuenta1 " + montoDeposito);
        cuenta1.abonar(montoDeposito);

        System.out.println("Saldo de cuenta1 " + cuenta1.obtenerSaldo());
        System.out.println("Saldo de cuenta2 " + cuenta2.obtenerSaldo());

        System.out.print("Escriba el monto a depositar en cuenta2: ");
        montoDeposito = entrada.nextDouble();
        System.out.println("Sumando saldo de cuenta2 " + montoDeposito);
        cuenta2.abonar(montoDeposito);

        System.out.println("Saldo de cuenta1 " + cuenta1.obtenerSaldo());
        System.out.println("Saldo de cuenta2 " + cuenta2.obtenerSaldo());

    }
}
```

3.3. La clase String

Dentro de un objeto de la clases String o StringBuffer, Java crea un array de caracteres de una forma similar a como lo hace el lenguaje C++. A este array se accede a través de las funciones miembro de la clase.

Los strings u objetos de la clase String se pueden crear explícitamente o implícitamente. Para crear un string implícitamente basta poner una cadena de caracteres entre comillas dobles. Por ejemplo, cuando se escribe

```
System.out.println("El primer programa");
```

Java crea un objeto de la clase String automáticamente.

Para crear un string explícitamente escribimos

```
String str=new String("El primer programa");
```

También se puede escribir, alternativamente

```
String str="El primer programa";
```

Para crear un string nulo se puede hacer de estas dos formas

```
String str="";
```

```
String str=new String();
```

Un string nulo es aquél que no contiene caracteres, pero es un objeto de la clase String. Sin embargo,

```
String str;
```

está declarando un objeto str de la clase String, pero aún no se ha creado ningún objeto de esta clase.

3.3.1. Operaciones básicas, comunes a String y StringBuffer

Existen una serie de métodos que son comunes a ambas clases.

Los siguientes métodos de acceso a las cadenas:

int length(); Devuelve el número de caracteres de la cadena.

char charAt(int i); Devuelve el carácter correspondiente de índice i.

Los siguientes métodos para crear cadenas derivadas:

String toString(); Devuelve una copia del objeto como una String.

String substring(int i, int fin); Devuelve una instancia de la clase String que contenga una subcadena desde la posición ini, hasta la fin (si no se indica hasta el final de la cadena), del objeto cadena que invoque el método.

Y el método para transformar la cadena en un vector de caracteres:

void getChars(int ini, int fin, char[] destino, int destIni); Convierte la cadena en un vector de caracteres destino.

3.3.2. Métodos de la clase String

Constructores

La clase String proporciona cadenas de sólo lectura y soporta operaciones con ellas. Se pueden crear cadenas implícitamente mediante una cadena entrecomillada o usando + ó += con dos objetos String.

También se pueden crear objetos String explícitamente con el mecanismo new.

La clase String soporta multitud de constructores.

String(); Constructor por defecto. El nuevo String toma el valor .

String(String s); Crea un nuevo String, copiando el que recibe por parámetro.

String(StringBuffer s); Crea un String con el valor que en ese momento tenga el String-Buffer que recibe como parámetro.

String(char[] v); El nuevo String toma el valor de los caracteres que tiene el vector de caracteres recibido por parámetro.

String(byte[] v); El nuevo String toma el valor de los caracteres que corresponden a los valores del vector de bytes en el sistema de caracteres de la ordenador en que se ejecute.

Búsqueda en cadenas String

Además presenta los siguientes métodos para buscar caracteres o subcadenas en la cadena, y devuelven el índice que han encontrado o el valor –1 si la búsqueda no ha sido satisfactoria:

int indexOf(char ch, int start): Devuelve el índice correspondiente a la primera aparición del carácter ch en la cadena, comenzando a buscar desde el carácter start (si no se especifica se busca desde el principio).

int indexOf(String str): Devuelve el índice correspondiente al carácter en que empieza la primera aparición de la subcadena str.

int lastIndexOf(char ch, int start): Devuelve el índice correspondiente a la última aparición del carácter ch en la cadena, comenzando a buscar desde el carácter start (si no se especifica se busca desde el final).

int lastIndexOf(String str): Devuelve el índice correspondiente al carácter en que empieza la última aparición de la subcadena str.

Comparaciones de cadenas String

Java no trabaja con el código ASCII habitual, sino con el código avanzado Unicode.

El código Unicode (código universal) se caracteriza, sobre todo, por el uso de dos bytes por carácter. Esto permite aumentar los caracteres hasta 65000, y así se pueden representar los caracteres que componen las lenguas, vivas o muertas, más importantes del mundo.

Hay que tener en cuenta que si nos salimos del rango 0-255 que coincide con el código ASCII puede que las comparaciones no sean las esperadas.

Las funciones de comparación son las siguientes:

boolean equals(Object o); Devuelve true si se le pasa una referencia a un objeto String con los mismos caracteres, o false si no.

boolean equalsIgnoreCase(String s); Compara cadenas ignorando las diferencias de ortografía mayúsculas/minúsculas.

boolean regionMatches(boolean b, int o, String s , int i, int n); Compara parte de dos cadenas, carácter a carácter.

boolean startsWith(String s, int i); Comprueba si la cadena tiene el prefijo s desde i.

boolean endsWith(String s); Comprueba si la cadena termina con el sufijo s.

int compareTo(Object o); Devuelve un entero que es menor, igual o mayor que cero cuando la cadena sobre la que se le invoca es menor, igual o mayor que la otra. Si el parámetro es un String, la comparación es léxica.

int compareToIgnoraCase(String s); Compara lexicográficamente, ignorando las diferencias de ortografía mayúsculas/minúsculas.

3.4. Paquetes

Los paquetes son el mecanismo por el que Java permite agrupar clases, interfaces, excepciones y constantes. De esta forma, se agrupan conjuntos de estructuras de datos y de clases con algún tipo de relación en común.

Con la idea de mantener la reutilización y facilidad de uso de los paquetes desarrollados es conveniente que las clases e interfaces contenidas en los mismos tengan cierta relación funcional. De esta manera los desarrolladores ya tendrán una idea de lo que están buscando y fácilmente sabrán qué pueden encontrar dentro de un paquete.

Creación de un paquete

a.) Declaración

Para declarar un paquete se utiliza la sentencia package seguida del nombre del paquete que estemos creando:

package NombrePaquete;

La estructura que ha de seguir un fichero fuente en Java es:

Una única sentencia de paquete (opcional).

Las sentencias de importación deseadas (opcional).

La declaración de una (y sólo una) clase pública (public).

Las clases privadas del paquete (opcional).

Por lo tanto la sentencia de declaración de paquete ha de ser la primera en un archivo fuente Java.

Nomenclatura

Para que los nombres de paquete puedan ser fácilmente reutilizados en toda una compañía o incluso en todo el mundo es conveniente darles nombres únicos. Esto puede ser una tarea realmente tediosa dentro de una gran empresa, y absolutamente imposible dentro de la comunidad de Internet.

Por eso se propone asignar como paquetes y subpaquetes el nombre de dominio dentro de Internet. Se verá un ejemplo para un dominio que se llamase japon.magic.com, un nombre apropiado sería com.magic.japon.paquete.

Subpaquetes

Cada paquete puede tener a su vez paquetes con contenidos parecidos, de forma que un programador probablemente estará interesado en organizar sus paquetes de forma jerárquica. Para eso se definen los subpaquetes.

Para crear un subpaquete bastará con almacenar el paquete hijo en un directorio Paquete/Subpaquete.

Así una clase dentro de un subpaquete como Paquete.Subpaquete.clase estará codificada en el fichero Paquete/Subpaquete.java.

El JDK define una variable de entorno denominada CLASSPATH que gestiona las rutas en las que el JDK busca los subpaquetes. El directorio actual suele estar siempre incluido en la variable de entorno CLASSPATH.

Uso

Con el fin de importar paquetes ya desarrollados se utiliza la sentencia import seguida del nombre de paquete o paquetes a importar.

Se pueden importar todos los elementos de un paquete o sólo algunos.

Para importar todas las clases e interfaces de un paquete se utiliza el metacaracter *:

import PaquetePrueba.*;

También existe la posibilidad de que se deseen importar sólo algunas de las clases de un cierto paquete o subpaquete:

import Paquete.Subpaquete1.Subpaquete2.Clase1;

Para acceder a los elementos de un paquete, no es necesario importar explícitamente el paquete en que aparecen, sino que basta con referenciar el elemento tras una especificación completa de la ruta de paquetes y subpaquetes en que se encuentra.

```
Paquete.Subpaquetes1.Subpaquete2.Clase_o_Interfaz.elemento
```

En la API de Java se incluyen un conjunto de paquetes ya desarrollados que se pueden incluir en cualquier aplicación (o applet) Java que se desarrolle.

Ámbito de los elementos de un paquete

Al introducir el concepto de paquete, surge la duda de cómo proteger los elementos de una clase, qué visibilidad presentan respecto al resto de elementos del paquete, respecto a los de otros paquetes...

Ya en la herencia se vieron los identificadores de visibilidad public (visible a todas las clases), private (no visible más que para la propia clase), y protected (visible a clases hijas).

Por defecto se considera los elementos (clases, variables y métodos) de un mismo paquete como visibles entre ellos (supliendo las denominadas clases amigas de C++).

3.5. La clase Random

La clase Random proporciona un generador de números aleatorios. Para crear una secuencia de números aleatorios tenemos que seguir los siguientes pasos:

Proporcionar a nuestro programa información acerca de la clase Random.

1.- Al principio del programa escribiremos la siguiente sentencia.

```
import java.util.Random;
```

2.- Crear un objeto de la clase Random

3.- Llamar a una de las funciones miembro que generan un número aleatorio

4.- Usar el número aleatorio.

La clase dispone de dos constructores, el primero crea un generador de números aleatorios cuya semilla es inicializada en base al instante de tiempo actual.

```
Random rnd = new Random();
```

El segundo, inicializa la semilla con un número del tipo long.

```
Random rnd = new Random(3816L);
```

El sufijo L no es necesario, ya que aunque 3816 es un número int por defecto, es promocionado automáticamente a long.

Aunque no podemos predecir que números se generarán con una semilla particular, podemos sin embargo, duplicar una serie de números aleatorios usando la misma semilla. Es decir, cada vez que creamos un objeto de la clase Random con la misma semilla obtendremos la misma secuencia de números aleatorios. Esto no es útil en el caso de loterías, pero puede ser útil en el caso de juegos, exámenes basados en una secuencia de preguntas aleatorias, las mismas para cada uno de los estudiantes, simulaciones que se repitan de la misma forma una y otra vez, etc.

Algunos métodos de la clase Random son:

Método	Descripción
setSeed(int n)	cambia la semilla
nextInt()	genera un número aleatorio entero
nextLong()	genera un número aleatorio long
nextFloat()	genera un número aleatorio flotante entre 0.0 y 1.0
nextDouble()	genera un número aleatorio doble entre 0.0 y 1.0
nextGaussian()	genera un número entre 0 y 1 con distribución Normal con media cero y varianza 1

3.5.1. Ejemplo 1

Escribir un código para generar una secuencia de 10 números aleatorios entre 0.0 y 1.0 escribimos

```
import java.util.Random;


public class NumerosFlotantesAleatorios {

    static public void main(String args[]) {
        int i;
        Random rnd = new Random();

        for (i = 0; i < 10; i++) {
            System.out.println(rnd.nextDouble());;
        }
    }
}
```

3.5.2. Ejemplo 2

Crear una secuencia de 10 números aleatorios enteros comprendidos entre 0 y 9 ambos incluidos escribimos:

```
import java.util.Random;

public class NumerosEnterosAleatorios {
    static public void main(String args[]) {
        int i;
        Random rnd = new Random();

        for (i = 0; i < 10; i++) {
            System.out.println( (int)(rnd.nextDouble() * 10.0));
        }
    }
}
```

3.6. La clase Math

La clase Math es una clase estática ya que no necesita que se instancie una variable para poder ser utilizada. La clase Math representa la librería matemática de Java, en donde se compilan la mayoría de la funciones matemáticas mas conocidas. Algunas de las funciones que están en esta clase son:

Funciones	Descripción
Math.abs(x)	valor absoluto
Math.sin(double)	seno de un ángulo
Math.cos(double)	coseno de un ángulo
Math.tan(double)	tangente de un ángulo
Math.asin(double)	seno inverso
Math.acos(double)	coseno inverso
Math.atan(double)	tangente inversa
Math.atan2(double,double)	tangente inversa
Math.exp(double)	antilogaritmo
Math.log(double)	logaritmo
Math.sqrt(double)	raiz cuadráda
Math.ceil(double)	redondeo de un número
Math.floor(double)	parte fraccionaria de un número
Math.rint(double)	entero mas cercano
Math.pow(a,b)	potenciación
Math.round(x)	redondeo
Math.random()	numero alatorio entre 0 y 1
Math.max(a,b)	máximo de dos números
Math.min(a,b)	minimo de dos números
Math.E	base exponencial
Math.PI	valor de π

3.7. Formateo de Salida

Para dar formato de Salida se utilizan los métodos format y printf en lugar de las instrucciones de print y println anteriormente utilizadas. Los métodos format y printf tienen la misma sintaxis y pueden ser utilizados indistintamente. En el siguiente ejemplo se muestra su uso:

```
import java.util.*;

public class PruebaFormato {

    public static void main(String[] args) {
      long n = 461012;
      System.out.format("%d%n", n);
      System.out.format("%08d%n", n);
      System.out.format("%+8d%n", n);
      System.out.format("%,8d%n", n);
```

```
      System.out.format("%+,8d%n%n", n);

      double pi = Math.PI;
      System.out.format("%f%n", pi);
      System.out.format("%.3f%n", pi);
      System.out.format("%10.3f%n", pi);
      System.out.format("%-10.3f%n", pi);
      System.out.format(Locale.FRANCE, "%-10.4f%n%n", pi);

      Calendar c = Calendar.getInstance();
      System.out.format("%tB %te, %tY%n", c, c, c);
      System.out.format("%tl:%tM %tp%n", c, c, c);
      System.out.format("%tD%n", c);
    }
}
```

Otro ejemplo mas elaborado es el que se muestra a continuación

```
import java.text.DecimalFormat;

public class FormatoDecimal {

   static public void FormatoAjustable(String pattern, double value ) {
      DecimalFormat myFormato = new DecimalFormat(pattern);
      String output = myFormato.format(value);
      System.out.println(value + "  " + pattern + "  " + output);
   }

   static public void main(String[] args) {

      FormatoAjustable("###,###.###", 123456.789);
      FormatoAjustable("###.##", 123456.789);
      FormatoAjustable("000000.000", 123.78987);
      FormatoAjustable("$###,###.###", 12345.67);
   }
}
```

Los símbolos utilizados para hacer la conversión y el formato se muestran en la siguiente tabla

Conv.	Band.	Explicación
d		Un entero decimal.
f		Un flotante.
n		nueva línea apropiado
tB		Conversión de fecha y hora nombre completo del mes específico a la localidad.
td, t e		Conversión de fecha y hora día del mes de 2 dígitos. td tiene ceros a la izquierda según se necesite, te no.
ty, tY		Conversión de fecha y hora—ty = año de 2 dígitos, tY = año de 4 dígitos.
tl		Una conversión de fecha y hora—hora en reloj de 12 horas.
tM		Conversión de fecha y hora—minutos en 2 dígitos, con ceros a la izquierda según sea necesario.
tp		Conversión de fecha y hora —am/pm específico a la localidad (minúsculas).
tm		Conversión de fecha y hora meses en 2 dígitos, con ceros a la izquierda según sea necesario.
tD		Conversión de fecha y hora—la fecha como %tm %td %ty
	08	Ocho caracteres en longitud, con ceros a la izquierda según sea necesario.
	+	Incluye el signo, ya sea positivo o negativo.
	,	Incluye caracteres de agrupación específicos a la localidad.
	-	Justificado a la izquierda.
	.3	Tres cifras después del punto decimal.
	10.3	Diez caracteres en longitud, justificado a la derecha, con tres cifras después del punto decimal.

Control de Flujo

4.1. Sentencia if

En cualquier lenguaje de programación es habitual tener que comprobar si se cumple una cierta condición. La forma normal de conseguirlo es empleando una construcción de la forma:

```
SI   condición_se_cumple   ENTONCES   pasos_a_dar
SINO pasos_alternos
```

Esta construcción se encuentra en la mayoría de los lenguajes y es conocida como intrusiones condicionales. En el caso de C, la forma exacta será empleando if (si, en inglés), seguido por la condición entre paréntesis, e indicando finalmente entre llaves los pasos que queremos dar si se cumple la condición, así :

if (condición) sentencias

Por ejemplo,

```
if (x == 3)  {
  System.out.println( "El valor es correcto" );
  resultado = 5;
}
```

Nota: Si solo queremos dar un paso en caso de que se cumpla la condición, no es necesario emplear llaves. Las llaves serán imprescindibles solo cuando haya que hacer varias cosas:

```
if (x == 3)
  System.out.println( "El valor es correcto" );
```

Una primera mejora, que también permiten muchos lenguajes de programación, es indicar qué hacer en caso de que no se cumpla la condición. Sería algo parecido a

```
SI  condición_a_comprobar  ENTONCES  pasos_a_dar
```

```
EN_CASO_CONTRARIO  pasos_alternativos
```

que en Java escribiríamos así:

```
if (condición)  { sentencias1 }   else   { sentencias2 }
```

Por ejemplo,

```
if (x == 3)  {
  System.out.println( "El valor es correcto" );
  resultado = 5;
}
else  {
  System.out.println( "El valor es incorrecto" );
  resultado = 27;
}
```

4.1.1. Ejemplo Mayor Menor

El siguiente ejemplo permite calcular dado tres números cual es el mayor y cual es el menor de ellos.

```
public class mayor_menor {

    static public void main (String args[])
    {
      int a, b, c, d;

      a = 4; b = 2; c = 3;
      d = (a > b) ?  a : b;
      d = (d > c) ?  d : c;

      System.out.println("el Mayor es " + d);


      // otra forma

      if(a> b) d = a;
      else d = b;
      if(d< c) d = c;
```

```
      System.out.println("el Mayor es " + d);

    }
}
```

4.1.2. Ejemplo Triángulo

Este ejemplo nos permite determinar dado los tres lados de un triángulo, si es un triángulo Equilátero, Escaleno Isósceles y calcula además del tipo de triángulo los ángulos internos del mismo. Toda la clase se presenta como un objeto

El diagrama UML para este ejemplo es:

Triangulo
- a : Double - b : Double - c : Double - A : Double - B : Double - C : Double
«constructor» Triangulo(Double, Double, Double) + EsUnTriangulo(double unaA, double unaB, double unaC): boolean + info():String + Area() : Double + Perimetro() : Double + toString() : String

```
package Capitulo_4.Triangulos;

/**
 *
 * @author calderon
 */
public class Triangulo {
    private double a;
    private double b;
    private double c;
    private double A;
    private double B;
    private double C;

    public Triangulo(double unaA, double unaB, double unaC) {
```

```
        if(EsUnTriangulo(unaA, unaB, unaC) ) {
            a = unaA;
            b = unaB;
            c = unaC;

            C = Math.acos((a*a + b*b - c*c)/(2.0f*a*b));
            A = Math.acos((c*c + b*b - a*a)/(2.0f*c*b));
            B = Math.acos((a*a + c*c - a*a)/(2.0f*a*c));

            A *= 180.0f/Math.PI;
            B *= 180.0f/Math.PI;
            C *= 180.0f/Math.PI;

        }
        else {
            a= -1;
            b= -1;
            c= -1;
        }
    }

    public boolean EsUnTriangulo(double unaA, double unaB, double unaC) {
        if(unaA < 0) return false;
        if(unaB < 0) return false;
        if(unaC < 0) return false;

        if(unaA + unaB >= unaC) return true;
        if(unaB + unaC >= unaB) return true;
        if(unaC + unaB >= unaA) return true;
        return false;
    }

    public String info() {
        String aux;

        aux = "Los lados del triangulo son a = " +a + " b = " + b + " c = " + c +"\n";
        aux += "Los angulos son A = " + A + " B = " + B + " C = " + C +"\n";

        if(a == b && a == c)
            aux += "Es un triangulo Equilatero \n";
        else if(a == b || a == c || b == c)
```

```
            aux += "Es un triangulo Isoceles \n";
        else aux += "Es un triangulo Escaleno \n";

        if(A == 90 || B == 90 || C == 90)
            aux += "Adicionalemte es un triangulo Rectangulo \n";

        aux += "El area del triangulo es " + Area() + " y el perimetro es " +
                Perimetro();

        return aux;
    }

    public double Area() {
        double s = (a + b + c)/2.0f;
        double area;
        area = (double)Math.sqrt(s*(s-a)*(s-b)*(s-c));
        return area;
    }

    public double Perimetro() {
        return (a+b+c);
    }

    @Override
    public String toString() {

        if(a < 0 && b < 0 && c <0)
            return "Los lados proporcioandos no forman un triangulo";
        else
            return info();
    }

}
```

4.1.3. Ejemplo Cuadrática

Para una ecuación cuadrática de la forma $f(x) = Ax^2 + Bx + C$, calcular los de x que hacen $f(x) = 0$ utilizando la formula general

$$x = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A}$$

Comenzamos por escribir una clase para manejar los números complejos de acuerdo con el siguiente diagrama UML

Complejo
- Real : Double - Imag : Double
«constructor» Complejo(Double, Double) + toString() : String

El código correspondiente para los complejos es:

```
package Capitulo_4.Ecuacion_cuadratica;

/**
 *
 * @author calderon
 */
public class Complejo {
    private double Real;
    private double Imag;

    public Complejo(double unR, double unC) {
        Real = unR;
        Imag = unC;
    }

    @Override
    public String toString() {
        String aux;

        if(Imag > 0)
            aux = Real + " + " + Imag + " j \n";
        else if (Imag ==0)
            aux = Real + "\n";
        else
            aux = Real + " - " + Imag*(-1.0) + " j \n";
        return aux;
    }
```

```
}
```

El Diagrama UML para la clase Cuadrática es:

Cuadratica
- A : Double - B : Double - C : Double - x1 : Complejo - x2 : Complejo
«constructor» Cuadratica(Double, Double, Double) + Solucion() : void + toString() : String

La solución en Java queda

```
package Capitulo_4.Ecuacion_cuadratica;

/**
 *
 * @author calderon
 */
public class Cuadratica {
    private double A, B, C;
    private Complejo x1, x2;

    public Cuadratica(double unaA, double unaB, double unaC) {
        A = unaA;
        B = unaB;
        C = unaC;
        Solucion();
    }

    public void Solucion() {
        double D;

        D = B*B - 4.0*A*C;

        if(D>0) {
            x1 = new Complejo((-B + Math.sqrt(D))/(2.0*A), 0);
            x2 = new Complejo((-B - Math.sqrt(D))/(2.0*A), 0);
        }
```

```
        else {
            x1 = new Complejo(-B /(2.0*A), +Math.sqrt(-D)/(2.0*A));
            x2 = new Complejo(-B /(2.0*A), -Math.sqrt(-D)/(2.0*A));

        }
    }

    @Override
    public String toString() {
        String aux;

        aux  = x1.toString() + x2.toString();

        return aux;
    }

}
```

4.1.4. Ejemplo Caída Libre

Un objeto es lanzado con una velocidad inicial $v_i = 20$ m/s hacia arriba. Determinar a) La altura máxima a la que llega y b) los tiempos en que el objeto esta a la mitad de la altura máxima.

Para este ejemplo utilizaremos las clases Complejo y Cuadratica del ejemplo anterior las cuales se introduces como tipos de objetos de la clase. En el siguiente diagrama UML se da la descripción de esta implementación.

CaidaLibre
- vi : Double - y0 : Double - g : Double
«constructor» CaidaLibre(Double, Double) + hmax() : Double + tiempos() : String + toString() : String

La implementación en Java queda de la siguiente manera

```
package Capitulo_4.CaidaLibre;

/**
```

```
 *
 * @author calderon
 */
public class CaidaLibre {
    private double vi, y0, g;

    public CaidaLibre(double unaY, double unaV) {
        vi = unaV;
        y0 = unaY;
        g = 9.8;
    }

    public double hmax() {
        return vi*vi/2.0/g;
    }

    public String tiempos() {
        Cuadratica a = new Cuadratica(-0.5*g, vi, y0 - 0.5*hmax());
        return a.toString();
    }

    public String toString() {
        String aux = "";

        aux += "Para un objeto que se lanza desde una altura de " + y0 + " mts\n";
        aux += "con una velocidad inicial de v0 = " + vi + " mts/seg \n";
        aux += "\n";
        aux += "La altura maxima que alcanza es " + hmax();
        aux += "\n";
        aux += "Los tiempos en los cuales esta a la mitad de la altura maxima son \n";
        aux += tiempos();
        aux += "\n";

        return aux;
    }
}
```

4.1.5. Ejemplo Suma de dígitos

Determinar si la suma de un número de 4 digitos es igual a 21

```
import java.util.Scanner;

public class suma_21 {
    static public void main(String args[]) {
        int n1, n2, n3, n4, suma, n, n0;
        System.out.printf("Dame un numero entero ");
        Scanner entrada = new Scanner(System.in);
        n0= entrada.nextInt();

        n = n0%10000;     // se queda solamente con la ultimas 4 cifras de n

        n1 = n/1000;     // calcula n1 como la division entera
        n %= 1000;       // se queda solamente con las ultimas 3 cifras de n

        n2 = n/100;      // calcula n2 como la division entera
        n %= 100;        // se queda solamente con las ultimas 2 cifras de n

        n3 = n/10;       // calcula n3 como la division entera
        n4 = n%10;       // se queda con el utimo digito

        suma = n1+n2+n3+n4;
        if(suma == 21) System.out.printf("La suma es igual a 21\n");
        else System.out.printf("Sigue intentando la suma es %d\n", suma);
    }
}
```

4.1.6. Operador Condicional

Existe una construcción adicional, que permite comprobar si se cumple una condición y devolver un cierto valor según si dicha condición se cumple o no. Es lo que se conoce como el "operador condicional (?)":

```
    condicion ? resultado_si cierto : resultado_si_falso
```

Es decir, se indica la condición seguida por una interrogación, después el valor que hay que devolver si se cumple la condición, a continuación un símbolo de "dos puntos" y finalmente el resultado que hay que devolver si no se cumple la condición.

Es frecuente emplearlo en asignaciones (aunque algunos autores desaconsejan su uso porque puede resultar menos legible que un "if"), como en este ejemplo:

x = (a == 10) ? b*2 : a ;

En este caso, si a vale 10, la variable tomará el valor de b*2, y en caso contrario tomará el valor de a. Esto también se podría haber escrito de la siguiente forma, más larga pero más legible:

```
    if (a == 10)  x = b*2; else x = a;
```

El siguiente es un ejemplo donde se utilizan las expresiones condicionales.

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

        int x =1, y;

        /*
         * La condicional
        */

        if(x==1) y = 20;
        else y = 10;

        System.out.printf("y = %d\n", y);

        /*
         * se puede escribir como
         */

        y=0;

        y = (x==1) ? 20 : 10;
        System.out.printf("y = %d\n", y);

        /*
         *Otro ejemplo es
         */

        if(x==1) System.out.printf("en Auto");
        else System.out.printf("en bici");

        /**
         *Utilizando expresiones
         */
```

```
        System.out.printf((x==1) ? "en Auto" : "en bici");
    }
}
```

4.2. Comparando datos

Si comparamos tipos primitivos con los operadores == o != el resultado es el esperado. Sin embargo, si comparamos objetos con estos dos operadores el resultado puede no ser el esperado. Tenemos el ejemplo siguiente:

```
public class Equivalencia {
   public static void main(String[] args) {
     Integer n1 = new Integer(47);
     Integer n2 = new Integer(47);

     System.out.println("n1==n2: "+ (n1 == n2));
     System.out.println("n1!=n2: "+ (n1 != n2));

     String s1 = new String("Hola");
     String s2 = new String("Hola");

     System.out.println("s1==s2: "+ (s1 == s2));
     System.out.println("s1!=s2: "+ (s1 != s2));
 }
}
```

Si lo ejecutamos el resultado es:

```
n1==n2: false
n1!=n2: true
s1==s2: false
s1!=s2: true
```

Esto se debe a que los operadores == y != comparan referencias, no valores. La referencia de n1 está almacenada en un lugar de la memoria y la de n2 en otra, lo mismo para s1 y s2. Recuerda que cada vez que se crea un objeto con new se le asigna un espacio de memoria en el cúmulo.

Para comparar objetos, no tipos primitivos, tenemos el método equals(), que se puede utilizar siempre, ya que está incluido en la clase java.lang.Object que por defecto heredan todas las clases. Este método compara valores en lugar de referencias, modificamos un poco el ejemplo anterior y utilizamos equals():

```
public class Equivalencia2 {
   public static void main(String[] args) {
     Integer n1 = new Integer(47);
     Integer n2 = new Integer(47);

     System.out.println("n1.equals(n2): "+n1.equals(n2));

     String s1 = new String("Hola");
     String s2 = new String("Hola");

     System.out.println("s1.equals(s2): "+s1.equals(s2));
   }
}
```

El resultado es:

```
n1.equals(n2): true
s1.equals(s2): true
```

Otro método para hacer comparación entre objetos es utilizar el comando compareTo, el cual regresa un número entero bajo las siguiente condiciones

n1.CompareTo(n2)	→ Regresa -1 si n1 es menor que n2
n1.CompareTo(n2)	→ Regresa 0 si n1 es igual que n2
n1.CompareTo(n2)	→ Regresa 1 si n1 es mayor que n2

Como ejemplo tenemos

```
public class Equivalencia3 {
   public static void main(String[] args) {
     Float n1 = new Float(1.0);
     Float n2 = new Float(2.0);
     Float n3 = new Float(3.0);

     System.out.println(n1.floatValue()+ ", "
     +n2.floatValue() + " " + n1.compareTo(n2));
     System.out.println(n2.floatValue()+ ", " +n1.floatValue()
     + " " + n2.compareTo(n1));
     System.out.println(n2.floatValue()+ ", " +n2.floatValue()
     + " " + n2.compareTo(n2));
   }
}
```

Estos métodos están definidos en las clases wrapper de los tipos básicos, pero es necesario

hacer la extensión en el caso de tipos objetos nuevos.

4.2.1. Sobre escritura del método equal()

Todos los objetos declarados en Java, por default tienen un método llamado equal(). Este método puede ser sobre escrito para hacer comparaciones entre objetos del mismo tipo.

Para mayor claridad vamos a escribir un objeto que represente una Fracción, cuyo diagrama UML es:

Fraccion
- numerador : Double - denominador : Double
«constructor» Fraccion(Integer, Integer) + equals(Object): boolean + toString() : String

La escritura de método equal() queda

```
package Capitulo_4.Fraccion_old;

/**
 *
 * @author felix
 */
public class Fraccion {
    private int numerador;
    private int denominador;

    public Fraccion(int unN, int unD) {
        numerador = unN;
        denominador = unD;
    }

    @Override
    public String toString(){
        String signo;
        signo = (numerador/denominador < 0) ? "-" : "+";
        return signo + Math.abs(numerador) + "/"
                + Math.abs(denominador);
    }
```

```
    @Override
    public boolean equals(Object  b) {
        if(b instanceof Fraccion) {

            Fraccion aux = (Fraccion)b;

            return denominador == aux.denominador &&
                numerador == aux.numerador;
        }
        return false;
    }
}
```

La manera de probar el código es

```
package Capitulo_4.Fraccion_old;

/**
 *
 * @author felix
 */
public class Prueba {
    static public void main(String args[]) {
        Fraccion a = new Fraccion(-3, 4);
        Fraccion b = new Fraccion(3, 2);
        Fraccion c = new Fraccion(-3, 4);

        System.out.println(a + " es igual a " + b + " " + a.equals(b));
        System.out.println(a + " es igual a " + c + " " + a.equals(c));

    }
}
```

Cuya ejecución da como resultado

```
+3/4 es igual a +3/2 false
+3/4 es igual a +3/4 true
```

4.3. Sentencia switch

Si queremos comprobar varias condiciones, podemos utilizar varios if - else - if - else - if encadenados, pero tenemos una forma más elegante en Java de elegir entre distintos valores posibles que tome una cierta expresión. Su formato es :

```
  switch (expresion) {
    case valor1: sentencias1; break;
    case valor2: sentencias2; break;
    case valor3: sentencias3; break;
    ...                                  // Puede haber más valores
}
```

4.3.1. Ejemplo ultimo dígito

Como ejemplo podemos ver la siguiente implementación en C, que permite calcular cual es el último dígito de un número dado

```
public class ejemplo_switch {
    static public void main(String args[]) {
        int d = 243;

        switch(d%10) {
            case 0: System.out.printf("cero ");   break;
            case 1: System.out.printf("uno ");    break;
            case 2: System.out.printf("dos ");    break;
            case 3: System.out.printf("tres ");   break;
            case 4: System.out.printf("cuatro "); break;
            case 5: System.out.printf("cinco ");  break;
            case 6: System.out.printf("seis ");   break;
            case 7: System.out.printf("siete ");  break;
            case 8: System.out.printf("ocho ");   break;
            case 9: System.out.printf("nueve ");  break;
            default : ;
        }

        System.out.printf("Unidades\n");
    }
}
```

Es decir, después de la orden switch indicamos entre paréntesis la expresión que queremos

evaluar. Después, tenemos distintos apartados, uno para cada valor que queramos comprobar; cada uno de estos apartados se precede con la palabra case, indica los pasos a dar si es ese valor el que se da, y terminar con break.

Un ejemplo sería:

```
  switch ( x * 10) {
    case 30: printf( "El valor de x era 3" ); break;
    case 50: printf( "El valor de x era 5" ); break;
    case 60: printf( "El valor de x era 6" ); break;
}
```

también podemos indicar qué queremos que ocurra en el caso de que el valor de la expresión no sea ninguno de los que hemos detallado:

```
  switch (expresion) {
    case valor1: sentencias1; break;
    case valor2: sentencias2; break;
    case valor3: sentencias3; break;
    ...                                  // Puede haber más valores
    default: sentencias;                 // Opcional: valor por defecto
}
```

Por ejemplo, así:

```
  switch ( x * 10) {
    case 30: printf( "El valor de x era 3" ); break;
    case 50: printf( "El valor de x era 5" ); break;
    case 60: printf( "El valor de x era 6" ); break;
    default: printf( "El valor de x no era 3, 5 ni 6" ); break;
}
```

Podemos conseguir que en varios casos se den los mismos pasos, simplemente eliminado la orden “break” de algunos de ellos. Así, un ejemplo algo más completo podría ser:

```
  switch ( x ) {
    case 1:
    case 2:
    case 3:
        System.out.printf( "El valor de x estaba entre 1 y 3" );
        break;
    case 4:
    case 5:
        System.out.printf( "El valor de x era 4 o 5" );
        break;
```

```
    case 6:
        System.out.printf( "El valor de x era 6" );
        valorTemporal = 10;
        System.out.printf( "Operaciones auxiliares completadas" );
        break;
    default:
        System.out.printf( "El valor de x no estaba entre 1 y 6" );
        break;
}
```

4.3.2. Ejemplo rango letras

El siguiente ejemplo permite calcular, dada una letra en que rango se encuentra.

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

        char c = 'a';

        if(c >= 'A' && c <= 'Z') c += 32;

        switch(c) {
            case 'a' :
            case 'b' :
            case 'c' :
            case 'd' : System.out.printf("La letra esta entre A y D\n"); break;
            case 'e' :
            case 'f' :
            case 'g' :
            case 'h' : System.out.printf("La letra esta entre E y H\n"); break;
            case 'i' :
            case 'j' :
            case 'k' :
            case 'l' : System.out.printf("La letra esra ebtre I y L\n"); break;
            case 'm' :
            case 'n' :
            case 'o' :
            case 'p' : System.out.printf("La letra esra ebtre M y P\n"); break;
            case 'q' :
            case 'r' :
            case 's' :
```

```
            case 't' : System.out.printf("La letra esra ebtre Q y T\n"); break;
            case 'u' :
            case 'v' :
            case 'w' :
            case 'x' : System.out.printf("La letra esra ebtre U y X\n"); break;
            case 'y' :
            case 'z' : System.out.printf("La letra esra ebtre X y Z\n"); break;
            default : System.out.printf("No es un caracter valido\n");
        }

    }
}
```

Un poco más adelante propondremos ejercicios que permitan afianzar todos estos conceptos.

4.4. Sentencia while

4.4.1. while

Java incorpora varias formas de conseguirlo. La primera que veremos es la orden while, que hace que una parte del programa se repita mientras se cumpla una cierta condición. Su formato será:

```
while (condición) {
    sentencia 1;
    sentencia 2;
    ...
 }
```

Es decir, la sintaxis es similar a la de if, con la diferencia de que aquella orden realizaba la sentencia indicada una vez como máximo (si se cumple la condición), pero while puede repetir la sentencia más de una vez (mientras la condición sea cierta). Al igual que ocurría con if, podemos realizar varias sentencias seguidas (dar más de un paso) si las encerramos entre llaves:

```
x = 20;
while ( x > 10) {
  System.out.printf("Aun no se ha alcanzado el valor limite\n");
  x --;
}
```

4.4.2. do-while

Existe una variante de este tipo de bucle. Es el conjunto do..while, cuyo formato es:

```
do {
  sentencia1;
  sentencia2;
  ...
} while (condición)
```

En este caso, la condición se comprueba al final, lo que quiere decir que las "sentencias" intermedias se realizarán al menos una vez, cosa que no ocurría en la construcción anterior (un único while antes de las sentencias), porque si la condición era falsa desde un principio, los pasos que se indicaban a continuación de while no llegaban a darse ni una sola vez.

Un ejemplo típico de esta construcción "do..while" es cuando queremos que el usuario introduzca una contraseña que le permitirá acceder a una cierta información:

```
do {
  System.out.printf("Introduzca su clave de acceso\n");
  lee_teclado("%f", &claveIntentada);  // LeerDatosUsuario realmente no existe
} while (claveIntentada != claveCorrecta)
```

En este ejemplo hemos la función scanf para leer un número flotante.

4.5. Sentencias do y for

Una tercera forma de conseguir que parte de nuestro programa se repita es la orden for. La emplearemos sobre todo para conseguir un número concreto de repeticiones. Su formato es

```
for ( valor_inicial ; condicion_continuacion ; incremento ) {
  sentencias1;
  sentencias2;
  ...
}
```

(es decir, indicamos entre paréntesis, y separadas por puntos y coma, tres Ordenes: la primera dará el valor inicial a una variable que sirva de control; la segunda orden será la condición que se debe cumplir para que se repitan las sentencias; la tercera orden será la que se encargue de aumentar -o disminuir- el valor de la variable, para que cada vez quede un paso menos por dar).

Esto se verá mejor con un ejemplo. podríamos repetir 10 veces un bloque de Ordenes haciendo:

```
for ( i=1 ; i<=10 ; i++ ) {
  ...
}
```

O bien podríamos contar descendiendo desde el 20 hasta el 2, con saltos de 2 unidades en 2 unidades, así:

```
for ( j = 20 ; j > 0 ; j -= 2 )
  System.out.printf( "%d\n", j );
```

Nota: se puede observar una equivalencia casi inmediata entre la orden for y la orden while. Así, el ejemplo anterior se podría reescribir empleando while, de esta manera:

```
j = 20;
while ( j > 0 )
  System.out.printf( "%d\n", j );
  j -= 2;
}
```

Precauciones con los bucles: Casi siempre, nos interesará que una parte de nuestro programa se repita varias veces (o muchas veces), pero no indefinidamente. Si planteamos mal la condición de salida, nuestro programa se puede quedar colgado, repitiendo sin fin los mismos pasos.

Se puede modificar un poco el comportamiento de estos bucles con las ordenes break y continue.

La sentencia break hace que se salten las instrucciones del bucle que quedan por realizar, y se salga del bucle inmediatamente. Como ejemplo:

```
for ( i=1 ; i<=10 ; i++ ) {
  System.out.printf( "Comenzada la vuelta " );
  printf( "%d\n", i );
  if (i==8)  break;
  System.out.printf( "Terminada esta vuelta\n" );
}
System.out.printf( "Terminado\n"  );
```

En este caso, no se mostraría el texto Terminada esta vuelta para la pasada con i=8, ni se darían las pasadas de i=9 e i=10, porque ya se ha abandonado el bucle.

La sentencia continue hace que se salten las instrucciones del bucle que quedan por realizar, pero no se sale del bucle sino que se pasa a la siguiente iteración (la siguiente vuelta o

pasada). Como ejemplo:

```
for ( i=1 ; i<=10 ; i++ ) {
  System.out.printf( "Comenzada la vuelta\n" );
  System.out.printf( "%d\n", i );
  if (i==8)  continue;
  System.out.printf( "Terminada esta vuelta\n" );
}
printf( "Terminado\n"  );
```

En este caso, no se mostraría el texto Terminada esta vuelta para la pasada con i=8, pero si se darían la pasada de i=9 y la de i=10.

también esta la posibilidad de usar una etiqueta para indicar donde se quiere saltar con break o continue. Solo se debería utilizar cuando tengamos un bucle que a su vez está dentro de otro bucle, y queramos salir de golpe de ambos. Es un caso poco frecuente, así que no profundizaremos más, pero si veremos un ejemplo:

```
for ( i=1 ; i<=10 ; i++ ) {
  System.out.printf( "Comenzada la vuelta\n" );
  System.out.printf( "%d\n", i );
  if (i==8)  break salida;
  System.out.printf( "Terminada esta vuelta\n" );
}
System.out.printf( "Terminado\n"  );
salida:
...
```

En este caso, a mitad de la pasada 8 se saltaría hasta la posición que hemos etiquetado como salida (se define como se ve en el ejemplo, terminada con el símbolo de dos puntos), de modo que no se escribiría en pantalla el texto Terminado (lo hemos saltado).

4.5.1. Ejemplo Sumatoria

Hacer un programa que permita calcular la siguiente sumatoria

$$S = \sum_{i=0}^{i=64} 2^i$$

La implementación en Java queda:

```
public class sumatoria {
```

```
    static public void main (String args[])
    {
      int i=1, s=0;

      // utilizando for()

      for(i=0; i<=30; i++){
        s=((2*s) +1);
        System.out.printf("%d s=%d\n", i, s);
      }

      // utilizando while

      i = 1;
      s = 0;
      while(i <=30)
      {
        s=((2*s) +1);
        System.out.printf("%d s=%d\n", i, s);
        i++;
      }

      // utilizando do-while

      i = 1;
      s = 0;
      do
      {
        s=((2*s) +1);
        System.out.printf("%d s=%d\n", i, s);
        i++;
      }
      while (i<=30);
    }
}
```

4.5.2. Ejemplo comparación for, while, do-while

El siguiente ejemplo permite hacer una comparación entre la implementación utilizando for y do-while para leer una clave desde teclado.

```
import java.util.Scanner;

public class clave {
    static public void main(String args[]) {
        int clave;
        Scanner entrada = new Scanner(System.in);
        // impelementacion utilizando for

        for(;;){
            System.out.printf("Dame tu clave ");
            clave = entrada.nextInt();

            if(clave == 1234) break;
        }

        // implementacion con do-while

        do {
            System.out.printf("Remita su clave nuevamente ");
            clave = entrada.nextInt();
        } while (clave != 1234);

        System.out.printf("Bienvenido\n");

    }
}
```

4.5.3. Ejemplo Combinaciones

Escribir un programa que permita imprimir todas las combinaciones de palabra de tres letras que se pueden escribir utilizando las letras del alfabeto. Así por ejemplo la salida de este código es

AAA AAB AAC ... ABA ABC

etc.

```
public class combinaciones {
    static public void main(String args[]) {
        int i, j, k, n=1;
        int letra = 'A';
```

```
        for(i=0; i<3; i++){
            for(j=0; j<3; j++){
                for(k=0; k<3; k++)
                    System.out.printf("%02d.- %c%c%c \n",
                     n++, letra +i, letra+j, letra +k);
            }
        }
    }
}
```

4.5.4. Ejemplo función

Escribir un programa que permita hacer la siguiente tabulación correspondiente a la función $f(x) = x^2$

x	$f(x)$
1	1
2	4
3	9
4	16

```
public class funcion {
    static public void main(String args[])
    {
      int x;

      for(x=1; x<=10; x++)
        System.out.printf("%d^2 = %f \n", x, cuadrado(x));
    }

    // Definicion  de la funcion

    static public float cuadrado(float y)
    {
      return y*y;
    }
}
```

4.5.5. Ejemplo triángulo impreso

Escribir un programa que permita imprimir un triángulo como el que se muestra a continuación

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

```
public class triangulo {
    static public void main(String args[]) {
        int i, j;

        for(i=0; i<20; i++) {
            for(j=0; j<i; j++)
                System.out.printf("*");
            System.out.printf("\n");
        }

    }
}
```

4.6. Iteradores

Los Iteradores son un patrón de diseño utilizado para recorrer las colecciones. En java es una interfaz denominada Iterator. Esta formado por 3 métodos:

- boolean hasNext(): retorna true en caso de haber mas elementos y false en caso de llegar al final de iterator

- Object next(): retorna el siguiente elemento en la iteración

- void remove(): remueve el ultimo elemento devuelto por la iteración

Una manera de utilizarlos es mediante un arreglo de datos tal como se muestra en el código donde se declara un arreglo de enteros con la variable val y todos los valores son recorridos por la sentencia “for(int i: val)”.

```
    static public void ejemplo03(){

        int val[] = new int[] {1,2,3,4,5,6,7,8,9,10};

        for(int i: val)
            System.out.println(i);


    }
```

Con un equivalente utilizando ciclos dado por

```
    static public void ejemplo03(){

        int val[] = new int[] {1,2,3,4,5,6,7,8,9,10};

        for(int i =0; i<9; i++)
            System.out.println(val[i]);


    }
```

En las clases Vector, ArrayList, HashSet y TreeSet un iterador se consigue a través del método: Iterator iterator(). El siguiente es un ejemplo de los iteradores utilizados en las clases ArrayList y Vector

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.Vector;

public class Iteradores {
    static public void ejemplo01() {
        ArrayList a = new ArrayList();
        Object x;

        a.add(new Integer(3));
        a.add(new Double(4));
        a.add(new Character('x'));
        a.add(new String("Hola"));

        for (Iterator it = a.iterator(); it.hasNext();){
            x = it.next();
            System.out.println(x);
```

```
        }

        for(Object b: a)
            System.out.println(b);
    }

    static public void ejemplo02() {
        Vector a = new Vector();
        Object x;

        a.add(new Integer(3));
        a.add(new Double(4));
        a.add(new Character('x'));
        a.add(new String("Hola"));

        for (Iterator it = a.iterator(); it.hasNext();){
            x = it.next();
            System.out.println(x);
        }

        for(Object b: a)
            System.out.println(b);
    }

    static public void main(String args[]) {
        ejemplo02();
    }
}
```

4.6.1. Ejemplo Agenda

Escribir un código para crear una agenda. La agenda tendrá información del nombre de la persona, dirección y Teléfono.

Para comenzar definiremos un objeto llamado Persona con los atributos que se mencionan y con los métodos que se muestran en el siguiente diagrama UML

P e r s o n a
- Nombre : String - ApellidoPaterno : String - ApellidoMaterno : String - Domicilio : String - Telefono : String
«constructor» Persona(String, String, String, String, String) + toString() : String + toequals(Object) : boolean

La codificación en Java del Objeto Persona es:

```
package Capitulo_4.Agenda;

public class Persona {
    private String Nombre;
    private String ApellidoPaterno;
    private String ApellidoMaterno;
    private String Domicilio;
    private String Telefono;

    public Persona(String unNombre, String unPaterno, String unMaterno,
            String unDomicilio, String unTelefono) {
        Nombre = unNombre;
        ApellidoPaterno = unPaterno;
        ApellidoMaterno = unMaterno;
        Domicilio = unDomicilio;
        Telefono = unTelefono;
    }

    @Override
    public String toString() {
        String aux;

        aux = "Nombre    : " + Nombre +
              " " + ApellidoPaterno +
              " " + ApellidoMaterno + "\n" +
              "Domicilio : " + Domicilio + "\n"+
              "Telefono  : " + Telefono + "\n\n";

        return aux;
    }
```

```
    @Override
    public boolean equals(Object O){
        Persona a = (Persona) O;

        return (this.Nombre.equals(a.Nombre) ||
                this.ApellidoPaterno.equals(a.ApellidoPaterno) ||
                this.ApellidoMaterno.equals(a.ApellidoMaterno));
    }
}
```

El diagrama UML para el objeto Agenda es:

A g e n d a
- datos : ArrayList <Persona>
«constructor» Agenda() + agrega(String, String, String, String, String) + busca(String) : String + CreaAgenda() : void + toString(): String

La implementación para el objeto Agenda en Java es:

```
package Capitulo_4.Agenda;

import java.util.ArrayList;
import java.util.Iterator;

public class Agenda {
    private ArrayList<Persona> datos;

    public Agenda() {
        datos = new ArrayList();
        CreaAgenda();
    }

    public void CreaAgenda() {
        agrega("Felix", "Calderon", "Solorio", "Morelia", "xxxxxx");
        agrega("Juan", "Lopez", "Dias", "Un calle", "un telefono");
    }

    public void agrega(String unNombre, String unPaterno,
    String unMaterno, String unDomicilio, String unTelefono) {
```

```
        Persona a = new Persona(unNombre, unPaterno,
        unMaterno, unDomicilio, unTelefono);
        datos.add(a);
    }

    public String busca(String nombre) {
        Persona aux = new Persona(nombre, nombre, nombre, "" , "");

        for(Persona a:datos) {
            if(a.equals(aux)) return a.toString();
        }

        return "Persona no encontrada\n";
    }

    @Override
    public String toString() {
        String aux = "";
        Persona x;
        Iterator it;
        for (it = datos.iterator(); it.hasNext();){
            x = (Persona) it.next();
            aux += x;
        }
        return aux;
    }
}
```

Finalmente para la ejecución escribimos la clase a la que generalmente llamamos prueba.

```
package Capitulo_4.Agenda;

public class Prueba {
    static public void main(String args[]){
        Agenda mi_agenda = new Agenda();
        System.out.println(mi_agenda);


        System.out.println(mi_agenda.busca("Felix"));
    }
}
```

4.7. Recursividad

Un método recursivo es un método que se llama a si mismo directa o indirectamente o a través de otro método. Un ejemplo interesante de recursion es la función factorial. La definición de factorial esta dada por el producto:

$$n! = n * (n-1) * (n-2) * ... * 3 * 2 * 1$$

Note que esta definición la podemos escribir de manera recursiva como:

$$n! = n * (n-1)!$$

En común representar a las funciones recursivas en dos pasos. El primero de ellos es conocido como Paso Base y Paso Recursivo. El primero es la condición más sencilla de la definición y normalmente es trivial hacer el cálculo y el segundo es el paso recursivo. Para el caso del Factorial tenemos

$factorial(N)$

Base

Si $N = 1$ el factorial es 1

Paso Recusivo

Si no $n \times factorial(N-1)$

Este método lo podemos escribir en Java como

```
public class Factorial {

    static public void main(String args[]) {
        int n = 10;
        System.out.printf("Factorial de %d es %d\n", n, factorial(n) );
    }

    static public int factorial(int N){
        if(N==1) return 1;
        else return N*factorial(N-1);
    }
```

Otro ejemplo de definición recursiva es la multiplicación de números naturales. El producto a*b, donde a y b son enteros positivos, puede definirse como

$Multiplica(x, y)$

Base

Si $y = 1$ el producto es y

Paso Recusivo

Si no el resultado es $x + Multiplica(x, y - 1)$

```
public class Multiplica {

    static public void main(String args[])
    {
      int x=4, y=5;
      System.out.printf("%d\n", multiplica(4,5));
    }

    static public int multiplica(int x, int y)
    {
       if(y == 1) return x;
       else return (x + multiplica(x,y-1));
    }
}
```

4.7.1. Torres de Hanoi

El ejemplo más conocido, de recursividad entre los programadores, es el problema de las torres de Hanoi. Para este juego, se tienen tres postes (A, B, C) y un conjunto n de discos colocados de mayor a menor diámetro en alguno de los postes, por ejemplo el poste A. La meta es mover todos los discos del poste A al poste C con las siguientes restricciones:

1.- En cada movimiento solo se puede tomar un disco y colocarlo en cualquier poste y 2.- No se debe colocar, un en un poste, un disco de diámetro mayor sobre un disco de diámetro menor.

La definición recursiva de las torres de Hanoi es:

$mover_discos(origen, auxiliar, destino, discos)$

Base

Si $discos = 1$ Mover un disco de *origen* a *destino*

Paso recursivo

Mover $n-1$ discos de *origen* a *auxiliar*

Mover 1 disco de *origen* a *destino*

Mover $n-1$ discos de *auxiliar* a *destino*

La implementación en C queda

```
public class Torres_Hanoi {

    static int movtos ;

    static public void main(String args[])
    {
       movtos = 1;
       mover_discos('A', 'B', 'C', 4);
    }

    static public void mover_discos(char a, char b, char c, int n)
    {
        if (n > 0) {
            mover_discos(a, c, b, n - 1);
            System.out.printf("%d Mover un disco del poste
            %c al poste %c\n", movtos++, a, c);
            mover_discos(b, a, c, n - 1);
        }
    }
}
```

La solución con tres discos es:

```
1 Mover un disco del poste A al poste C
2 Mover un disco del poste A al poste B
3 Mover un disco del poste C al poste B
4 Mover un disco del poste A al poste C
5 Mover un disco del poste B al poste A
6 Mover un disco del poste B al poste C
7 Mover un disco del poste A al poste C
```

4.7.2. Método de Bisecciones

Este método es conocido también como de corte binario. de partición en dos intervalos iguales o método de Bolzano. Es un método de búsqueda incremental donde el intervalo

de búsqueda se divide en dos. Si la función cambia de signo sobre un intervalo, se calcula el valor en el punto medio. El nuevo intervalo de búsqueda será aquel donde el producto de la función cambie de signo.

La definición recursiva para este algoritmos es:

Dada una función $f(x)$ y un intervalo $[inicio, fin]$ de búsqueda donde existe un cruce por cero, podemos notar que $f(a) \times f(b) < 0$ porque la función en el intervalo pasa de menos a mas o de mas a menos debido a que cruza por cero. Así tenemos:

$Biseccion(inicio, fin)$

Base

Si $b - a < \epsilon$ la solución es $(a + b)/2$

Paso Recusivo

Calcular $mitad = \frac{inicio+fin}{2}$

Si $f(inicio) \times f(mitad) < 0$ entonces $Biseccion(inicio, mitad)$

Si no $Biseccion(mitad, fin)$ La implementación recursiva de este algoritmo en Java es:

```
public class Bisecciones {
    static public void main(String args[])
    {
      System.out.printf("La solucion esta en %f\n", Biseccion(0,1));
    }

    static public double Biseccion(double ini, double fin)
    {
      double mitad;    mitad = (fin + ini)/2.0;
      System.out.printf("f_1(%f) = %f f_m(%f) = %f f_2(%f) = %f\n",
              ini, funcion(ini), mitad, funcion(mitad), fin, funcion(fin));
      if((fin - ini) > 1e-12)
      {
        if(funcion(ini)*funcion(mitad) < 0)
          return Biseccion(ini, mitad);
        else return Biseccion(mitad, fin);
      }
      else return (mitad);
    }

    static public double funcion(double x)
    {
```

```
        return (x - Math.cos(x));
    }
}
```

Ejemplo

Considere la función $f(x) = x - cosx$, a priori sabemos que la función tiene un cruce por cero en el intervalo $[0, 1]$, así que nuestra búsqueda se concentrará en este.

iter	inicio	mitad	fin	f(ini)	f(mitad)	f(fin)
0	0.0	0.5	1.0	-1.0	-0.3775	0.4596
1	0.5	0.75	1.0	-0.3775	0.0183	0.4596
2	0.5	0.625	0.75	-0.3775	-0.1859	0.0183
3	0.625	0.6875	0.75	-0.1859	-0.0853	0.0183
4	0.6875	0.71875	0.75	-0.0853	-0.0338	0.0183
5	0.71875	0.734375	0.75	-0.0338	-0.0078	0.0183
6	0.734375	0.7421875	0.75	-0.0078	0.0051	0.0183
7	0.734375	0.73828125	0.7421875	-0.0078	-0.0013	0.0051
8	0.73828125	0.740234375	0.7421875	-0.0013	0.0019	0.0051
9	0.73828125	0.7392578125	0.740234375	-0.0013	0.0002	0.0019

4.8. Assertions

Definimos una aserción o acertó como una expresión que siempre es verdadera y es inadmisible que sea diferente. En Java cuando queremos proteger un código para que no se den valores fuera de un rango utilizamos el comando assert, el cual tiene la siguiente sintaxis

assert Expresion1; assert Expresión1:Expresión2;

En cualquiera de los dos casos Expresión1 tiene que ser una expresión booleana o se producirá un error de compilación. Cuando se evalúa un aserto que solo tenga Expresión1, se comprueba la veracidad de la expresión y si es verdadera se continúa la ejecución del programa, pero si es falsa, se lanza una excepción de tipo AssertionError. Si el aserto contiene además una Expresión2 y Expresión1 es falsa, se evalúa Expresion2 y se le pasa como parámetro al constructor del AssertionError (Si Expresion1 se evalúa como cierto, la segunda expresión no se evalúa).

A continuación se muestra un ejemplo del uso de aserciones para limitar a que un código solo acepte números entre 0 y 10.

```
import java.util.Scanner;

public class Aserciones
{
     public static void main( String args[] )
     {
        Scanner input = new Scanner( System.in );

        System.out.print("Dame un numero entre 0 y 10: " );

        int numero = input.nextInt();

        // genera un mensaje de error si el numero no esta en rango
        assert ( numero >= 0 && numero <= 10 ) : "Numero incorrecto: " + numero;

        System.out.println( "Me diste el numero "+ numero );
     }
}
```

Para que se activen las aserciones debemos correr el código dando la bandera -ea, el siguiente es una corrida del código desde linea de comando.

```
macbook-pro-de-felix-calderon-2:Capitulo_4 felix$ java -ea Aserciones
Enter a number between 0 and 10: 11
Exception in thread "main" java.lang.AssertionError: bad number: 11
at Aserciones.main(Aserciones.java:15)
macbook-pro-de-felix-calderon-2:Capitulo_4 felix$
```

En el caso de utilizar Netbeans o Eclipse se debe buscar las opciones de la Máquina virtual de Java y dar la bandera -ea

4.9. Introducción a Excepciones

Una excepción es un evento, que se produce durante la ejecución de un programa, que interrumpe el flujo normal de las instrucciones del programa. Así cuando se produce un error dentro de un método, el método crea un objeto y se lo entrega libre al sistema de ejecución. El objeto, llamado un objeto de excepción, contiene información sobre el error, incluyendo su tipo y el estado del programa cuando se produjo el error. Creación de un objeto de excepción y se la entregó al sistema de ejecución, esto llamado lanzamiento de una excepción.

El primer paso en la construcción de un manejador de excepciones es encerrar el código que podría lanzar una excepción dentro de un bloque try. En general, un bloque try tiene el siguiente aspecto:

```
try {
    codigo ...
}
catch finaliza bloque . .
```

A continuación se muestra un ejemplo donde se puede notar el uso de excepciones. En el ejemplo sin excepciones se asigna una cadena de texto a una variable llamada cadena, note que tiene espacios en blanco lo cual no es un número.

```
public class Excepciones {
    static public void ejemplo_sin_excepciones() {
        int numero;
        String cadena="  1";
        numero = Integer.parseInt(cadena);
    }

    static public void ejemplo_con_excepciones() {
        int numero;
        String cadena = "  1";
        try{
            numero = Integer.parseInt(cadena);
        }
        catch(NumberFormatException ex){
            System.out.println("No es un numero, es una cadena de texto.");
        }
    }

    static public void Division_por_cero() {
        int d, a;

        try {
            d = 0;
            a = 42 / d;
            System.out.println("Esto no se imprimira");
        }
        catch (ArithmeticException e) {// captura el error de division
            System.out.println("Division por cero.");
        }
```

```
        System.out.println("Termino la ejecucion");
    }

    public static void main(String args[]){
        ejemplo_con_excepciones();
    }
}
```

Si corremos el primer ejemplo sin excepciones la ejecución se terminará y se produce un mensaje como el siguiente

```
Exception in thread "main" java.lang.NumberFormatException:
   For input string: "  1"
        at java.lang.NumberFormatException.forInputString
          (NumberFormatException.java:48)
        at java.lang.Integer.parseInt(Integer.java:449)
        at java.lang.Integer.parseInt(Integer.java:499)
        at Excepciones.ejemplo_sin_excepciones(Excepciones.java:5)
        at Excepciones.main(Excepciones.java:20)
Java Result: 1
```

Sin embargo la ejecución del ejemplo con excepciones no terminará la ejecución dando oportunidad de volver al capturar o tomar una acción alterna. La ejecución en este caso es:

```
No es un numero, es una cadena de texto.
```

En el último ejemplo de División por cero, en el caso de que el divisor sea cero el resultado de esta será infinito en cuyo caso el compilador terminará la ejecución con el siguiente mensaje

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Excepciones.main(Excepciones.java:37)
Java Result: 1
```

Con el manejo de excepciones podemos eliminar el problema y sugerir un nuevo dato o captura uno nuevo desde teclado

Modelado de Objetos

5.1. Modelos de especificación del Dominio: La técnica CRC

En la fase de diseño del desarrollo de software, una de las tareas es descubrir las estructuras que hacen posible la implementación de un conjunto de tareas en una computadora. Cuando se utiliza el proceso de diseño orientado a objetos, se tienen llevar a cabo las siguientes tareas:

1. Descubrir clases.

2. Determinar las responsabilidades de cada clase.

3. Describir las relaciones entre las clases.

Una clase representa un concepto útil. Por ejemplo, suponga que su trabajo consiste en imprimir una factura, como la que se muestra enseguida

```
                        F A C T U R A

Sam
Calle Principal #100
Algun pueblo, Mexico 98765

Descripcion                     Precio  Cda   Total
Tostador                         29,95    3   89,85
Secador de Cabello               24,95    1   24,95
Aspiradora                       19,99    2   39,98

CANTIDAD A PAGAR: $  154,78
```

Las clases obvias que vienen a la mente son de factura, partida, y el Cliente. Es una buena idea tener una lista de clases de candidatos en una pizarra o una hoja de papel. Como una lluvia de ideas, sólo tiene que poner todas las ideas para las clases en la lista. Siempre se

puede tachar los que no eran útiles.

Cuando estemos buscando clases, debemos tener los siguientes puntos en mente:

• Una clase representa un conjunto de objetos con el mismo comportamiento. Entidades con varias apariciones en la descripción del problema, tales como clientes o productos, serán buenos candidatos para definirlos como objetos. Descubre lo que tienen en común, y diseñe clases que capturen esos puntos en común.

• Algunas entidades deben ser representadas como objetos, otros como tipos primitivos. Por ejemplo, ¿Una dirección debe ser un objeto de una clase de agenda, o deberá ser simplemente una cadena? No hay una respuesta-que sea perfecta depende de la tarea que desea resolver. Si el software tiene que analizar las direcciones (por ejemplo, para determinar los gastos de envío), entonces, la clase que represente al objeto dirección será apropiado. Sin embargo, si su software nunca necesitará esa capacidad, no debe perder el tiempo en un diseño excesivamente complejo. Es su trabajo encontrar un diseño equilibrado, que no esté demasiado limitado ni excesivamente general.

• No todas las clases se pueden descubrir en la fase de análisis. Los programas más complejos necesitan clases para fines tácticos, tales como archivo o base de datos de acceso, interfaces de usuario, mecanismos de control, y otras más.

• Algunas de las clases que usted necesita ya existen, ya sea en la biblioteca estándar o en un programa que ha desarrollado previamente. También puede ser capaz de utilizar la herencia para extender las clases existentes en clases que coincidan con sus necesidades.

Una vez que se ha identificado un conjunto de clases, es necesario definir el comportamiento de cada clase. Es decir, usted necesita saber qué métodos tiene cada objeto tiene y qué hacer para resolver el problema de programación. Una regla sencilla para encontrar estos métodos es la búsqueda de verbos en la descripción de la tarea y, a continuación verificar si los verbos coinciden con los objetos apropiados. Por ejemplo, en el programa de factura, la clase necesita calcular la cantidad total debida. Ahora hay que averiguar qué clase es responsable de este método. ¿Los clientes calculan lo que deben? ¿Los facturas la cantidad debida? ¿Los elementos el total? La mejor opción es calcular la cantidad debida como responsabilidad de la clase Factura.

Una excelente manera de llevar a cabo esta tarea es el método de tarjetas CRC. CRC es sinónimo de Clases, Responsabilidad, Colaboradores, y en su forma más simple, el método funciona de la siguiente manera: Use una tarjeta para cada clase u objeto, al pensar en los verbos recuerde que describen la tareas que deben ser implementadas por los métodos y al elegir la tarjeta de la clase que usted cree que debería ser responsable y escribir la responsabilidad en la tarjeta. A continuación se presenta un ejemplo de tarjeta CRC

C l a s e s	
R e s p o s a b i l i d a d e s	C o l a b o r a d o r e s

Para cada responsabilidad, se requieren otras clases para cumplir tal cometido. Estas clases son los colaboradores.

Por ejemplo, supongamos que usted decide que una factura se debe calcular la cantidad debida. A continuación, se escribe calcular cantidad debida en el lado izquierdo de una ficha con la Factura título.

Si una clase se puede llevar a cabo esa responsabilidad por sí mismo, no hacer nada más. Pero si la clase necesita la ayuda de otras clases, escribir los nombres de estos colaboradores en el lado derecho de la tarjeta. Para calcular el total de la factura tiene que pedir a cada elemento de línea sobre el precio total. Por lo tanto, la clase partida es un colaborador.

A continuación se muestra la Tarjeta CRC con esta información para la clase Factura.

F a c t u r a	
Calcular el total a pagar	Partida

5.2. Diagrama de Clases (modelo conceptual)

En el diseño de un programa, es útil documentar las relaciones entre las clases. Esto le ayudará de muchas maneras. Por ejemplo, si usted encuentra clases con comportamiento común, puede ahorrar esfuerzo, colocando el comportamiento común en una superclase. Si usted sabe que algunas clases no están relacionados entre sí, se pueden asignar diferentes programadores para implementar cada uno de ellas, sin tener que preocuparse de que uno de ellos tiene que esperar a que el otro.

La herencia es una relación muy importante, pero resulta, que no es la única relación útil, y puede ser usado en exceso.

La herencia es una relación entre una clase general (a la que se denomina superclase) y una clase más especializada (la subclase). Esta relación se describe a menudo como la relación "... es un ..."por ejemplo "Cada camión es un vehículo". Cada cuenta de ahorros es una cuenta bancaria. Cada círculo es una elipse (con la misma anchura y altura).

A veces se abusa de la Herencia, sin embargo. Por ejemplo, considere una clase neumático que describe un neumático de coche. ¿Pero la clase neumático debe ser una subclase de la clase Círculo? Suena conveniente. Hay un buen número de métodos útiles en la clase Círculo por ejemplo, la clase de neumático puede heredar métodos que calculan el área, el perímetro, y el punto central, que debería ser útil en la elaboración de formas de neumático. Aunque puede ser conveniente para él programador, esta disposición no tiene sentido conceptualmente. No es cierto que todos los neumáticos son un círculos. Los neumáticos son piezas de automóviles, mientras que los círculos son objetos geométricos. Existe una relación entre los neumáticos y círculos, sin embargo un neumático tiene un círculo como perímetro. Java nos permite modelar este tipo de relación utilizando variables de instancia:

```
public class Neumatico {
private String rating;
private Circulo [ ] limite;
}
```

El término técnico para esta relación es la agregación. Por lo tanto decimos que un agregado de la clase neumático son los objetos Circulo. En general, decimos que una clase tiene agregados de otra clase, si tiene objetos de la otra clase.

Aquí hay otro ejemplo. Cada coche es un vehículo. Cada coche tiene un neumático (de hecho, tiene normalmente cuatro o, si se cuenta el de repuesto, cinco). Por lo tanto, debe utilizar la herencia del vehículo y usar la agregación de los objetos del neumático:

```
public class Carro extends Vehicle {
private Neumatico[ ] neumaticos;
. .. }
```

En este capitulo utilizaremos la notación UML para los diagramas de clases. En la notación UML utilizaremos para la herencia, una flecha con un Triángulo abierto que apunta a la superclase. En la notación UML, la agregación se denota por una línea sólida con un símbolo en forma de diamante siguiendo a la clase de agregación. La Figura 5.4 muestra un diagrama de clases con una herencia y una relación de agregación.

La relación de agregación se refiere a la relación de dependencia, recordemos que una clase depende de otro si uno de sus métodos utiliza un objeto de la otra clase de alguna manera. Por ejemplo, muchas de nuestras aplicaciones dependen de la clase de escáner, ya que utilizan un objeto escáner para leer la entrada.

Utilizaremos la notación UML para los diagramas de clases. Por ejemplo, una clase puede utilizar la clase escáner sin tener que declarar una variable de instancia del explorador de clases. La clase puede simplemente construir una variable local del tipo de escáner, o sus métodos puede recibir objetos de escáner como parámetros.

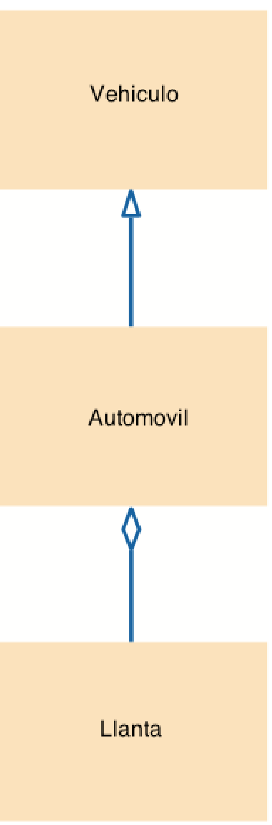

Figura 5.4: Diagrama UML para un vehículo

En general, es necesario agregación cuando un objeto tiene que recordar otro objeto entre llamadas a métodos. La notación UML para la dependencia es una línea discontinua con una flecha abierta que apunta a la clase dependiente.

En La siguiente Tabla muestra un resumen de los cuatro símbolos de relación UML que utilizamos para el desarrollo de Programas orientados a objetos.

Relación	Símbolo	Estilo de linea	Tipo de flecha
Herencia	———▷	Solida	Triángulo
Interface implementada	- - - - - - - ▷	Punteada	Triángulo
Agregación	◇———	Solida	Diamante
Dependencia	- - - - - - - →	Punteada	Abierta

5.3. Ejemplos prácticos para descubrir Clases y Métodos utilizando la Técnica CRC

Como ejemplo de aplicación consideraremos el caso de una Factura. Una vez desarrollado el proceso de lluvia de ideas, para este ejemplo tomemos en cuenta que tiene 4 clases y los diagramas CRC son:

F a c t u r a	
Formato de la Factura	Partida
Agregar un producto y cantidad	Cliente
Calcular el total	

P a r t i d a	
Formato de la Partida	Producto
Calcular el precio total	

C l i e n t e	
Formato para desplegar clientes	

Producto	
Obtener la descripción del producto	
Obtener el precio del producto	

Después de haber descubierto las clases y sus relaciones con tarjetas CRC, se deben registrar en los diagramas UML. Las relaciones de dependencia estarán en la columna de la colaboración de las tarjetas CRC. Cada clase depende de las clases con las que colabora.

En nuestro ejemplo, la clase Factura colabora con la Cliente, Partida, y las clases de productos. La clase Partida colabora con la clase Producto. Por lo tanto, los agregados de la clase de Factura a las clases de Dirección y Partida. Los agregados de clase Partida la clase de Productos. Sin embargo, no hay una relación entre una Factura y un Producto. Una Factura no almacena productos directamente se almacenan en los objetos Partida. Finalmente No hay herencia en este ejemplo. En la Figura 5.5 se presenta el diagrama UML con todas sus dependencias.

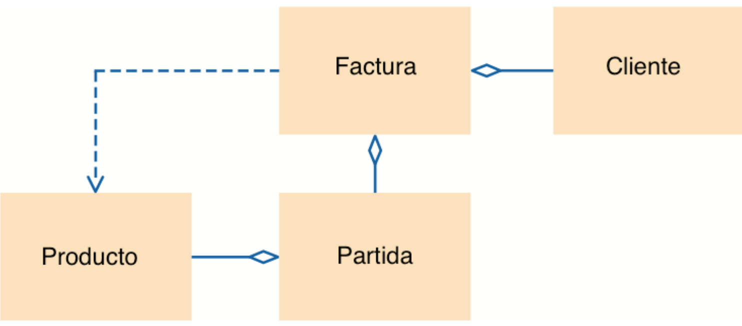

Figura 5.5: Diagrama UML para generar una Factura

La implementación en Java queda como:

```
package Factura;

/**
 * Metodo Imprime la Factura
 * @author felix
 */

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

```
        Cliente samsDomicilio = new Cliente("Sam",
                "Calle Principal #100", "Algun pueblo", "Mexico", "98765");

        Factura samsFactura = new Factura(samsDomicilio);

        samsFactura.agrega(new Producto("Tostador", 29.95), 3);
        samsFactura.agrega(new Producto("Secador de Cabello", 24.95), 1);
        samsFactura.agrega(new Producto("Aspiradora", 19.99), 2);

        System.out.println(samsFactura.formato()); }
}

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

package Factura;

import java.util.ArrayList;

/**
 * Clase para manejar el Objeto Factura
 * @author felix
 */

public class Factura {
    private Cliente datosCliente;
    private ArrayList<Partida> items;

    /**
     * Constructor de Factura
     * @param unDomicilio
     */

    public Factura(Cliente unCliente) {
        items = new ArrayList<Partida>();
        datosCliente = unCliente;
    }

    /**
     * Agraga un producto y su cantidad a la Factura
     * @param unProducto
     * @param unaCantidad
```

```
     */

    public void agrega(Producto unProducto, int unaCantidad) {
        Partida unItem = new Partida(unProducto, unaCantidad);
        items.add(unItem);
    }

    /**
     * Da formato a la cadena de salida a la Factura
     * @return Regresa la Factura en formato establecido
     */

    public String formato() {
        String r = "                        F A C T U R A \n\n"
                 + datosCliente.formato()
                 + String.format("\n\n%-30s%8s%5s%8s\n",
                 "Descripcion", "Precio", "Cda", "Total");
        for (Partida item : items) {
            r = r + item.formato() + "\n";
        }

        r = r + String.format("\nCANTIDAD A PAGAR: $%8.2f", obtenCantidadDebida());
                return r;
    }

    /**
     * Obtiene la Cantidad total a pagar por la factura
     * @return Calcula el total a pagar por la Factura
     */

    public double obtenCantidadDebida() {
        double cantidadDebida = 0;

        for (Partida item : items)
            cantidadDebida = cantidadDebida + item.obtenPrecioTotal();
        return cantidadDebida;
    }
}

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
package Factura;

/**
 * Clase para manejar el Domicilio de Facturacion
 * @author felix
 */

public class Cliente {
    private String nombre;
    private String domicilio;
    private String ciudad;
    private String estado;
    private String CP;

    /**
     * Constructor para el domicilio de la Factura
     * @param unNombre Nombre a quien se factura
     * @param unaCalle Domicilio de Facturacion
     * @param unaCiudad Ciudad de Facturacion
     * @param unEstado Estado
     * @param unCP Codigo Postal
     */

    public Cliente(String unNombre, String unDomicilio, String unaCiudad,
            String unEstado, String unCP) {
        nombre    = unNombre;
        domicilio = unDomicilio;
        ciudad    = unaCiudad;
        estado    = unEstado;
        CP = unCP;
    }

    /**
     * Formato del Domicilio de Facturacion
     * @return Domicilio en formato de la Factura
     */

    public String formato() {
        return nombre + "\n" + domicilio + "\n"
                + ciudad + ", " + estado + " " + CP;
    }
```

```
}

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

package Factura;

/**
 * Clase para representar una partida de la factura.
 * La partida es renglon con la descripcion del articulo, cantidad y total
 * @author felix
 */

public class Partida {
    private Producto elProducto;
    private int cantidad;

    /**
     * Contructor de una partida
     * @param unProducto Un producto
     * @param unaCantidad Una cantidad
     */

    public Partida(Producto unProducto, int unaCantidad) {
        elProducto = unProducto;
        cantidad = unaCantidad;
    }

    /**
     * Calcula el precio total de la partida
     * @return Calcula el precio total de la partida
     */

    public double obtenPrecioTotal() {
        return elProducto.obtenPrecio() * cantidad;
    }

    /**
     * Da formato a una partida de la Factura
     * @return Regresa una cadena con la información de un articulo
     */
```

```
    public String formato() {
        return String.format("%-30s%8.2f%5d%8.2f",
                elProducto.obtenDescripcion(),
                elProducto.obtenPrecio(), cantidad, obtenPrecioTotal());
    }
}

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

package Factura;

/**
 * Esta clase almacena la información
 * @author felix
 */

public class Producto {
    private String descripcion;
    private double precio;

    /**
     * Constructor para un producto
     * @param aDescription descripcion del producto
     * @param aPrice
     */

    public Producto(String unaDescripcion, double unPrecio) {
        descripcion = unaDescripcion;
        precio = unPrecio;
    }

    /**
     * Obtiene la Descripción de un producto
     * @return Regresa la descripcion de un Producto
     */

    public String obtenDescripcion() {
        return descripcion;
    }
```

```
    /**
     * Obtiene el precio de un producto
     * @return Regresa el precio de un Producto
     */

    public double obtenPrecio() {
        return precio;
    }
}
```

5.3.1. Documentación

La manera en que Java permite documentar un programa o conjunto de programas en un paquete utilizando la instrucción javadoc. En el caso de nuestro paquete de Factura, podemos crear la documentación haciendo

```
javadoc Factura
```

donde Factura es el directorio donde se encuentran todos los archivos del paquete a los que se les adiciono la leyenda package Factura. La instrucción generará una archivo índex.html con la información referente a las clases tal como se muestra en la Figura 5.6

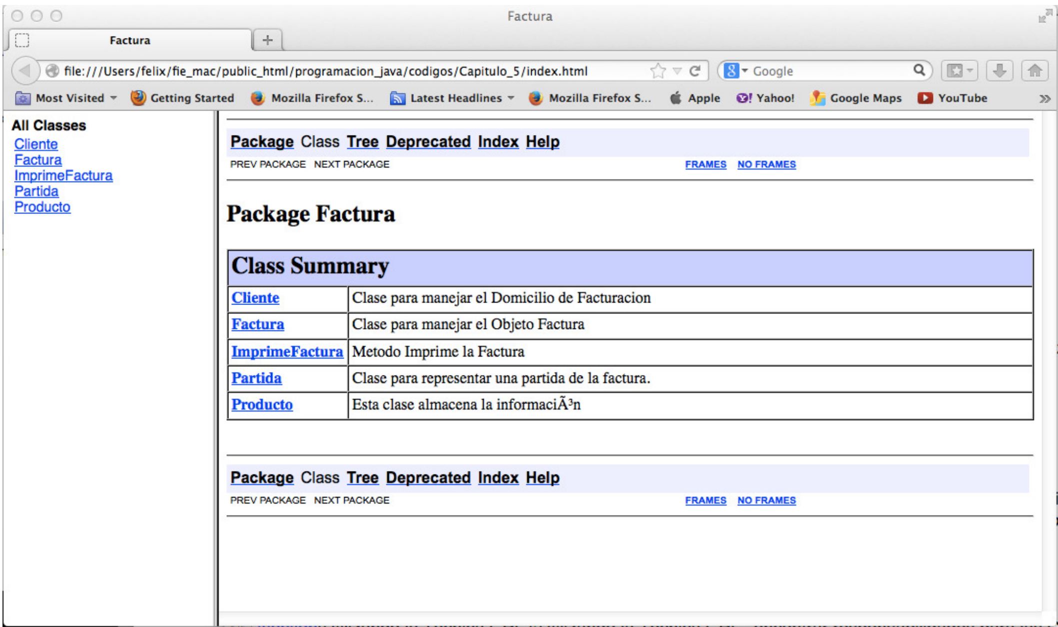

Figura 5.6: Documentación en Java del paquete Factura

5.4. Utilizando la Técnica CRC, encontrar responsabilidades para las Clases y Colaboraciones

Para mostrar esta técnica, se muestran la simulación de dos procesos, el primero es el relacionado con una Escuela y el segundo con un Cajero. En ambos casos se muestran los diagramas CRC, UML y el código Java Correspondiente.

5.4.1. Ejemplo Escuela

Como ejemplo haremos los diagramas CRC y UML para una escuela así como el código correspondiente. Se detectan los objetos

- Alumno
- Curso
- Escuela
- Fecha
- Materia
- Persona
- Profesor

En la figura 5.7, se muestran las interconexiones entre las clases para modelar la Escuela

Fecha

Comenzaremos por ver aquellos objetos que no tienen colaboradores. La fecha la utilizaremos para dar la fecha de nacimiento de una persona y calcular su edad

Fecha	
Responsabilidad	**Colaborador**
«costructor» Fecha	
formato	
getAnio	

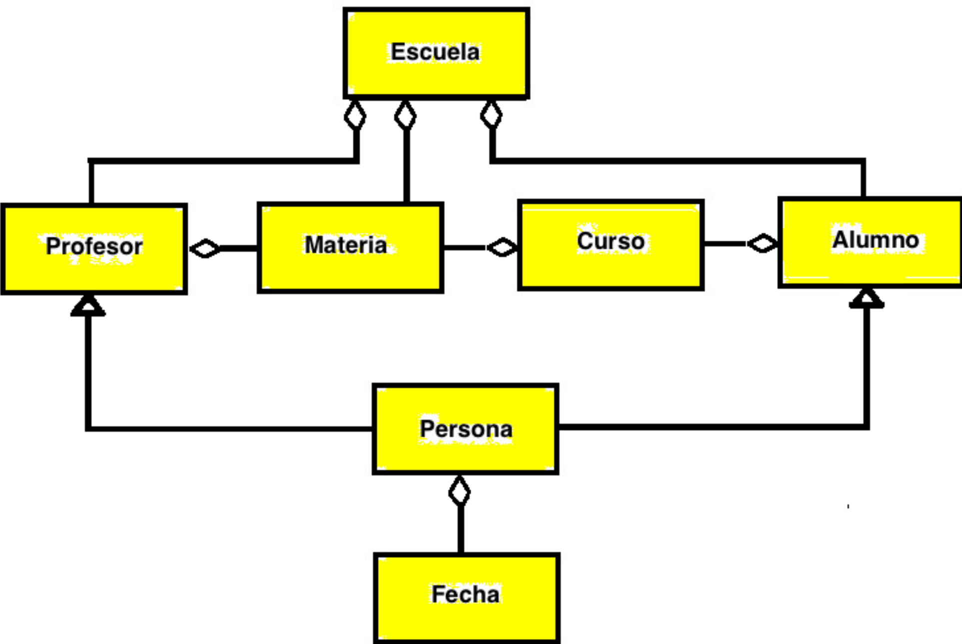

Figura 5.7: Interconexion de clase para modelar Escuela

Fecha
- dia : Entero - mes : Entero - anio : Entero
«costructor» Fecha(unDia: Entero, unMes :Entero, unAnio : Entero) formato() : String getAnio() : Entero

Curso

Definimos un Curso como una materia que cursa el alumno, dicha materia tiene una clave, descripción, calificación y sección donde el alumno la tomo.

Curso	
Responsabilidad	**Colaborador**
«costructor» Curso	
getClave	
getDescripcion	

Curso
- clave_materia : String - Descripcion : String - Calificacion : Entero - sección : String
«costructor» curso(unaClave: String, unaDescripcion : String, unaCalificacion : Entero, unaSeccion : String) getClave() : String getDescripcion() : Entero informacion() : String

Materia

Cada una de las Materias que se imparten en la escuela la intentaremos modelar utilizando una clave, Descripción o nombre de la materia, Horas a la semana que se dedican para esta materia y el Grado en el cual se imparte la materia.

Materia	
Responsabilidad	**Colaborador**
«costructor» Materia	
informacion	
getDescripcion	
getClave	

Materia
- clave_materia : String - Descripcion : String - Horas : Entero - Grado : Entero
«costructor» Materia(unaClave: String, unaDescripcion : String, unaHora : Entero, unGrado : Entero) informacion() : String getdescripcion() : String getClave() : String

Persona

Una Persona dentro de la Escuela puede tomar diferentes roles, pude ser alumno, maestro o empleado. Para nuestro caso solamente consideraremos Alumnos y Profesores. Las clase Alumno y Profesor las haremos descendientes de esta clase, por lo tal debemos poner especial cuidado.

Persona	
Responsabilidad	**Colaborador**
«costructor» Persona	Fecha
informacion	
Edad	

Persona
- Nombre : String - Parterno : String - Materno : String - Domicilio : String - fecha_nacimiento : Fecha
«costructor» Persona (unNombre:String, unPaterno:String, unMaterno:String, unDomicilio : String, unaFecha : Fecha) informacion() : String Edad(unAño : Entero) : String

Alumno

En el caso del objeto Alumno adicionalmente a la información para una Persona, agregamos una matricula y una lista de cursos a los que llamaremos kardex.

Alumno	
Responsabilidad	**Colaborador**
«costructor» Alumno	Persona
agrega_curso	Curso
cursos	
informacion	
cursa	

Alumno
- Nombre : String (heredada) - Parterno : String (heredada) - Materno : String (heredada) - Domicilio : String (heredada) - fecha_nacimiento : Fecha (heredada) - Matricula : String - kardex : Arreglo de Cursos
«costructor» Alumno (unNombre:String, unPaterno:String, unMaterno:String, unDomicilio : String, unaFecha : Fecha , unaClave :String) agregacurso(unCurso:Curso) cursos() : String informacion() : String cursa(unaMateria : String) : Boolean

Profesor

La otra Persona que involucramos en nuestra escuela es Profesor, adicionalmente a la información como persona agregamos una clave de Profesor y una lista de Materias que imparte.

Profesor	
Responsabilidad	**Colaborador**
«costructor» Profesor	Persona
agrega_materia	Materia
informacion	

Profesor
- Nombre : String (heredada) - Parterno : String (heredada) - Materno : String (heredada) - Domicilio : String (heredada) - fecha_nacimiento : Fecha (heredada) - Clave_Profesor : String - Lista_Materias : Arreglo de Materias
«costructor» Alumno (unNombre:String, unPaterno:String, unMaterno:String, unDomicilio : String, unaFecha : Fecha , unaClave :String) agrega_materia(unaMateria:Materia) informacion() : String

Escuela

En el caso del objeto Escuela tenemos:

Escuela	
Responsabilidad	**Colaborador**
«costructor» Escuela	Alumno
agrega_alumno	Profesor
agrega_materia	Materias
Nuevo	
existeMateria	
Lista_Alumnos	
Lee_Materias	
Lista_Alumnos_por_materia	
Lista_Materias_por_Alumno	
Lee_Alumnos	

Escuela
- Nombre : String - Alumnos : Arreglo de Alumno - Profesores : Arreglo de Profesor - Materias : Arreglo de Materia
«costructor» Escuela (unNombre:String) agrega_alumno(unAlumno : Alumno) agrega_materia(unaMateria : Materia) Nuevo(unaClave : String, unaCalificacion :Entero, unaSeccion : String) : Curso existeMateria(unaClave : String) : Entero Lista_Alumnos(unaMateria : String) Lee_Materias() Lista_Alumnos_por_materia() Lista_Materias_por_Alumno() Lee_Alumnos()

5.4.2. Ejemplo Cajero

Para ver la utilización de la técnica CRC haremos la simulación de un cajero automático de banco. El Cajero manejará un conjunto de clientes de un Banco y cada cliente tendrá dos cuentas una de ahorros y otra de cheques. En el cajero se podrán hacer las operaciones de deposito y retiro de una cuenta que se seleccione. El ingreso al cajero será a través de un número de cliente y una clave de acceso de cuatro dígitos.

Los objetos que podemos modelar para simplificar el problema son:

- Cuenta
- Cliente
- Banco
- Cajero

En la figura 5.8 se muestra el diagrama de interconexión de objetos

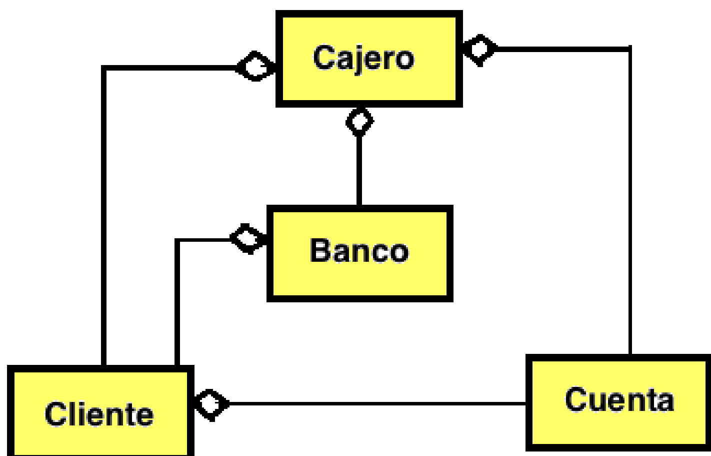

Figura 5.8: Interconexion de clase para modelar un Cajero

Cuenta
- saldo : doble
«costructor» Cuenta () «costructor» Cuenta (saldoInicial : doble) deposito(cantidad : doble) retiro(cantidad : doble) obtenSaldo() : double

Banco
- clientes : Arreglo de Cliente
«costructor» Banco () leeClientes() agregaCliente(unCliente : Cliente) buscaCliente(unNumero: Entero)

Cliente
- NumeroCliente : Entero - clave : Entero - cuenta_de_Cheque : Cuenta - cuenta_de_Ahorros : Cuenta
«costructor» Cliente (unNumero : Entero, unaClave : Entero) empata(unNumero: Entero, unaClave: Entero) obten_cuenta_de_Cheques() : Cuenta obten_cuenta_de_Ahorros() : Cuenta

Cajero
- estado : Entero - NumeroCliente : Entero - ClienteActual : Cliente - CuentaActual : Cuenta - elBanco : Banco
«costructor» Cajero () verEstado() : Entero muestraEstado(estadoNuevo : Entero) ponNumeroCliente(unCliente : Entero) seleccionaCliente(unaClave : Entero) seleccionaCuenta(unaCuenta : Entero) retiro(miRetiro: doble) deposito(miDeposito: doble) muestraSaldo() : doble salir()

Diseño de Clases en Java

6.1. Repaso de Clases y Objetos

En general una clase en Java representará un objeto. Las clases en Java tendrán atributos y métodos, los atributos estas relacionados con las características del objeto como pueden ser tamaño, color, forma, etc y los métodos están asociados a operaciones o funciones con o sobre los atributos.

```
class carro{

//atributos

private int color;
...

// ************************
métodos

public acelerar() {
 ....
}
}
```

6.2. Anatomía de una Clase (java y UML)

Clase

1. Atributos

    a) Estáticos : las variables y los métodos marcados con el modificador 'static', son variables y/o métodos que pertenecen a la clase, no a una instancia de dicha

clase en particular

b) Dinámicos : básicamente los métodos no estáticos o dinámicos, se usan para trabajar sobre una instancia de un objeto.

c) Visibilidad

1) Private : Sólo se puede acceder al elemento desde métodos de la clase, o sólo puede invocarse el método desde otro método de la clase.

2) Protectec : es visible para esa clase y sus clases heredadas el uso puede ser el mismo que private

3) Public :Todo el mundo puede acceder al elemento. Si es un dato miembro, todo el mundo puede ver el elemento, es decir, usarlo y asignarlo. Si es un método todo el mundo puede invocarlo.

4) Fiendly : significa que todas las clases del paquete actual tienen acceso al miembro amistoso , pero todas las clases fuera del paquete tienen un acceso privado a este método

2. Métodos

a) Constructores : Su función es iniciar los atributos de un objeto.

b) Consultores : Permiten ver los contenidos de los atributos. Estos se hacen con el propósito de garantizar el encapsulamiento de los datos.

c) Modificadores : Modifican el contenido de los atributos.

d) Habituales : Cualquier método para realizar una tarea especifica

e) Visibilidad : Se utilizan los mismos modificadores de visibilidad que los atributos

6.3. Encapsulación

Java, como un lenguaje orientado a objetos, que implementa la encapsulación. Este concepto consiste en la ocultación del estado o de los datos miembro de un objeto o atributos, de forma que sólo es posible modificar los mismos mediante los métodos definidos para dicho objeto. Con esto se tiene control sobre los atributos que pueda llegar a tener un objeto, así por ejemplo un circulo no puede tener radio negativo.

```
public class circulo {

    private double radio;
```

```
    public void asigna_radio(double unRadio) {
        radio = unRadio > 0 ? unRadio : 0;
    }

    public double obten_radio(){
        return radio;
    }
}
```

Cada objeto está aislado del exterior, de forma que la aplicación es un conjunto de objetos que colaboran entre sí mediante el paso de mensajes invocando sus operaciones o métodos. De esta forma, los detalles de implementación permanecen ocultos a las personas que usan las clases, evitando así modificaciones o accesos indebidos a los datos que almacenan las clases.

Además, el usuario de la clase no se tiene que preocupar de cómo están implementados los métodos y propiedades, concentrándose sólo en cómo debe usarlos.

La encapsulación es una de las principales ventajas que proporciona la programación orientada a objetos. Como ejemplo tenemos el siguiente código

```
public class MiClase {
    private int tipo;

    public void setTipo(int untipo) {
        tipo = untipo;
    }

    public int getTipo() {
        return tipo;
    }

    public static void main(String args[]){
        MiClase mc = new MiClase();
        mc.setTipo(5);

        System.out.println("Dato = " + mc.gerTipo());
    }
}
```

6.4. Anatomía de un método

En général un método de Java puede ser representado por:

un_modificador_acceso visibilidad_accesar_metodo valor_retorna Nombre_metodo(Variables_que_recibe)
{

}

1. Modificadores de acceso

   a) Estáticos : los métodos marcados con el modificador 'static', métodos que pertenecen a la clase, no a una instancia de dicha clase en particular

   b) Dinámicos : básicamente los métodos no estáticos o dinámicos, se usan para trabajar sobre una instancia de un objeto.

2. Visibilidad de acceso

3. Valores de retorno: cualquier tipo de datos u objeto que se desea regresar

4. Nombre del método : Los nombres de los método normalmente especifican lo que este hace

5. Variables del método: Conjunto de tipos de variable u objetos que el método necesita

Como ejemplo podemos ver:

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

Otro ejemplo

```
public class AnatomiaMetodo {
    public static void miMetodo1() {
        System.out.println("Metodo estatico");
    }

    public void miMetodo2() {
        System.out.println("Metodo dinamico");
    }

}
```

Para probar hacer

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

```
        AnatomiaMetodo.miMetodo1();
        AnatomiaMetodo a = new AnatomiaMetodo();

        a.miMetodo2();
        AnatomiaMetodo.miMetodo2();
    }

}
```

6.5. Revisión de constructores

Un constructor es un método especial de una clase que se llama automáticamente siempre que se declara un objeto de esa clase. Su función es inicializar el objeto y sirve para asegurarnos que los objetos siempre contengan valores válidos.

Cuando se crea un objeto en Java se realizan las siguientes operaciones de forma automática:

1. Se asigna memoria para el objeto.

2. Se inicializan los atributos de ese objeto con los valores predeterminados por el sistema.

3. Se llama al constructor de la clase que puede ser uno entre varios.

El constructor de una clase tiene las siguientes características:

Tiene el mismo nombre que la clase a la que pertenece.

En una clase puede haber varios constructores con el mismo nombre y distinto número de argumentos (se puede sobrecargar)

No se hereda.

No puede devolver ningún valor (incluyendo void).

Debe declararse público (salvo casos excepcionales) para que pueda ser invocado desde cualquier parte donde se desee crear un objeto de su clase.

MÉTODO CONSTRUCTOR POR DEFECTO

Si para una clase no se define ningún método constructor se crea uno automáticamente por defecto. El constructor por defecto es un constructor sin parámetros que no hace nada. Los atributos del objeto son iniciados con los valores predeterminados por el sistema. Por ejemplo, en la clase Fecha:

```
import java.util.*;
public class Fecha {

    private int dia;
    private int mes;
    private int año;

    private boolean esBisiesto() {
        return ((año % 4 == 0) && (año % 100 != 0) || (año % 400 == 0));
    }

    public void setDia(int d) {
       dia = d;
    }

    public void setMes(int m) {
        mes = m;
    }

    public void setAño(int a) {
        año = a;
    }

    public void asignarFecha() {
        Calendar fechaSistema = Calendar.getInstance();
        setDia(fechaSistema.get(Calendar.DAY_OF_MONTH));
        setMes(fechaSistema.get(Calendar.MONTH));
        setAño(fechaSistema.get(Calendar.YEAR));
    }

    public void asignarFecha(int d) {
        Calendar fechaSistema = Calendar.getInstance();
        setDia(d);
        setMes(fechaSistema.get(Calendar.MONTH));
        setAño(fechaSistema.get(Calendar.YEAR));
    }

    public void asignarFecha(int d, int m) {
        Calendar fechaSistema = Calendar.getInstance();
        setDia(d);
        setMes(m);
```

```
        setAño(fechaSistema.get(Calendar.YEAR));
    }

    public void asignarFecha(int d, int m, int a) {
        setDia(d);
        setMes(m);
        setAño(a);
    }

    public int getDia() {
        return dia;
    }

    public int getMes() {
        return mes;
    }

    public int getAño() {
        return año;
    }

    public boolean fechaCorrecta() {
        boolean diaCorrecto, mesCorrecto, anyoCorrecto;
        anyoCorrecto = (año > 0);
        mesCorrecto = (mes >= 1) && (mes <= 12);
        switch (mes) {
            case 2:
                if (esBisiesto()) {
                    diaCorrecto = (dia >= 1 && dia <= 29);
                } else {
                    diaCorrecto = (dia >= 1 && dia <= 28);
                }
                break;
            case 4:
            case 6:
            case 9:
            case 11:
                diaCorrecto = (dia >= 1 && dia <= 30);
                break;
            default:
                diaCorrecto = (dia >= 1 && dia <= 31);
```

```
        }
        return diaCorrecto && mesCorrecto && anyoCorrecto;
    }
}
```

no se ha definido ningún constructor, por lo que al declarar un objeto el compilador utilizará un constructor por defecto. En este caso el método constructor por defecto es:

```
Fecha() { }
```

Vamos a añadir un constructor a la clase Fecha para poder iniciar los atributos de los nuevos objetos con valores determinados que se envían como parámetros. Si los valores corresponden a una fecha incorrecta se asigna la fecha del sistema:

```
public Fecha(int d, int m, int a) {
        dia = d;
        mes = m;
        año = a;

        if (!fechaCorrecta()) {
            Calendar fechaSistema = Calendar.getInstance();
            setDia(fechaSistema.get(Calendar.DAY_OF_MONTH));
            setMes(fechaSistema.get(Calendar.MONTH));
            setAño(fechaSistema.get(Calendar.YEAR));
        }
}
```

Con este método constructor podremos crear objetos de la siguiente manera:

```
Fecha fecha1 = new Fecha(10,2,2011);
```

Se crea un objeto fecha1 y se le asignan esos valores a sus atributos.

Si se crea un objeto con una fecha no válida:

```
Fecha fecha2 = new Fecha(10,20,2011);
```

A dia, mes y año se les asignará los valores del sistema. Java crea un constructor por defecto si no hemos definido ninguno en la clase, pero si en una clase definimos un constructor ya no se crea automáticamente el constructor por defecto, por lo tanto si queremos usarlo deberemos escribirlo expresamente dentro de la clase.

En este ejemplo hemos definido un método constructor por lo que también es necesario poner el método constructor por defecto.

public Fecha()

Este constructor se invocará cuando se declare un objeto sin parámetros. La clase tiene ahora dos constructores por lo que podemos crear objetos Fecha de dos formas:

```
Fecha f1 = new Fecha();   //se ejecuta el constructor sin parámetros
Fecha f2 = new Fecha(1,1,2010); //se ejecuta el constructor con parámetros
```

INVOCAR A UN MÉTODO CONSTRUCTOR

Un constructor se invoca cuando se crea un objeto de la clase mediante el operador new. Si es necesario invocarlo en otras situaciones, la llamada a un constructor solo puede realizarse desde dentro de otro constructor de su misma clase y debe ser siempre la primera instrucción. Si tenemos varios métodos constructores en la clase, podemos llamar a un constructor desde otro utilizando this. Por ejemplo, si en el constructor con parámetros de la clase Fecha fuese necesario llamar al constructor sin parámetros de la misma clase escribimos:

```
public Fecha(int d, int m, int a) {
         this();  //llamada al constructor sin parámetros
         dia = d;
         ..........
}
```

CONSTRUCTOR COPIA Se puede crear un objeto a partir de otro de su misma clase escribiendo un constructor llamado constructor copia. Este constructor copia los atributos de un objeto existente al objeto que se está creando. Los constructores copia tiene un solo argumento, el cual es una referencia a un objeto de la misma clase que será desde el que queremos copiar. Por ejemplo, para la clase Fecha podemos escribir un constructor copia que permita crear objetos con los valores de otro ya existente:

```
Fecha fecha = new Fecha(1,1,2011);   //se invoca al constructor con parámetros
Fecha fecha1 = new Fecha(fecha);  //se invoca al constructor copia
```

El constructor copia que escribimos para la clase es: //constructor copia de la clase Fecha

```
public Fecha(final Fecha f) {
        dia = f.dia;
        mes = f.mes;
        año = f.año;
}
```

El constructor copia recibe una referencia al objeto a copiar y asigna sus atributos uno a uno a cada atributo del objeto creado. Declarar el parámetro como final no es necesario pero protege al objeto a copiar de cualquier modificación accidental que se pudiera realizar sobre él.

6.6. Miembros estáticos

Las variables de clase o miembros estáticos son aquellos a los que se antepone el modificador static. Vamos a comprobar que un miembro dato estático guarda el mismo valor en todos los objetos de dicha clase.

Sea una clase denominada Alumno con dos miembros dato, la nota de selectividad, y un miembro estático denominado nota de corte. La nota es un atributo que tiene un valor distinto para cada uno de los alumnos u objetos de la clase Alumno, mientras que la nota de corte es un atributo que tiene el mismo valor para a un conjunto de alumnos. Se define también en dicha clase una función miembro que determine si está (true) o no (false) admitido.

```
public class Alumno {
    double nota;
    static double notaCorte=6.0;
    public Alumno(double nota) {
        this.nota=nota;
    }
    boolean estaAdmitido(){
        return (nota>=notaCorte);
    }
}
```

Creamos ahora un array de cuatro alumnos y asignamos a cada uno de ellos una nota.

```
        Alumno[] alumnos={new Alumno(5.5), new Alumno(6.3),
new Alumno(7.2), new Alumno(5.0)};
```

Contamos el número de alumnos que están admitidos

```
        int numAdmitidos=0;
        for(int i=0; i<alumnos.length; i++){
            if (alumnos[i].estaAdmitido()){
                numAdmitidos++;
            }
        }
        System.out.println("admitidos "+numAdmitidos);
```

Accedemos al miembro dato notaCorte desde un objeto de la clase Alumno, para cambiarla a 7.0

```
        alumnos[1].notaCorte=7.0;
```

Comprobamos que todos los objetos de la clase Alumno tienen dicho miembro dato estático notaCorte cambiado a 7.0

```
for(int i=0; i<alumnos.length; i++){
    System.out.println("nota de corte "+alumnos[i].notaCorte);
}
```

El miembro dato notaCorte tiene el modificador static y por tanto está ligado a la clase más que a cada uno de los objetos de dicha clase. Se puede acceder a dicho miembro con la siguiente sintaxis

`Nombre_de_la_clase.miembro_estático`

Si ponemos

```
Alumno.notaCorte=6.5;
for(int i=0; i<alumnos.length; i++){
    System.out.println("nota de corte "+alumnos[i].notaCorte);
}
```

Veremos que todos los objetos de la clase Alumno habrán cambiado el valor del miembro dato estático notaCorte a 6.5.

Un miembro dato estático de una clase se puede acceder desde un objeto de la clase, o mejor, desde la clase misma.

6.7. Relaciones entre clases

6.8. Interfaces

Una interfaz en java es un conjunto de métodos abstractos y propiedades. En ellas se especifica qué se debe hacer pero no su implementación. Serán las clases que implementen estas interfaces las que describan la lógica del comportamiento de los métodos.

Ventajas al utilizar interfaces:

1. Se organiza la programación.
2. Obligar a que ciertas clases utilicen los mismos métodos (nombres y parámetros).
3. Establecer relaciones entre clases que no estén relacionadas.

¿Cómo usarlas?

Java utiliza dos palabras reservadas para trabajar con interfaces que son interface e implements. Para declarar una interfaz se debe seguir el siguiente orden:

```
modificador_acceso interface NombreInterfaz
{
    código de interfaz
}
```

El modificador de acceso permitido dentro de una interfaz es public o puede no existir. Los atributos que definamos en el cuerpo de la interfaz serán atributos de tipo constante en las clases en las que se implemente.

Para implementar una interfaz en una clase se debe seguir el siguiente orden:

```
modificador_acceso NombreClase implements NombreInterfaz1 [, NombreInterfaz2]
```

NOTA: Una clase puede implementar varias interfaces, separando los nombres por comas.

Ejemplo:

```
Definición de una interfaz:
    interface Nave
    {
        public static final int VIDA = 100;

        public abstract void moverPosicion (int x, int y);
        public abstract void disparar();
        .....
    }
```

Uso de la interfaz definida:

```
    public class NaveJugador implements Nave
    {
        public void moverPosicion (int x, int y)
        {
            //Implementación del método
        }
        public void disparar()
        {
            //Implementación del método
        }
        .....
    }
```

En resumen los métodos forman la interfaz del objeto con el mundo exterior, un ejemplo, los botones en el frente del televisor, son la interfaz entre el usuario y el cableado eléctrico en el otro lado de la carcasa de plástico. Presiona el botón ”POWER”para encender o apagar el televisor.

En su forma más común, una interfaz es un grupo de métodos relacionados con cuerpos vacíos. El comportamiento de una bicicleta, si se especifica como una interfaz, podría aparecer como sigue:

```
interface Bicicleta

{
       void changeCadence(int newValue); // Revoluciones de la rueda por minuto
       void changeGear(int newValue);
       void speedUp(int increment);
       void applyBrakes(int decrement);
}
```

Para implementar esta interfaz, el nombre de la clase debe cambiar (a una marca concreta de la bicicleta, por ejemplo, como ACMEBicycle), y tendrá que utilizar la palabra clave implements en la declaración de la clase:

```
ACMEBicycle clase implements Bicicleta
(// resto de esta clase implementada como antes)
```

Al implementar una interfaz esto permite que la clase se vuelva más formal sobre el comportamiento que se compromete a proporcionar.

6.9. Herencia

La herencia es un mecanismo que permite la definición de una clase a partir de la definición de otra ya existente. La herencia permite compartir automáticamente métodos y datos entre clases, subclases y objetos.

La herencia está fuertemente ligada a la reutilización del código en la OOP. Esto es, el código de cualquiera de las clases puede ser utilizado sin más que crear una clase derivada de ella, o bien una subclase.

Hay dos tipos de herencia: Herencia Simple y Herencia Múltiple. La primera indica que se pueden definir nuevas clases solamente a partir de una clase inicial mientras que la segunda indica que se pueden definir nuevas clases a partir de dos o más clases iniciales. Java sólo permite herencia simple.

6.9.1. Subclases

El concepto de herencia conduce a una estructura jerárquica de clases o estructura de árbol, lo cual significa que en la OOP todas las relaciones entre clases deben ajustarse a dicha estructura.

En esta estructura jerárquica, cada clase tiene sólo una clase padre. La clase padre de cualquier clase es conocida como su superclase. La clase hija de una superclase es llamada una subclase.

* Una superclase puede tener cualquier número de subclases.

* Una subclase puede tener sólo una superclase.

6.9.2. Sobreescritura de métodos

Cada vez que se tiene una clase que hereda un método de una superclase, se tiene la oportunidad de sobreescribir el método (a menos que dicho método esté marcado como final). El beneficio clave al sobreescribir un método heredado es la habilidad de definir un comportamiento específico para los objetos de la subclase.Veamos un ejemplo de la sobreescritura de un método heredado:

```
public class Animal {

   public void comer(){
      System.out.println("Animal comiendo...");
   }
}

class Caballo extends Animal{
   public void comer(){
      System.out.println("Caballo comiendo...");
   }
}
```

Al momento de que Caballo hereda de la clase Animal obtiene el método comer() definido en Animal, sin embargo, se desea especificar un poco más el comportamiento de Caballo al momento de llamar a comer(), por lo tanto se define un método con el mismo nombre dentro de la clase Caballo. Debido a que ambos métodos tienen el mismo nombre, para saber qué método se invocará en tiempo de ejecución es necesario saber a qué objeto se está refiriendo. P. ej.:

```
public static void main(String... args){
```

```
   Animal a = new Animal();
   Caballo c = new Caballo();
   a.comer();
   c.comer();
}
```

Al ejecutar el código anterior obtenemos lo siguiente:

```
Animal comiendo...
Caballo comiendo...
```

Ahora en su versión polimórfica:

```
public static void main(String... args){
   Animal a = new Caballo();
   Caballo c = new Caballo();
   a.comer();
   c.comer();
}
```

Obtenemos lo siguiente:

```
Caballo comiendo...
Caballo comiendo...
```

En la primera ejecución del método tenemos una referencia a un Animal pero el objeto es un Caballo, por lo tanto, el método invocado es la versión definida en la clase Caballo. Es importante mencionar que al momento de invocar un método sobre una referencia Animal en un objeto Caballo solamente se podrá ejecutar el métodos si la clase Animal lo define, p. ej.:

```
class Animal {

   public void comer(){
      System.out.println("Animal comiendo...");
   }
}

class Caballo extends Animal{
   public void comer(){
      System.out.println("Caballo comiendo...");
   }
   public void relinchar(){
      System.out.println("Caballo relinchando...");
   }
```

```
}

class ProbarMetodos{

   public static void main(String... args){
      Animal a = new Caballo();
      Caballo c = new Caballo();
      a.comer();
      c.comer();
      a.relinchar(); //error!
   }
}
```

Aún cuando la clase Caballo define un método llamado relinchar(), la clase Animal no sabe que dicho método existe, por lo tanto, el compilador arrojará un error cuando se intente invocar al método relinchar() desde una referencia Animal, no importa que el objeto Caballo sí lo tenga.

Reglas para sobreescribir un método::

Las reglas básicas para la sobreescritura de métodos son las siguientes:

+ La lista de argumentos del método debe ser exactamente la misma.

+ El tipo de retorno debe de ser el mismo o un subtipo del tipo de retorno declarado originalmente.

+ El nivel de acceso no debe de ser más restrictivo.

+ El nivel de acceso puede ser menos restrictivo.

+ Los métodos de instancia pueden ser sobreescritos solamente si han sido heredados por la subclase.

+ Los métodos sobreescritos pueden arrojar cualquier excepción no verificada(de tiempo de ejecución) por el compilador.

+ Los métodos sobreescritos NO pueden arrojar excepciones verificadas por el compilador.

+ No se puede sobreescibir un método marcado como final.

+ No se puede sobreescibir un método marcado como estático (static).

+ Si un método no puede ser heredado, no puede ser sobreescrito.

Invocar la versión de la superclase de un método sobreescrito::

En algunas ocasiones necesitamos invocar al método escrito en la superclase en lugar de la versión que hemos sobreescrito, para ello se utiliza la palabra super seguida de un punto(.) y posteriormente el nombre del método a invocar, p. ej.:

```
class Caballo extends Animal{
      public void comer(){
      super.comer();
   }
}
```

6.9.3. Visibilidad

6.9.4. ejemplo

Como ejemplo básico de herencia implementamos la clase Ciudadano hereda desde la clase Persona, para tomar funcionalidad y extenderla

```
class Humano {
    protected String nombre;
    protected String apellido;
    public Humano(String nombre,String apellido) {
        this.nombre = nombre;
        this.apellido = apellido;
    }
    public String nombreCompleto() {
        return this.apellido + ", " + this.nombre;
    }
    public String identificacion() {
        return this.nombreCompleto();
    }
}

class Ciudadano extends Humano {
    protected String documento;

    public Ciudadano(String nombre,String apellido, String documento) {
        super(nombre,apellido);
        this.documento = documento;
    }

    public String identificacion() {
```

```
        return super.identificacion() + ", documento: " + this.documento;
    }
}


public class Herencia {
    public static void  main (String args[]) {
        Humano a = new Humano("Emilio","Rosso");
        Ciudadano b = new Ciudadano("Emilio","Rosso","3052454545");
        Humano [] arregloDeHumanos;
        arregloDeHumanos.push(a);
        arregloDeHumanos.push(b);
        identificarPolimorfico(arregloDeHumanos);
    }

    public static void identificarPolimorfico(
      Humano [] arregloDeHumanos) {
        for(int i = 0; i < arregloDeHumanos.length ; i++) {
            System.out.println("Identificando: " +
            arregloDeHumanos[i].identificacion());
        }
    }

}
```

6.10. Polimorfismo

El polimorfismo consiste en obtener un mismo método con diferentes funciones, es decir, suponiendo que tengamos una clase Saludo con el método Saludo el cual sería su constructor, pero en este caso tendríamos varios constructores pero siempre siguiendo las reglas de independencia en cuestión a los parámetros de cada constructor para que así puedan diferenciarse uno de otro, sin embargo cada uno tendrá diferentes procesos internos en sí mismo.

```
class Saludo {
public Saludo(){
       MensajeSaludo="Hola Amigo";
}
public Saludo(String Palabra){
       MensajeSaludo=Palabra;
```

```
}
public Saludo(String Palabra, String Nombre){
MensajeSaludo=Palabra.concat(" ").concat(Nombre);
}
}
```

6.10.1. Polimorfismo usando herencia

Mediante el polimorfismo podemos variar las propiedades de un objeto que a heredado de otra clase, veamos esto siguiendo un ejemplo :

```
public class concede {
public void estoyEscribiendo(){
System.out.println("Escribo una nota");
}
}
public class acepta extends concede {

}

    public class Herencia {
        public static void main(String[] args) {
            acepta herencia = new acepta();
            concede herenciaDos = new concede();
            herencia.estoyEscribiendo();
            herenciaDos.estoyEscribiendo();
        }
    }
```

Al ejecutar el anterior código nos desplegara dos veces el mensaje:

```
    Escribo una nota
    Escribo una nota
```

Mediante el polimorfismo podemos hacer que la clase acepta que es la que extiende a todas las propiedades de concede pueda modificar aquello datos que recibe, en nuestro ejemplo haremos lo siguiente:

package conceptos;

public class acepta extends concede public void estoyEscribiendo() System.out.println(”Tu también escribes una nota”);

Este es un ejemplo claro de polimorfismo ya que la clase acepta esta heredando todo lo que tiene la clase concede, lo que realizamos en esta clase es añadir un polimorfismo y sobre escribir lo que heredamos, es decir tenemos estoyEscribiendo que hemos heredado pero estamos sobrescribiendo lo que he heredado, en pocas palabras estoy sobrescribiendo el método que he heredado cambiando el mensaje que voy a desplegar.

6.10.2. Polimorfismo usando Interfaces

6.11. Revisión de Excepciones

En esta sección haremos una revisión de las excepciones haciendo uso de todas las herramientas vistas en el capítulo

6.11.1. Ejemplo 1

Como ejemplo de aplicación de la herencia y de manejo de interfaces haremos un ventana de tipo windows que maneje algunas acciones que se desencadenaran cuando se presione un botón o se de enter en un cuadro de texto. Ente ejemplo lo llevaremos a cabo en 4 pasos que se detallan a continuación.

Paso 1

Definir Ventana y crear una Ventana tipo Windows.

```
import javax.swing.*;

public class MiVentana extends JFrame{
    public MiVentana() {
        super("Mi Ventana");
        this.setSize(400, 400);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setVisible(true);
    }
}
```

Paso 2

Agregar botón, campo de texto y una etiqueta.

```
import javax.swing.*;
import java.awt.*;
```

```
public class MiVentana extends JFrame{
    private JButton miBoton;
    private JTextField miCampoTexto;
    private JLabel miEtiqueta;

    public MiVentana() {
        super("Mi Ventana");

        this.definirVentana();

        this.setSize(400, 400);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setVisible(true);
    }

    public void definirVentana() {
        this.setLayout(new FlowLayout());
        miBoton = new JButton("Enviar");
        miCampoTexto = new JTextField(20);
        miEtiqueta = new JLabel();

        this.add(miCampoTexto);
        this.add(miBoton);
        this.add(miEtiqueta);
    }
}
```

Paso 3

Abstracción y eventos en Java

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;


public class MiVentana extends JFrame implements ActionListener {
    private JButton miBoton;
    private JTextField miCampoTexto;
    private JLabel miEtiqueta;
```

```
    public MiVentana() {
        super("Mi Ventana");

        this.definirVentana();

        this.setResizable(false);
        this.setLocationRelativeTo(null);
        this.setSize(400, 400);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setVisible(true);
    }

    public void definirVentana() {
        this.setLayout(new FlowLayout());
        miBoton = new JButton("Enviar");
        miCampoTexto = new JTextField(20);
        miEtiqueta = new JLabel();

        this.add(miCampoTexto);
        this.add(miBoton);
        this.add(miEtiqueta);
        miBoton.addActionListener(this);
    }

    public void actionPerformed(ActionEvent arg0) {
        if(arg0.getSource()== miBoton) {
           miEtiqueta.setText(miCampoTexto.getText());
           miCampoTexto.setText("");
        }
    }
}
```

Paso 4

Finalmente haremos uso de las excepciones apara que nuestro programa este protegido

```
    public void actionPerformed(ActionEvent arg0) {
        if(arg0.getSource() == miBoton) {
            String a = miCampoTexto.getText();
            double val = 0;
            try {
```

```
                val = Double.parseDouble(a);
            }
            catch (NumberFormatException ex) {
                miEtiqueta.setText("Error");
                miCampoTexto.setText("");
                return;
            }

            double val2 = val*1.1;
            miEtiqueta.setText("El 10% de " + val + " es " + val2);
        }
    }
```

Paso 5

Otra manera de hacer lo mismo es:

```
iimport javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class MiVentana2 extends JFrame {
    private JButton miBoton;
    private JTextField miCampoTexto;
    private JLabel miEtiqueta;

    public MiVentana2() {
        super("Mi Ventana 2");

        this.definirVentana();

        this.setResizable(false);
        this.setLocationRelativeTo(null);
        this.setSize(400, 400);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setVisible(true);
    }

    public void definirVentana() {
        this.setLayout(new FlowLayout());
        miBoton = new JButton("Enviar");
        miCampoTexto = new JTextField(20);
```

```
        miEtiqueta = new JLabel();

        this.add(miCampoTexto);
        this.add(miBoton);
        this.add(miEtiqueta);
        miBoton.addActionListener((ActionListener) new MiBotonListener());
        miCampoTexto.addKeyListener((KeyListener) new MiCampoTextoKeyListener());
    }


    private class MiBotonListener implements ActionListener {
        public void actionPerformed(ActionEvent arg0) {
            String a = miCampoTexto.getText();
            double val = 0;
            try {
                val = Double.parseDouble(a);
            }
            catch (NumberFormatException ex) {
                miEtiqueta.setText("Error");
                miCampoTexto.setText("");
                return;
            }

            double val2 = val*1.1;
            miEtiqueta.setText("El 10% de " + val + " es " + val2);
        }
    }

    private class MiCampoTextoKeyListener implements KeyListener {
        public void keyTyped(KeyEvent arg0) {
            if(arg0.getKeyChar() == '\n' ) {
                String a = miCampoTexto.getText();
                double val = 0;
                try {
                    val = Double.parseDouble(a);
                }
                catch (NumberFormatException ex) {
                    miEtiqueta.setText("Error");
                    miCampoTexto.setText("");
                    return;
                }
```

```
                double val2 = val*1.1;
                miEtiqueta.setText("El 10% de " + val + " es " + val2);
            }

        }

        public void keyPressed(KeyEvent arg0) {
        }

        public void keyReleased(KeyEvent arg0) {

        }
    }
}
```

6.11.2. Ejemplo 2

Conversión de Grados Centrigrados a Grados Farenhait.

```
import java.awt.FlowLayout;
import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JTextField;


public class Temperatura extends JFrame{
    private JTextField campo1;
    private JTextField campo2;
    private JLabel etiqueta1;
    private JLabel etiqueta2;

    public Temperatura() {
        super("Conversion");

        this.definirVentana();

        this.setResizable(false);
        this.setSize(200, 300);
```

```
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setVisible(true);
    }

    public void definirVentana() {
        this.setLayout(new FlowLayout());
        campo1 = new JTextField(15);
        campo2 = new JTextField(15);
        etiqueta1 = new JLabel("Grados C");
        etiqueta2 = new JLabel ("Frados F");

        this.add(etiqueta1);
        this.add(campo1);
        this.add(etiqueta2);
        this.add(campo2);

        campo1.addKeyListener((KeyListener) new campo1_KeyListener());
        campo2.addKeyListener((KeyListener) new campo2_KeyListener());
    }

    private class campo1_KeyListener implements KeyListener {

        public void keyTyped(KeyEvent arg0) {
            if(arg0.getKeyChar() == '\n' ) {
                String a = campo1.getText();
                double val = 0;
                try {
                    val = Double.parseDouble(a);
                }
                catch (NumberFormatException ex) {
                    campo1.setText("error");
                    return;
                }
                val = val*9.0/5.0 + 32;
                campo2.setText("" +val);
            }
        }
        public void keyPressed(KeyEvent arg0) {
        }

        public void keyReleased(KeyEvent arg0) {
```

```
        }
    }
    private class campo2_KeyListener implements KeyListener {

        public void keyTyped(KeyEvent arg0) {
            if(arg0.getKeyChar() == '\n' ) {
                String a = campo2.getText();
                double val = 0;
                try {
                    val = Double.parseDouble(a);
                }
                catch (NumberFormatException ex) {
                    campo2.setText("error");
                    return;
                }

                val = (val - 32)*5.0/9.0;
                campo1.setText("" +val);
            }
        }
        public void keyPressed(KeyEvent arg0) {
        }
        public void keyReleased(KeyEvent arg0) {

        }
    }

    static public void main(String args[]){
        Temperatura a = new Temperatura();
    }
}
```

6.11.3. Ejemplo 3

En este ejemplo se hace uso de la clase JPanel para crear paneles de dibujo así como de control. También se implementas los eventos de ratón para manejo en una pantalla de dibujo.

Paso 1

Creación del panel de dibujo

```
package Dibujo;

import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.awt.event.MouseMotionListener;
import javax.swing.JPanel;

public class Lienzo extends JPanel implements MouseMotionListener, MouseListener {
    private int x0, y0, x1, y1;

    public Lienzo(){
        this.setBackground(Color.WHITE); // Pone un color de fondo
        this.setPreferredSize(new Dimension(400, 400)); // Tamaño del lienzo
        this.addMouseMotionListener(this);
        this.addMouseListener(this);
    }

    @Override
    public void paint(Graphics g){
        super.paint(g);

        g.drawLine(x0, y0, x1, y1);
    }

    public void mouseDragged(MouseEvent arg0) {
        x1 = arg0.getX();
        y1 = arg0.getY();
        repaint();
    }

    public void mouseMoved(MouseEvent arg0) {
    }

    public void mouseClicked(MouseEvent arg0) {
    }
```

```
    public void mousePressed(MouseEvent arg0) {
        x0 = arg0.getX();
        y0 = arg0.getY();
    }

    public void mouseReleased(MouseEvent arg0) {
    }

    public void mouseEntered(MouseEvent arg0) {
    }

    public void mouseExited(MouseEvent arg0) {
    }
}
```

Para probar el Lienzo hacemos

```
package Dibujo;

import javax.swing.JFrame;

public class PruebaLienzo extends JFrame {
    public PruebaLienzo() {
        super("Prueba Lienzo");
        Lienzo miLienzo = new Lienzo();
        add(miLienzo);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setVisible(true);
        pack();
    }
    static public void main(String args[]) {
        new PruebaLienzo();
    }
}
```

Paso 2

Creación de panel de control con botones

```
package Dibujo;

import java.awt.Color;
import java.awt.GridLayout;
```

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.BorderFactory;
import javax.swing.JButton;
import javax.swing.JPanel;

public class Botones extends JPanel implements ActionListener {
    static private int botonPrecionado;
    private String nombres[];

    public Botones(String unTitulo, String unNombre[], int N) {
        super(new GridLayout(N/2, 2));
        int i;

        nombres = new String[N];
        for(i=0; i<N; i++)
            nombres[i] = unNombre[i];

        setBackground(Color.lightGray);
        setBorder(BorderFactory.createTitledBorder(unTitulo));
        JButton b;

        for(i=0; i<N; i++ ) {
            b = new JButton(unNombre[i]);
            b.addActionListener(this);
            this.add(b);
        }
    }

    public void actionPerformed(ActionEvent arg0) {
        int i;

        for(i=0; i<nombres.length; i++)
            if(nombres[i].equals(arg0.getActionCommand())) {
                botonPrecionado = i;
                break;
            }
        System.out.println(nombres[botonPrecionado]);
    }

}
```

Paso 3

Panel de control con botones de radio

```
package Dibujo;

import java.awt.Color;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.BorderFactory;
import javax.swing.ButtonGroup;
import javax.swing.JPanel;
import javax.swing.JRadioButton;

public class SeleccionPanel extends JPanel implements ActionListener{
    static private int seleccion;
    private String nombres[];

    public SeleccionPanel(String Nombre, String[] opciones, int N){
        super(new GridLayout(N/2, 2));
        int i;

        nombres = new String[N];
        for(i=0; i<N; i++)
            nombres[i] = opciones[i];

        setBackground(Color.lightGray);
        setBorder(BorderFactory.createTitledBorder(Nombre));
        ButtonGroup group = new ButtonGroup();
        JRadioButton option;

        for(i=0; i<N; i++) {
          option = new JRadioButton(opciones[i]);
          option.addActionListener(this);
          group.add(option);
          add(option);
      }
    }

    public void actionPerformed(ActionEvent arg0) {
        int i;
```

```
        for(i=0; i<nombres.length; i++)
            if(nombres[i].equals(arg0.getActionCommand())) {
                seleccion= i;
                break;
            }
        System.out.println(nombres[seleccion]);

    }
}
```

Paso 4

Finalmente creamos un programa para probar todas las clases implementadas.

```
package Dibujo;

import java.awt.BorderLayout;
import java.awt.GridLayout;
import javax.swing.JFrame;
import javax.swing.JPanel;

public class PruebaTodo extends JFrame {
    private Botones misBotones;
    private SeleccionPanel miSeleccion;
    private Lienzo miLienzo;

    public PruebaTodo () {
        JPanel AreaControl = new JPanel(new GridLayout(2, 1));
        misBotones = new Botones("Botones",
           new String[] {"uno", "dos", "tres", "cuatro"}, 4);
        miSeleccion = new SeleccionPanel("Algo",
           new String[] {"a", "b", "c", "d", "e"}, 5);

        AreaControl.add(misBotones);
        AreaControl.add(miSeleccion);

        miLienzo = new Lienzo();
        this.add(AreaControl, BorderLayout.WEST);
        this.add(miLienzo,BorderLayout.EAST);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        pack();
```

```
        setVisible(true);
    }

    static public void main(String args[]) {
        new PruebaTodo();
    }
}
```

Uso de Collections y Generics

7.1. Uso de == y del método equal, y sobreescritura del método hashCode

Utilizar el símbolo == para comparar objetos resulta inoperante dado que hace una comparación de los apuntadores de cada uno de los objetos. En lugar de esto utilizamos un método especial del que disponen todos los objetos denominado equals. Equals se basa en una definición de igualdad que ha de estar establecida. Por ejemplo, podríamos definir que dos objetos son iguales si contienen el mismo número de bolígrafos, de la misma marca, modelo y color. Para algunos tipos de objetos, como los String, la definición de igualdad que viene incorporada en el API de Java nos va a resultar suficiente.

El siguiente es un ejemplo de como se debe implementar el uso de este método

```
public class Compara_Personas {
    private String nombre;
    private int clave;

    public Compara_Personas(String unNombre, int unaClave){
        nombre = unNombre;
        clave = unaClave;
    }

    public String imprime(){
        String aux ;

        aux = nombre + " " + clave;

        return aux;
    }
```

```
    @Override
    public boolean equals(Object a) {
        if(a instanceof Compara_Personas && this instanceof Compara_Personas) {
            if(this.nombre.equals(((Compara_Personas)a).nombre )) return true;
            else return false;
        }
        else return false;
    }

    @Override
    public int hashCode() {
        int hash = 7;
        hash = 97 * hash + (this.nombre != null ? this.nombre.hashCode() : 0);
        return hash;
    }

    static public void main(String args[]) {
        Compara_Personas a = new Compara_Personas("Juan", 123);
        Compara_Personas b = new Compara_Personas("Juan", 123);

        System.out.println(a.equals(b));
    }
}
```

7.2. Clases de la API Collections (Set, List, Map, etc)

Para trabajar con colecciones en Java podemos hacer uso del framework Collections. Las clases e interfaces que componen este framework se encuentran en los paquetes java.util y java.util.concurrent. Todas hacen uso del polimorfismo paramétrico que proporciona generics; concepto que tratamos ampliamente en la entrada Generics en Java.

La raíz de la jerarquía de colecciones es la interfaz Collection<E>. De entre las interfaces que extienden Collection<E> las más interesantes son List<E>, Set<E> y Queue<E> que definen, respectivamente, listas, conjuntos y colas.

La lista es una de las colecciones más básicas, se trata de una estructura de datos secuencial, en la que cada elemento tiene una posición o índice, y que permite elementos duplicados. Set<E> es la interfaz utilizada para modelar los conjuntos matemáticos; como en estos, los elementos no tienen un orden, y no se permiten elementos duplicados. Una cola o

Queue<E>, por último, es una estructura de datos de tipo FIFO (First in first out) lo que significa que el primer elemento en introducirse en la cola será el elemento que se devolverá al extraer por primera vez de la cola, y así sucesivamente (en realidad existen implementaciones de Queue<E> que no son FIFO, pero eso queda fuera del enfoque de esta entrada).

Otra estructura de datos que forma parte del framework, aunque no deriva de Collection<E>, dado que no se trata de una colección per se sino de un mapeo, es la que define la interfaz Map<K,V>, el conocido diccionario o matriz asociativa, en el que cada valor tiene asociado una clave que usaremos para recuperar el elemento, en lugar de un índice como el caso de las listas. Por ejemplo podríamos tener un mapeo en el que las claves fueran los días de la semana y los valores, el número de líneas de código que escribimos.

Para cada una de las interfaces que definen los tipos de colección disponibles tenemos una clase abstracta que contiene la funcionalidad común a varias implementaciones de la interfaz. Estas son AbstractList<E>, AbstractSet<E>, AbstractQueue<E> y AbstractMap<K,V>.

En esta entrada trataremos las listas y los mapeos, por ser las estructuras de datos más utilizadas.

7.2.1. List

Hay varias clases que implementan la interfaz List <E >. Las más utilizadas habitualmente son ArrayList<E> y la vieja conocida Vector<E>, que forma parte del framework Collections desde Java 1.2. Ambas extienden de AbstractList<E> y tienen una interfaz muy parecida.

Como en el caso de StringBuffer y StringBuilder la principal diferencia entre ambas clases es que Vector<E> es sincronizada y ArrayList<E> no sincronizada. Debido a esto no tendremos que preocuparnos de problemas de sincronización a la hora utilizar varios hilos con Vector<E> (hasta cierto punto). Pero a cambio, y por la misma razón, el rendimiento de Vector<E> es mucho peor que el de ArrayList<E>. Ergo, si no necesitamos sincronización o no estamos trabajando con más de un hilo de ejecución, lo adecuado es utilizar ArrayList<E>.

En todo caso, si trabajamos con varios hilos, también existe la posibilidad de sincronizar ArrayList<E> de forma externa o utilizar el método Collections.synchronizedList(List<T> list) para crear un wrapper que añade sincronización a ArrayList<E>. Vector<E>, no obstante, es ligeramente más rápido que un ArrayList<E> sincronizado con synchronizedList.

Tanto ArrayList<E> como Vector<E> utilizan un objeto Array<E> para almacenar los elementos internamente. Las colecciones de tipo Array<E>, sin embargo, tienen un ta-

maño fijo que se determina de antemano al llamar al constructor. Esto implica que cuando sobrepasamos dicho tamaño hay que crear un nuevo Array<E> y copiar el antiguo, lo que puede ser muy costoso computacionalmente. Por otro lado, si la capacidad máxima inicial del array fuera demasiado grande, estaríamos desperdiciando espacio.

Por defecto ArrayList<E> y Vector<E> utilizan arrays con capacidad para 10 elementos. Cuando el número de elementos sobrepasa la capacidad disponible Vector<E> dobla el tamaño del array interno, mientras que ArrayList<E> utiliza la fórmula (capacidad * 3) / 2 + 1.

Para indicar valores que se ajusten lo máximo posible al uso que vamos a hacer de nuestra lista tanto Vector<E> como ArrayList<E> cuentan, además de con un constructor vacío y un constructor al que pasar una colección con los elementos con los que inicializar el vector o la lista, con un constructor al que pasar un entero con la capacidad inicial de la colección.

Vector<E> cuenta también con un constructor extra mediante el que indicar el incremento en capacidad a utilizar cuando el número de elementos rebase la capacidad del vector, lo cuál puede ser muy interesante.

En cualquier caso, independientemente del constructor que utilicemos, al crear cualquier colección es una buena práctica utilizar el tipo más genérico posible para referirnos al nuevo objeto. Por ejemplo, sería preferible utilizar: view plaincopy to clipboardprint?

List<String> lista = new ArrayList<String>();

a view plaincopy to clipboardprint?

ArrayList<String> lista = new ArrayList<String>();

ya que al utilizar la primera versión, si es necesario en un futuro, podremos migrar fácilmente a una implementación distinta de List<E> gracias al polimorfismo.

Otra implementación para listas muy utilizada es LinkedList<E>, diseñada para obtener un mejor rendimiento cuando se insertan o eliminan muchos elementos de la mitad de la colección. Este tipo de lista, como su nombre indica, utiliza una lista doblemente enlazada internamente, en lugar de un array. En una lista enlazada tendremos un nodo por cada elemento introducido y cada uno de estos nodos tendrá el valor asociado a esa posición y una referencia al siguiente nodo. De esta forma, al insertar o eliminar un elemento cualquiera, basta con actualizar la referencia al siguiente nodo, en lugar de mover cada elemento a una posición superior o inferior como ocurriría con una implementación basada en Array<E> como Vector<E> y ArrayList<E>.

Sin embargo en las listas enlazadas, a diferencia de en los arrays, que se implementan directamente en hardware, el acceso a los elementos se realiza de forma secuencial, siguiendo las referencias de cada uno de los nodos, por lo que el acceso a una posición aleatoria de la

colección es mucho más lento. Esta clase de colección, por tanto, tan solo debería utilizarse en casos en los que vayamos a insertar o eliminar muchos elementos de la mitad de la colección, pero no vayamos a realizar muchos accesos aleatorios.

Como Vector<E> y ArrayList<E>, y de hecho como todos los tipos de colección concretos, LinkedList<E> cuenta con un constructor vacío y un constructor que recibe una colección con los valores iniciales para la lista. De esta forma es sencillo copiar valores de una colección a otra o convertir un tipo de colección en otra distinta.

La interfaz que nos proporciona es muy parecida a la de Vector<E> y ArrayList<E>. De hecho esta clase extiende de la clase AbstractSequentialList<E>, que a su vez es hija de AbstractList<E>, la clase padre de Vector<E> y ArrayList<E>.

Por último, otra clase interesante dentro de la jerarquía de List<E> es CopyOnWriteArrayList<E>, perteneciente al paquete java.util.concurrent, y que, a diferencia de ArrayList<E> y LinkedList<E> es thread-safe, como Vector<E>, y ofrece un muy buen rendimiento a la hora de leer de la colección. Su desventaja es que el array que utiliza por debajo es inmutable, lo que hace que tengamos que crear un nuevo array cada vez que la colección se modifica. Puede ser una buena alternativa si utilizamos varios hilos de ejecución y no vamos a modificar mucho la lista.

De entre los métodos comunes a las clases ArrayList<E>, Vector<E>, LinkedList<E> y CopyOnWriteArrayList<E> los más interesantes son los siguientes:

- boolean add(E o): Añade un nuevo elemento al final de la colección.
- boolean add(int index, E element): Añade un nuevo elemento en la posición especificada.
- boolean addAll(Collection<? extends E> c): Añade todos los elementos de la colección especificada a esta colección.
- void clear(): Elimina todos los elementos de la colección.
- boolean contains(Object o): Comprueba si el elemento especificado es parte de la colección.
- E get(int index): Recupera el elemento que se encuentra en la posición espeficicada.
- int indexOf(Object o): Devuelve la primera posición en la que se encuentra el elemento especificado en la colección, o -1 si no se encuentra.
- int lastIndexOf(Object o): Devuelve la última posición en la que se encuentra el elemento especificado en la colección, o -1 si no se encuentra.
- E remove(int index): Elimina el elemento de la posición indicada.

- boolean remove(Object o): Elimina la primera ocurrencia del elemento indicado. Si se encontró y se borró el elemento, devuelve true, en caso contrario, false.
- E set(int index, E element): Reemplaza el elemento que se encuentra en la posición indicada por el elemento pasado como parámetro. Devuelve el elemento que se encontraba en dicha posición anteriormente.
- int size(): Devuelve el número de elementos que se encuentran actualmente en la colección.

Ejemplo 1

Crear una lista de valores enteros probar algunas de las funciones validas para una Lista

```
import java.util.ArrayList;
import java.util.List;

public class ejemplo_lista {

    static public void prueba() {
        List <Integer> lista = new ArrayList<Integer>();
        List <Integer> aux = new ArrayList <Integer> ();

        // agrega un elemento

        lista.add(new Integer(1));

        // agrega un elemento en la posicion indicada

        lista.add(0, new Integer(2));

        // agrega todos los elementos a otra lista

        aux.addAll(lista);

        for(Integer b: lista)
            System.out.println(b);

        // elimina todos los elementos de la lista

        aux.clear();
```

```
        for(Integer b: aux)
            System.out.println(b);
    }

    static public void main(String args[]){
        prueba();
    }
}
```

Ejemplo 2

Crear un conjunto con valores enteros probar algunas de las funciones validas para un Conjunto

```
import java.util.HashSet;
import java.util.Set;

public class ejemplo_set {
    static public void prueba() {
        Set <String> A = new HashSet <String> ();
        A.add("c");
        A.add("a");
        A.add("a");
        A.add("b");

        // Funcion para preguntar si contiene

        System.out.println(A.contains("a"));

        for(String b: A)
            System.out.println(b);
    }
    static public void main(String args[]){
        prueba();
    }
}
```

7.2.2. Map

La interfaz Map<K,V> y las clases que la implementan vienen a reemplazar la antigua clase Dictionary.

HashMap<K,V> es el tipo de mapeo más sencillo y probablemente el más usado. Es la clase a utilizar si queremos asociar pares de claves y valores sin orden, sin más. Internamente, como su nombre indica, utiliza un hash para almacenar la clave. No permite claves duplicadas, pero si utilizar null como clave.

Hashtable<K,V> es una vieja conocida del JDK 1.0, que, como Vector<E> pasó a formar parte del framework de colecciones en Java 1.2. Esta clase es muy similar a HashMap<K,V>, con la excepción de que es sincronizada y que no acepta utilizar el valor null como clave. Como en el caso de Vector<E> contra ArrayList<E>, nos interesará utilizar HashMap<K,V> siempre que no estemos utilizando varios hilos de ejecución. En el caso de que necesitemos sincronización, otra opción es utilizar el método Collections.synchronizedMap sobre un HashMap, que funciona de forma similar a Collections.synchronizedList.

LinkedHashMap<K,V> es una clase introducida con el J2SE 1.4 que extiende HashMap<K,V> y utiliza una lista doblemente enlazada para poder recorrer los elementos de la colección en el orden en el que se añadieron. Esta clase es ligeramente más rápida que HashMap<K,V> a la hora de acceder a los elementos, pero es algo más lenta al añadirlos.

Por último tenemos TreeMap<K,V>, en el que los pares clave-valor se almacenan en un árbol ordenado según los valores de las claves. Como es de esperar es la clase más lenta a la hora de añadir nuevos elementos, pero también a la hora de accederlos.

De entre los métodos comunes a las cuatro clases los más utilizados son:

- void clear(): Elimina todos los elementos de la colección.
- boolean containsKey(Object key): Comprueba si la clave especificada se encuentra en la colección.
- boolean containsValue(Object value): Comprueba si el valor especificado se encuentra en la colección.
- V get(Object key): Devuelve el valor asociado a la clave especificada o null si no se encontró.
- boolean isEmpty(): Comprueba si la colección está vacía.
- Set keySet(): Devuelve un conjunto con las claves contenidas en la colección.
- V put(K key, V value): Añade un nuevo par clave-valor a la colección

- V remove(Object key): Elimina el par clave-valor correspondiente a la clave pasada como parámetro.
- int size(): Devuelve el número de pares clave-valor que contiene la colección.
- Collection values(): Devuelve una colección con los valores que contiene la colección.

Ejemplo

El siguiente ejemplo muestra el uso de Map para asociar a los jugadores de Futbol con un número de camiseta

```
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;

public class ejemplo_Map {
    static public void prueba() {
        Map <Integer, String> map = new HashMap <Integer, String>();
        map.put(1, "Casillas");
        map.put(15, "Ramos");
        map.put(3, "Pique");
        map.put(5, "Puyol");
        map.put(11, "Capdevila");
        map.put(14, "Xabi Alonso");
        map.put(16, "Busquets");
        map.put(8, "Xavi Hernandez");
        map.put(18, "Pedrito");
        map.put(6, "Iniesta");
        map.put(7, "Villa");

        Iterator it = map.keySet().iterator();

        while(it.hasNext()){
            Integer key = (Integer) it.next();
            System.out.println("Clave: " + key + " -> Valor: " + map.get(key));
        }
    }

    static public void main(String args[]) {
        prueba();
    }
```

```
}
```

7.2.3. Collections

Collections es una clase perteneciente al paquete java.util que proporciona una serie de métodos estáticos para manejar colecciones que pueden ser de mucha utilidad. A esta clase pertenecen los métodos synchronizedList y synchronizedMap que ya comentamos anteriormente. Otros métodos interesantes son:

- int binarySearch(List list, Object key): Busca un elemento en una lista ordenada utilizando un algoritmo de búsqueda binaria. El método devuelve un entero indicando la posición en la que se encuentra el elemento, o bien un número negativo si no se encontró. Este número negativo indica la posición la que se encontraría el elemento de haber estado en la colección.
- int frequency(Collection c, Object o): Devuelve el número de veces que se repite el elemento especificado en la colección.
- Object max(Collection coll): Devuelve el mayor elemento de la colección.
- Object min(Collection coll): Devuelve el menor elemento de la colección.
- boolean replaceAll(List list, Object oldVal, Object newVal): Reemplaza todas las ocurrencias en la lista de un cierto elemento por otro objeto.
- void reverse(List list): Invierte la lista.
- void shuffle(List list): Modifica la posición de distintos elementos de la lista de forma aleatoria.
- void sort(List list): Ordena una lista utilizando un algoritmo merge sort.

Ejemplo

Escribir un pequeño código que permita probar algunas de las funciones para la utilidades en la clase colecciones.

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class ejemplo_Coleccion {
    static public void prueba() {
        List <String> A = new ArrayList <String> ();
```

```
        A.add(new String("c"));
        A.add(new String("a"));
        A.add(new String("b"));

        Collections.sort(A);

        for(String b: A)
            System.out.println(b);

        System.out.println("Esta la letra b " + Collections.binarySearch(A, "b"));
        System.out.println("El minimo es "  + Collections.min(A));
        System.out.println("El maximo es "  + Collections.max(A));

        Collections.reverse(A);

        for(String b: A)
            System.out.println(b);
     }

    static public void main(String args[]){
        prueba();
    }
}
```

7.3. Generics

Antes de Java 5 cuando introducíamos objetos en una colección estos se guardaban como objetos de tipo Object, aprovechando el polimorfismo para poder introducir cualquier tipo de objeto en la colección. Esto nos obligaba a hacer un casting al tipo original al obtener los elementos de la colección.

```
import java.util.ArrayList;
import java.util.List;

public class Ejemplo {
  public static void main(String[] args) {
    List lista = new ArrayList();
    lista.add("Hola mundo");

    String cadena = (String) lista.get(0);
```

```
        System.out.println(cadena);
    }
}
```

Esta forma de trabajar no solo nos ocasiona tener que escribir más código innecesariamente, sino que es propenso a errores porque carecemos de un sistema de comprobación de tipos. Si introdujéramos un objeto de tipo incorrecto el programa compilaría pero lanzaría una excepción en tiempo de ejecución al intentar convertir el objeto en String:

```
import java.util.ArrayList;
import java.util.List;

public class Ejemplo {
    public static void main(String[] args) {
        List lista = new ArrayList();
        lista.add(22);

        String cadena = (String) lista.get(0);
        System.out.println(cadena);
    }
}
```

Podríamos utilizar el operador instanceof para comprobar el tipo del objeto antes de hacer el casting o antes de introducirlo en la colección, pero es poco elegante, añade aún más código a escribir, y no nos evitaría tener que hacer el casting.

Desde Java 5 contamos con una característica llamada generics que puede solventar esta clase de problemas. Los generics son una mejora al sistema de tipos que nos permite programar abstrayéndonos de los tipos de datos, de forma parecida a los templates o plantillas de C++ (pero mejor).

Gracias a los generics podemos especificar el tipo de objeto que introduciremos en la colección, de forma que el compilador conozca el tipo de objeto que vamos a utilizar, evitándonos así el casting. Además, gracias a esta información, el compilador podrá comprobar el tipo de los objetos que introducimos, y lanzar un error en tiempo de compilación si se intenta introducir un objeto de un tipo incompatible, en lugar de que se produzca una excepción en tiempo de ejecución.

Para utilizar generics con nuestras colecciones tan solo tenemos que indicar el tipo entre < y> a la hora de crearla. A estas clases a las que podemos pasar un tipo como "parámetro" se les llama clases parametrizadas, clases genéricas, definiciones genéricas o simplemente genéricas (generics). Veamos un ejemplo con la colección List.

```
import java.util.ArrayList;
```

```
import java.util.List;

public class Ejemplo {
  public static void main(String[] args) {
    List<String> lista = new ArrayList<String>();
    lista.add("Hola mundo");

    String cadena = lista.get(0);
    System.out.println(cadena);
  }
}
```

Este código, sin embargo, no compilaría, dado que hemos indicado que nuestra colección contendrá objetos de tipo String, y no Integer:

```
import java.util.ArrayList;
import java.util.List;

public class Ejemplo {
  public static void main(String[] args) {
    List<String> lista = new ArrayList<String>();
    lista.add(new Integer(22));

    Integer numero = lista.get(0);
    System.out.println(numero);
  }
}
```

Algo a tener en cuenta es que el tipo parámetro debe ser una clase; no podemos utilizar tipos primitivos. Esto, sin embargo, no es ningún problema gracias a las características de autoboxing y unboxing de Java 5. En el código siguiente, por ejemplo, se crea un objeto Integer intermedio de forma transparente, que es el objeto que se introducirá en realidad en la colección, y no un int de valor 22, como podría parecer. Posteriormente el objeto Integer se vuelve a convertir en int automáticamente.

```
import java.util.ArrayList;
import java.util.List;

public class Ejemplo {
  public static void main(String[] args) {
    List<Integer> lista = new ArrayList<Integer>();
    lista.add(22);
```

```
      int numero = lista.get(0);
      System.out.println(numero);
    }
  }
```

Ahora, ¿cómo podemos crear nuestras propias clases parametrizadas? ¿cómo funciona a partir de Java 5 una colección como List? Sencillo. Solo tenemos que añadir el tipo parámetro después del nombre de la clase, y utilizar el nombre de este tipo genérico donde usaríamos un tipo concreto (con algunas excepciones). Por convención se suele utilizar una sola letra mayúscula para el tipo genérico.

Como ejemplo vamos a crear una pequeña clase que almacene un objeto, con su getter y setter, y que permita imprimir la cadena que lo representa, o bien imprimir la cadena al revés:

```
public class Imprimidor<T> {
  private T objeto;

  public Imprimidor(T objeto) {
    this.objeto = objeto;
  }

  public void setObjeto(T objeto) {
    this.objeto = objeto;
  }

  public T getObjeto() {
    return objeto;
  }

  public void imprimir() {
    System.out.println(objeto.toString());
  }

  public void imprimir_reves() {
    StringBuffer sb = new StringBuffer(objeto.toString());
    System.out.println(sb.reverse());
  }
}
```

Ahora podríamos utilizar nuestra nueva clase de la misma forma que hacíamos con List:

```
public class Ejemplo {
```

```
    public static void main(String[] args) {
      Imprimidor<String> impStr = new Imprimidor<String>("Hola mundo");
      impStr.imprimir();

      Imprimidor<Integer> impInt = new Imprimidor<Integer>(143);
      impInt.imprimir_reves();
    }
  }
```

En ocasiones también puede interesarnos limitar los tipos con los que se puede parametrizar nuestra clase. Por ejemplo, podríamos querer crear una clase con un método que imprimiera el resultado de dividir un número entre 2. No tendría sentido permitir que se pudiera parametrizar la clase con el tipo String; nos interesaría pues encontrar alguna forma de indicar que el tipo utilizado debe ser un subtipo de la clase Number. Nuestro código podría tener un aspecto similar al siguiente:

```
public class Divisor<T extends Number> {
    private T numero;

    public Divisor(T numero) {
        this.numero = numero;
    }

    public void dividir() {
        System.out.println(numero.doubleValue() / 2);
    }

    static public void main(String args[]) {
        Divisor <Integer> a = new Divisor (10);
        //Divisor <String> a = new Divisor (10);
        a.dividir();
    }
}
```

Ahora, en Java no existe la herencia múltiple, pero lo que si podemos hacer es implementar varias interfaces distintas. Si quisieramos obligar que el tipo implemente varias interfaces distintas, o que extienda una clase e implemente una o varias interfaces, tendríamos que separar estos tipos con el carácter &:

```
    public class Divisor<T extends A & B> {
      ...
    }
```

Otra cosa que podemos hacer es utilizar más de un parámetro de tipo, separando los tipos con comas. Supongamos por ejemplo que quisiéramos crear una clase que imprimiera la suma de dos números. Escribiríamos algo como esto:

```
public class Sumador<T1 extends Number, T2 extends Number> {
    private T1 numero1;
    private T2 numero2;

    public Sumador(T1 numero1, T2 numero2) {
        this.numero1 = numero1;
        this.numero2 = numero2;
    }

    public void sumar() {
       System.out.println(numero1.doubleValue() + numero2.doubleValue());
    }

    static public void main(String args[]) {
        Sumador <Float, Float> a = new Sumador(1,2);
        //Sumador <char, Float> b = new Sumador(1,2);
        a.sumar();
    }
}
```

Por supuesto también podemos crear clases abstractas e interfaces de forma similar a las clases genéricas:

```
    public interface MiColeccion<T> {
      public void anyadir(T objeto);
      public T obtener();
      public void ordenar();
    }
```

E incluso métodos, en cuyo caso se indica el nombre a utilizar para el tipo genérico antes del valor de retorno del método. Sustituyamos como ejemplo la clase que imprimía las cadenas al revés por un método estático:

```
     public class EjemploGenerics {
      public static <T> void imprimir_reves(T objeto) {
        StringBuffer sb = new StringBuffer(objeto.toString());
        System.out.println(sb.reverse());
      }
    }
```

A la hora de llamar al método podemos especificar el tipo de la misma forma que hacíamos con las clases genéricas, aunque en este caso es menos útil:

```
public class EjemploGenerics {
  public static <T> void imprimir_reves(T objeto) {
    StringBuffer sb = new StringBuffer(objeto.toString());
    System.out.println(sb.reverse());
  }

  public static void main(String args[]) {
    String cadena = new String("Hola");

    EjemploGenerics ej = new EjemploGenerics();
    ej.<string>imprimir_reves(cadena);
  }
}</string>
```

También podemos omitirlo, por supuesto:

```
public class EjemploGenerics {
  public static <T> void imprimir_reves(T objeto) {
    StringBuffer sb = new StringBuffer(objeto.toString());
    System.out.println(sb.reverse());
  }

  public static void main(String args[]) {
    String cadena = new String("Hola");
    Integer entero = new Integer(12);
    Float flotante = new Float(13.6);
    Object objeto = new Object();

    imprimir_reves(cadena);
    imprimir_reves(entero);
    imprimir_reves(flotante);
    imprimir_reves(objeto);
  }
}
```

7.4. Uso de java.util.Comparator and java.lang.Comparable así como el paquete java.util para escribir código que ordene y realice búsquedas

El uso de objetos que puedan compararse permite realizar algunas tareas en Java de manera muy eficiente. La clase Comparable de Java permite hacer que un objeto pueda ser comparado con otro con el propósito de utilizar métodos de ordenamiento y búsqueda. Para llevar a cabo el ordenamiento de una lista haremos uso del método estático de la clase Colections denominado sort. Dicho método tiene la siguiente sintaxis

`Collections.sort(List)`

Para el caso de búsqueda podemos realizar una búsqueda iterativa o hacer uso de una búsqueda binaria definida como

`Collections.binarySearch((List)miLista, Objet a)`

En ambos casos es necesario hacer un objeto que implemente la interface Comparable de Java y por lo tanto el método compareTo(Object a).

El siguiente es un ejemplo de como hacer la implementación de estos métodos para buscar una persona ya sea por nombre, apellido o edad.

```
 import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class Uso_Comparadores implements Comparable {

    private String Nombre;
    private String Apellido;
    private int Edad;

    public Uso_Comparadores(String unNombre, String unApellido, int unaEdad) {
        Nombre   = unNombre;
        Apellido = unApellido;
        Edad     = unaEdad;
    }

    public int compareTo(Object arg0) {
        Uso_Comparadores c = (Uso_Comparadores) arg0;
        // ordenar por Apellido
        //return this.Apellido.compareTo(c.Apellido);
```

```
        // ordenar por Nombre
        //return this.Nombre.compareTo(c.Nombre);

        return Integer.toString(this.Edad).compareTo(Integer.toString(c.Edad));
    }

    public String Datos() {
        return Nombre + " " + Apellido + " " + Edad;
    }

    static public void main(String args[]) {
        List <Uso_Comparadores> miLista = new ArrayList();

        miLista.add(new Uso_Comparadores("Juan", "Reyes", 21 ));
        miLista.add(new Uso_Comparadores("Luis", "Alba", 31 ));
        miLista.add(new Uso_Comparadores("Maria Luisa", "Jaimes", 23 ));
        miLista.add(new Uso_Comparadores("Adan", "Zavala", 18 ));

        System.out.println("Datos sin ordenar");
        for(Uso_Comparadores i : miLista)
            System.out.println(i.Datos());

        System.out.println("Datos ordenados");
        Collections.sort(miLista);

        for(Uso_Comparadores i : miLista)
            System.out.println(i.Datos());

        System.out.println("Resultado de la Busqueda");

        Object a = new Uso_Comparadores("algo", "algo", 18);
        int indice = Collections.binarySearch((List)miLista, a);
        System.out.println(miLista.get(indice).Datos());
    }
}
```

7.4.1. Ejemplo de una lista de Empleados

El siguiente es un ejemplo de como hacer el ordenamiento de Empleados en un TreeMap

```
import java.util.Comparator;
import java.util.Set;
import java.util.TreeMap;


public class MapaOrdenado {

    public static void main(String a[]){

        //Comparacion por Nombre (String comparison)

        TreeMap<Empleado,String> tm =
           new TreeMap<Empleado, String>(new ComparaNombre());
        tm.put(new Empleado("Ramom",3000), "RAM");
        tm.put(new Empleado("Juan",6000), "JOHN");
        tm.put(new Empleado("Critina",2000), "CRISH");
        tm.put(new Empleado("Tomas",2400), "TOM");
        Set<Empleado> keys = tm.keySet();
        for(Empleado key:keys){
            System.out.println(key+" ==> "+tm.get(key));
        }
        System.out.println("===================================");

        //Comparacion por salario

        TreeMap<Empleado,String> trmap =
           new TreeMap<Empleado, String>(new ComparaSalario());
        trmap.put(new Empleado("Ramon",3000), "RAM");
        trmap.put(new Empleado("Juan",6000), "JOHN");
        trmap.put(new Empleado("Cristina",2000), "CRISH");
        trmap.put(new Empleado("Tomas",2400), "TOM");
        Set<Empleado> ks = trmap.keySet();
        for(Empleado key:ks){
            System.out.println(key+" ==> "+trmap.get(key));
        }
    }
```

```
}

class ComparaNombre implements Comparator<Empleado>{

    @Override
    public int compare(Empleado e1, Empleado e2) {
        return e1.getNombre().compareTo(e2.getNombre());
    }
}

class ComparaSalario implements Comparator<Empleado>{

    @Override
    public int compare(Empleado e1, Empleado e2) {
        if(e1.getSalario() > e2.getSalario()){
            return 1;
        } else {
            return -1;
        }
    }
}

class Empleado{
    private String nombre;
    private float salario;

    public Empleado(String n, int s){
        this.nombre = n;
        this.salario = s;
    }
    public String getNombre() {
        return nombre;
    }
    public float getSalario() {
        return salario;
    }
    @Override
    public String toString(){
        return "Nombre: "+this.nombre+"-- Salario: "+this.salario;
    }
}
```

7.4.2. Ejemplo de Una agenda

Considere el caso de una agenda personal donde deseamos guardar información correspondiente a personas. Cada una de las personas podrá tener uno o más teléfonos y estos quedarán identificados por el lugar donde lo tienen. Así por ejemplo Juan Pérez tendrá un teléfono de casa, un teléfono del trabajo y un teléfono movil.

Para comenzar creamos la clase Persona

```
package Agenda2;

import java.util.Map;
import java.util.TreeMap;

public class Persona {
    private String Nombre, Paterno, Materno;
    int Edad;
    private Map <String, String> telefonos;

    public Persona(String unNombre, String unPaterno, String unMaterno,
            int unaEdad) {
        Nombre = unNombre;
        Paterno = unPaterno;
        Materno = unMaterno;
        Edad = unaEdad;

        telefonos = new TreeMap <String, String>();
    }

    public void agregaTelefono(String unaDescripcion, String unTelefono){
        telefonos.put(unaDescripcion, unTelefono);
    }

    public String getNombre() {
        return Nombre;
    }
    public String getPaterno() {
        return Paterno;
    }
```

```
    public String getMaterno() {
        return Materno;
    }
    public int getEdad() {
        return Edad;
    }

    void imprime() {
        System.out.println(Nombre + " " + Paterno + " " + Materno +
                " " + Edad);

        if(telefonos.keySet().isEmpty()) return;

        for(String a: telefonos.keySet()){
            System.out.println("\t" + a + " : " + telefonos.get(a) );
        }
    }

    @Override
    public String toString() {
        return Nombre + " " + Paterno + " " + Materno + " " +Edad;
    }
}
```

Clase Compara

Esta clase es necesario implementar ya que es la responsable de decidir sobre el objeto persona como se ordenaran los datos. Esta implementación queda como:

```
package Agenda2;

import java.util.Comparator;

class Compara implements Comparator <Persona>{

    public Compara() {
    }

    @Override
    public int compare(Persona t, Persona t1) {
```

```
        // Ordena por apellido Paterno

        //return t.getPaterno().compareTo(t1.getPaterno());

        // Ordena por Edad

        return (t.getEdad()-t1.getEdad());

    }
}
```

Implementación con Listas

La implementación con Listas y utilizando Collections para ordenar queda

```
package Agenda2;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class MiAgenda_Listas {
    static public void main(String args[]) {
        List <Persona> ls = new ArrayList <Persona>();

        Persona p = new Persona("Jorge", "Pintor", "Leon", 19);
        p.agregaTelefono("Movil", "1231231");
        p.agregaTelefono("Casa", "912931239");
        ls.add(p);

        p = new Persona("Roberto", "Garcia", "Palomares", 25);
        ls.add(p);

        p = new Persona("Felipe", "Rodriguez", "Gonzalez", 24);
        p.agregaTelefono("Tralpu", "'2340923'4");
        p.agregaTelefono("Morelia", "1231");
        p.agregaTelefono("Nextel", "3423");
        p.agregaTelefono("Movil", "234");
        ls.add(p);

        for(Persona i: ls)
```

```
            System.out.println(i.toString());

        Collections.sort(ls, new Compara());

        System.out.println("Despues de Ordenar");

        for(Persona i: ls) {
            i.imprime();
        }
    }
}
```

Implementación con TreeMaps

La implementación con TreeMaps queda como

```
package Agenda2;

import java.util.Set;
import java.util.TreeMap;

public class MiAgenda_TreeMap {

    static public void main(String args[]) {
        TreeMap<Persona, String> tm = new TreeMap <Persona, String> (new Compara());
        Persona p = new Persona("Jorge", "Pintor", "Leon", 19);
        p.agregaTelefono("Movil", "1231231");
        p.agregaTelefono("Casa", "912931239");

        tm.put(p, "asd");

        p = new Persona("Roberto", "Garcia", "Palomares", 25);
        tm.put(p, "lkjdf");

        p = new Persona("Felipe", "Rodriguez", "Gonzalez", 24);
        p.agregaTelefono("Tralpu", "'2340923'4");
        p.agregaTelefono("Morelia", "1231");
        p.agregaTelefono("Nextel", "3423");
        p.agregaTelefono("Movil", "234");
        tm.put(p, "ñsldf");
```

```
        Set <Persona> ks = tm.keySet();
        for(Persona k:ks)
            k.imprime();
    }
}
```

Concurrencia

8.1. Creación de hilos a partir de java.lang.Thread y java.lang.Runnable

Los objetos proporcionan una forma de dividir un programa en secciones independientes. A menudo, también es necesario convertir un programa en subtareas separadas que se ejecuten independientemente.

Cada una de estas subtareas independientes recibe el nombre de hilo, y uno programa como si cada hilo se ejecutara por si mismo y tuviera la UCP para solo. Algún mecanismo subyacente divide de hecho el tiempo de UCP entre ellos, pero generalmente el programador no tiene por qué pensar en ello, lo que hace de la programación de hilos múltiples una tarea mucho mas sencilla.

La forma más simple de crear un hilo es heredad de la clase Thread que tiene todo lo necesario para crear y ejecutar hilos. El método más importante de Thread es run(), que debe ser sobrescrito para hacer que el hilo haga lo que se le manda. Por consiguiente, run() es el código que se ejecutará simultáneamente con los otros hilos del programa. Otro método importante de los hilos e start(), este método permite arrancar un proceso en el instante que uno lo considere pertinente. Una vez que se da la orden start(), el método run() tomará la ejecución del hilo.

El siguiente es un ejemplo de 5 hilos que realizar una cuenta decreciente

```
public class HiloSimple extends Thread {
  private int cuentaAtras = 5;
  private static int conteoHilos = 0;
  private int numeroHilos = ++conteoHilos;

  public HiloSimple() {
    super(Integer.toString(conteoHilos));
    System.out.println("Creando Hilo " + (numeroHilos-1));
```

```
  }
    @Override
  public String toString() {
    return "Hilo # " + getName() + " (" + cuentaAtras + "), " ;
  }

    @Override
  public void run() {
    while(true) {
      System.out.println(this);
      if(--cuentaAtras == 0)
        return;
    }
  }

  public static void main(String[] args) {
    for(int i = 0; i < 5; i++)
      new HiloSimple().start();
    System.out.println("Todos los hilos Arrancados");
  }
}
```

8.2. Estados de un hilo

Durante el ciclo de vida de un thread, éste se puede encontrar en diferentes estados. Los estados en los que se puede encontrar un hilo son:

8.2.1. Nuevo Thread

La siguiente sentencia crea un nuevo thread pero no lo arranca, lo deja en el estado de Nuevo Thread:

```
Thread MiThread = new MiClaseThread();
```

Cuando un thread está en este estado, es simplemente un objeto Thread vacío. El sistema no ha destinado ningún recurso para él. Desde este estado solamente puede arrancarse llamando al método start(), o detenerse definitivamente, llamando al método stop(); la llamada a cualquier otro método carece de sentido y lo único que provocará será la generación de una excepción de tipo IllegalThreadStateException.

8.2.2. Ejecutable

Ahora veamos las dos línea de código que se presentan a continuación:

```
Thread MiThread = new MiClaseThread();
MiThread.start();
```

La llamada al método start() creará los recursos del sistema necesarios para que el thread puede ejecutarse, lo incorpora a la lista de procesos disponibles para ejecución del sistema y llama al método run() del thread. En este momento nos encontramos en el estado "Ejecutable"del diagrama. Y este estado es Ejecutable y no En Ejecución, porque cuando el thread está aquí no esta corriendo. Muchos ordenadores tienen solamente un procesador lo que hace imposible que todos los threads estén corriendo al mismo tiempo. Java implementa un tipo de scheduling o lista de procesos, que permite que él procesador sea compartido entre todos los procesos o threads que se encuentran en la lista. Sin embargo, para nuestros propósitos, y en la mayoría de los casos, se puede considerar que este estado es realmente un estado "En Ejecución", porque la impresión que produce ante nosotros es que todos los procesos se ejecutan al mismo tiempo.

Cuando el thread se encuentra en este estado, todas las instrucciones de código que se encuentren dentro del bloque declarado para el método run(), se ejecutarán secuencialmente.

8.2.3. Parado

El thread entra en estado "Paradoçuando alguien llama al método suspend(), cuando se llama al método sleep(), cuando el thread está bloqueado en un proceso de entrada/salida o cuando el thread utiliza su método wait() para esperar a que se cumpla una determinada condición. Cuando ocurra cualquiera de las cuatro cosas anteriores, el thread estará Parado.

Por ejemplo, en el trozo de código siguiente:

```
Thread MiThread = new MiClaseThread();
MiThread.start();
try {
     MiThread.sleep( 10000 );
} catch( InterruptedException e ) {
     ;
    }
```

la línea de código que llama al método sleep():

```
MiThread.sleep( 10000 );
```

hace que el thread se duerma durante 10 segundos. Durante ese tiempo, incluso aunque él procesador estuviese totalmente libre, MiThread no correría. Después de esos 10 segundos. MiThread volvería a estar en estado "Ejecutablez ahora sí que el procesador podría hacerle caso cuando se encuentre disponible.

Para cada una de los cuatro modos de entrada en estado Parado, hay una forma específica de volver a estado Ejecutable. Cada forma de recuperar ese estado es exclusiva; por ejemplo, si el thread ha sido puesto a dormir, una vez transcurridos los milisegundos que se especifiquen, él solo se despierta y vuelve a estar en estado Ejecutable. Llamar al método resume() mientras esté el thread durmiendo no serviría para nada.

Los métodos de recuperación del estado Ejecutable, en función de la forma de llegar al estado Parado del thread, son los siguientes:

Si un thread está dormido, pasado el lapso de tiempo Si un thread está suspendido, luego de una llamada al método resume() Si un thread está bloqueado en una entrada/salida, una vez que el comando E/S concluya su ejecución Si un thread está esperando por una condición, cada vez que la variable que controla esa condición varíe debe llamarse a notify() o notifyAll()

8.2.4. Muerto

Un thread se puede morir de dos formas: por causas naturales o porque lo maten (con stop()). Un thread muere normalmente cuando concluye de forma habitual su método run(). Por ejemplo, en el siguiente trozo de código, el bucle while es un bucle finito -realiza la iteración 20 veces y termina-:

```
public void run() {
    int i=0;
    while( i < 20 )
        {
        i++;
        System.out.println( "i = "+i );
        }
    }
```

Un thread que contenga a este método run(), morirá naturalmente después de que se complete el bucle y run() concluya.

También se puede matar en cualquier momento un thread, invocando a su método stop(). En el trozo de código siguiente:

```
Thread MiThread = new MiClaseThread();
MiThread.start();
try {
    MiThread.sleep( 10000 );
} catch( InterruptedException e ) {
    ;
    }
MiThread.stop();
```

se crea y arranca el thread MiThread, lo dormimos durante 10 segundos y en el momento de despertarse, la llamada a su método stop(), lo mata.

El método stop() envía un objeto ThreadDeath al thread que quiere detener. Así, cuando un thread es parado de este modo, muere asíncronamente. El thread morirá en el momento en que reciba la excepción ThreadDeath.

Los applets utilizarán el método stop() para matar a todos sus threads cuando el navegador con soporte Java en el que se están ejecutando le indica al applet que se detengan, por ejemplo, cuando se minimiza la ventana del navegador o cuando se cambia de página.

8.2.5. El método isAlive()

La interface de programación de la clase Thread incluye el método isAlive(), que devuelve true si el thread ha sido arrancado (con start()) y no ha sido detenido (con stop()). Por ello, si el método isAlive() devuelve false, sabemos que estamos ante un "Nuevo Thread.º ante un thread "Muerto". Si nos devuelve true, sabemos que el thread se encuentra en estado .Ejecutable.º "Parado". No se puede diferenciar entre "Nuevo Threadz "Muerto", ni entre un thread "Ejecutable.º "Parado".

8.2.6. Ejemplo Contadores

Un ejemplo más interesante es poner dos contadores, uno que se incremente de 1 en 1 y otro que se incremente de dos en dos, mientras podemos tomar información de un campo de texto y hacer algo con ella. Este ejemplo tiene dos contadores, uno que va de 1 en uno al que llamaremos contador 1 y otro que va de dos en dos que llamaremos contador 2.

```
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.logging.Level;
import java.util.logging.Logger;
```

```
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JTextField;

public class Ejemplo_ver_hilo extends JFrame{
    private JTextField campoTexto;
    private JTextField campoTexto1;
    private JTextField campoTexto2;
    private JButton miBoton;
    private Contador1 contador1;
    private Contador2 contador2;

    public Ejemplo_ver_hilo() {
        super("Hilos");
        this.definirVentana();

        this.setSize(400, 100);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setVisible(true);
        contador1.start();
        contador2.start();
    }

    public void definirVentana() {
        campoTexto = new JTextField(10);
        campoTexto1 = new JTextField(3);
        campoTexto2 = new JTextField(3);
        contador1 = new Contador1();
        contador2 = new Contador2();
        miBoton = new JButton("Ok");

        miBoton.addActionListener(new accionBotonHilo());

        this.setLayout(new FlowLayout());

        add(campoTexto);
        add(campoTexto1);
        add(campoTexto2);
        add(miBoton);
    }
```

```
private class Contador1 extends Thread {
    private int cuenta = 0;

    @Override
    public void run(){
        while(true) {
            campoTexto1.setText(Integer.toString(cuenta++));
            try {
                sleep(1000);
            } catch (InterruptedException ex) {
                Logger.getLogger(Ejemplo_ver_hilo.class.getName()).log(Level.SEVERE, null,
            }
        }
    }
}

private class Contador2 extends Thread {
    private int cuenta = 0;

    @Override
    public void run(){
        while(true) {
            campoTexto2.setText(Integer.toString(cuenta+=2));
            try {
                sleep(1000);
            } catch (InterruptedException ex) {
                Logger.getLogger(Ejemplo_ver_hilo.class.getName()).log(Level.SEVERE, null,
            }
        }
    }
}

private class accionBotonHilo implements ActionListener {
    public void actionPerformed(ActionEvent arg0) {
        javax.swing.JOptionPane.showMessageDialog(null, campoTexto.getText());
    }
}

static public void main(String args[]) {
    new Ejemplo_ver_hilo();
}
```

```
}
```

8.2.7. Ejemplo Cánicas

El siguiente es un ejemplo de un conjunto de círculos negros que permanecen en movimiento mientras se ejecutan procesos de agregar eliminar círculos. El programa es llamado canicas

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.ArrayList;
import java.util.List;
import java.util.Random;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;

public class Canicas extends JFrame {
    private AreaDibujo miAreaDibujo;
    private JButton miBoton1;
    private JButton miBoton2;
    List <Canica> misCanicas;
    Random r;
    MueveCanicas salto;

    public Canicas(){
        JPanel AreaControl = new JPanel(new GridLayout(2, 1));
        miBoton1 = new JButton("Agregar");
        miBoton2 = new JButton("Quitar");

        miBoton1.addActionListener(new botonAgrega());
        miBoton2.addActionListener(new botonQuitar());

        r = new Random();
```

```
        AreaControl.add(miBoton1);
        AreaControl.add(miBoton2);

        miAreaDibujo = new AreaDibujo();
        misCanicas = new ArrayList();

        this.add(AreaControl, BorderLayout.WEST);
        this.add(miAreaDibujo,BorderLayout.EAST);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        pack();
        setVisible(true);
        salto = new MueveCanicas();
        salto.start();
    }

    private class AreaDibujo extends JPanel{
        public AreaDibujo(){
            this.setBackground(Color.WHITE);
            this.setPreferredSize(new Dimension(600, 600));
        }

        @Override
        public void paint(Graphics g){
            super.paint(g);

            if(!misCanicas.isEmpty()) {
                for(Canica a : misCanicas) {
                    a.dibuja(g);
                }
            }
        }
    }

    private class Canica {
        int x0, y0;

        public Canica(int unaX, int unaY) {
            x0 = unaX;
            y0 = unaY;
        }
```

```
     public void mueve(int dx, int dy) {
         int aux;

         aux = x0+dx;
         if(aux >=0 && aux <=600)
             this.x0 = aux;

         aux = y0+dy;
         if(aux >=0 && aux <=600)
             this.y0 = aux;
    }

     public void dibuja(Graphics g) {
        g.setColor(Color.BLACK);
        g.fillOval(x0, y0, 10, 10);
        g.drawString("C", x0, y0);
    }
}

private class botonAgrega implements ActionListener{

    public void actionPerformed(ActionEvent arg0) {
        int x = Math.abs(r.nextInt())%600;
        int y = Math.abs(r.nextInt())%600;
        misCanicas.add(new Canica(x, y));
        repaint();
    }
}

private class botonQuitar implements ActionListener{

    public void actionPerformed(ActionEvent arg0) {
        misCanicas.remove(0);
        repaint();
    }
}
private class MueveCanicas extends Thread {

    @Override
    public void run(){
        while(true) {
```

```
                for(Canica a : misCanicas)
                     a.mueve(r.nextInt()%2, r.nextInt()%2);
                repaint();
                 try {
                     sleep(10);
                 } catch (InterruptedException ex) {
                     Logger.getLogger(
                      EjemploHilo.class.getName()).log(Level.SEVERE, null, ex);
                 }
            }
        }
    }

    static public void main(String args[]) {
        new Canicas();
    }
}
```

8.3. Solución de problemas de acceso concurrente

Consideremos el siguiente ejemplo, donde se crea un hilo que incrementa un acumulador de enteros al que llamaremos total

```
class ThreadB extends Thread{
    int total=0;
    @Override
    public void run(){
            for(int i=0; i<100 ; i++){
                total += 1;
            }
    }
}
```

El programa principal que manda llamar a este hilo es:

```
public class ThreadA {
    public static void main(String[] args){
        ThreadB b = new ThreadB();
        b.start();
        System.out.println("Total is: " + b.total);
    }
```

```
}
```

Cuando se ejecuta la salida es:

```
El Total es: 0
```

Note en este caso que le valor de la variable total, una vez realizada las operaciones, debía ser 100. ¿Que ocasiono este problema?. La respuesta es que el hilo B no esta sincronizado. Para llevar a cabo el sincronización tenemos que hacer uso de instrucciones adicionales como synchronized, notify and wait. A continuación se presenta el ejemplo de uso

```
public class ThreadA {
    public static void main(String[] args){
        ThreadB b = new ThreadB();
        b.start();

        synchronized(b){
            try{
                System.out.println("Esperando al hilo b a que termine...");
                b.wait();
            }catch(InterruptedException e){
                e.printStackTrace();
            }

            System.out.println("Total is: " + b.total);
        }
    }
}

class ThreadB extends Thread{
    int total=0;
    @Override
    public void run(){
        synchronized(this){
            for(int i=0; i<100 ; i++){
                total += 1;
            }
            notify();
        }
    }
}
```

En este caso la salida de la ejecución es:

```
Esperando al hilo b a que termine...
Total is: 100
```

8.4. Uso de wait, notify, or notifyAll

El problema de Productor Consumidor es el ejemplo clásico de uso de hilos. Para este problema tenemos dos hilos cuya función es producir un bien en el primer caso y consumir ese bien una vez que este se produce. Es importante que esta acción este sincronizada ya que no se puede consumir lo que no sé a producido y el consumidor agota las existencias almacenada. Adicionalmente el productor no producto hasta que no se haya agotado el anterior.

Una primer implementación pude ser:

```
public class ProductorConsumidor_ant {

    public ProductorConsumidor_ant() {
        Almacen almacen = new Almacen();
        Productor p = new Productor(almacen, 1);
        Consumidor c = new Consumidor(almacen, 1);
        p.start();
        c.start();
    }

    class Almacen {
        private int almacenados;
        private boolean disponible;

        public Almacen() {
            disponible = false;
            almacenados = 0;
        }

        public int get() {
            disponible = false;
            return almacenados;
        }

        public void put(int unValor) {
            almacenados = unValor;
```

```
            disponible = true;
        }
    }

    class Productor extends Thread {
        private Almacen almacen;
        private int numero;

        public Productor(Almacen unDeposito, int unNumero) {
            almacen = unDeposito;
            this.numero = unNumero;
        }

        @Override
        public void run() {
            for (int i = 0; i < 10; i++) {
                almacen.put(i);
                System.out.println("Productor #" + this.numero
                + " pone: " + i);
                try {
                sleep((int)(Math.random() * 100));
                } catch (InterruptedException e) { }
            }
        }
    }

     class Consumidor extends Thread {
       private Almacen almacen;
       private int numero;

       public Consumidor(Almacen unDeposito, int unNnumero) {
          almacen = unDeposito;
          this.numero = unNnumero;
       }

       @Override
       public void run() {
          int value = 0;
             for (int i = 0; i < 10; i++) {
                value = almacen.get();
                System.out.println("Consumidor #"
```

```
                            + this.numero
                            + " toma: " + value);
            }
        }
    }

    static public void main(String args[]) {
        new ProductorConsumidor_ant();
    }
}
```

Cuando este se ejecuta tenemos

```
Productor #1 pone: 0
Consumidor #1 toma: 0
Consumidor #1 toma: 0
Consumidor #1 toma: 0
Consumidor #1 toma: 0
Consumidor #1 toma: 0
Consumidor #1 toma: 0
Consumidor #1 toma: 0
Consumidor #1 toma: 0
Consumidor #1 toma: 0
Consumidor #1 toma: 0
Productor #1 pone: 1
Productor #1 pone: 2
Productor #1 pone: 3
Productor #1 pone: 4
Productor #1 pone: 5
Productor #1 pone: 6
Productor #1 pone: 7
Productor #1 pone: 8
Productor #1 pone: 9
```

Note que el productor 1 pone un producto el producto 0 en el alancen y el consumidor consume 10 veces ese producto, lo cual no esta de acuerdo con el planteamiento de problema. El consumidor 1 esta consumiendo lo que aún no se ha producido.

En este caso tenemos que hacer uso de un mecanismo de sincronización y de comandos que permitan que que el consumidor espere a que se produzca un articulo y que el productor espere a que se consuma para generar uno nuevo. Para llevar a cabo esto solamente es necesario modificar la clase Almacen para hacer la sincronización del productor y el consumidor.

```
class Almacen {
   private int contenidos;
   private boolean disponible = false;

   public synchronized int get() {
      while (disponible == false) {
         try {
            wait();
         }
         catch (InterruptedException e) {
         }
      }
      disponible = false;
      notifyAll();
      return contenidos;
   }
   public synchronized void put(int value) {
      while (disponible == true) {
         try {
            wait();
         }
         catch (InterruptedException e) {
         }
      }
      contenidos = value;
      disponible = true;
      notifyAll();
   }
}
```

Java I/O

Una forma sencilla de hacer la lectura de archivos es utilizando la clase File y la clase RandomAccessFile. A continuación se muestran a detalle el uso de estas dos clases.

9.1. La clase File

Antes de proceder al estudio de las clases que describen la entrada/salida vamos a estudiar la clase File, que nos proporciona información acerca de los archivos, de sus atributos, de los directorios, etc. También explicaremos cómo se crea un filtro mediante el interface FilenameFilter para obtener la lista de los archivos que tengan por ejemplo, la extensión .java.

La clase File tiene tres constructores

```
File(String path)
File(String path, String name)
File(File dir, String name)
```

El parámetro path indica el camino hacia el directorio donde se encuentra el archivo, y name indica el nombre del archivo. Los métodos más importantes que describe esta clase son los siguientes:

```
String getName()
String getPath()
String getAbsolutePath()
boolean exists()
boolean canWrite()
boolean canRead
boolean isFile()
boolean isDirectory()
boolean isAbsolute()
long lastModified()
```

```
long length()
boolean mkdir()
boolean mkdirs()
boolean renameTo(File dest);
boolean delete()
String[] list()
String[] list(FilenameFilter filter)
```

Mediante un ejemplo explicaremos algunos de los métodos de la clase File.

Creamos un objeto fichero de la clase File, pasándole el nombre del archivo, en este caso, el nombre del archivo código fuente ArchivoApp1.java.

```
File fichero=new File("ArchivoApp1.java");
```

Si este archivo existe, es decir, si la función exists devuelve true, entonces se obtiene información acerca del archivo:

```
getName devuelve el nombre del archivo
getPath devuelve el camino relativo
getAbsolutePath devuelve el camino absoluto.
canRead nos indice si el archivo se puede leer.
canWrite nos indica si el archivo se puede escribir
length nos devuelve el tamaño del archivo, si dividimos la cantidad devuelta entre 1

if(fichero.exists()){
    System.out.println("Nombre del archivo "+fichero.getName());
    System.out.println("Camino             "+fichero.getPath());
    System.out.println("Camino absoluto    "+fichero.getAbsolutePath());
    System.out.println("Se puede escribir  "+fichero.canRead());
    System.out.println("Se puede leer      "+fichero.canWrite());
    System.out.println("Tamaño             "+fichero.length());
}
```

```
import java.io.File;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.RandomAccessFile;
import java.util.StringTokenizer;
```

9.2. La clase RandomAccessFile

La clase Java RandomAccessFile se utiliza para acceder a un fichero de forma aleatoria. Los constructores de la clase son:

RandomAccessFile(String path, String modo); RandomAccessFile(File objetoFile, String modo);

Lanzan una excepción FileNotFoundException. El argumento modo indica el modo de acceso en el que se abre el fichero.

Los valores permitidos para este parámetro son: modo "r": Abre el fichero en modo solo lectura.El fichero debe existir. Una operación de escritura en este fichero lanzará una excepción IOException.

modo "rw": Abre el fichero en modo lectura y escritura. Si el fichero no existe se crea.

Ejemplo: abrir un fichero aleatorio para lectura

Se abre el fichero clientes.dat para lectura usando el primer constructor.

```
RandomAccessFile fichero = new RandomAccessFile("/ficheros/clientes.dat", "r");
```

Ejemplo : abrir un fichero aleatorio para lectura/escritura Se abre el fichero personas.dat para lectura/escritura usando el segundo constructor. Si el fichero no existe se crea.

```
File f = new File ("/ficheros/personas.dat");
RandomAccessFile fichero = new RandomAccessFile(f, "rw");
```

ACCESO A LOS DATOS EN FICHEROS ALEATORIOS Para acceder de forma aleatoria a los datos contenidos en el fichero, la clase RandomAccessFile dispone de varios métodos. Entre ellos:

long getFilePointer(); Devuelve la posición actual del puntero del fichero. Indica la posición (en bytes) donde se va a leer o escribir.

long length(); Devuelve la longitud del fichero en bytes.

void seek(long pos); Coloca el puntero del fichero en una posición pos determinada. La posición se da como un desplazamiento en bytes desde el comienzo del fichero. La posición 0 indica el principio del fichero. La posición length() indica el final del fichero.

Además dispone de métodos de lectura/escritura:

public int read(): Devuelve el byte leído en la posición marcada por el puntero. Devuelve -1 si alcanza el final del fichero. Se debe utilizar este método para leer los caracteres de un fichero de texto.

public final String readLine(): Devuelve la cadena de caracteres que se lee, desde la posición marcada por el puntero, hasta el siguiente salto de línea que se encuentre.

public xxx readXxx(): Hay un método read para cada tipo de dato básico: readChar, readInt, readDouble, readBoolean, etc.

public void write(int b) : Escribe en el fichero el byte indicado por parámetro. Se debe utilizar este método para escribir caracteres en un fichero de texto.

public final void writeBytes(String s) : Escribe en el fichero la cadena de caracteres indicada por parámetro.

public final void writeXxx(argumento): También existe un método write para cada tipo de dato básico: writeChar, writeInt, writeDouble, writeBoolean, etc.

9.2.1. Ejemplos de operaciones con ficheros de acceso aleatorio

```
public class Archivo {
    static String ruta = "/Users/felix/fie_mac/public_html/programacion_java/codigos/Cap
    static public void ejemplo1(){
        File arch = new File(ruta+"un_archivo.txt");

        if(arch.exists())
            System.out.println("Si existe el archivo");


        else
            System.out.println("No existe el arvhivo");

    }

    static public void ejemplo2() throws FileNotFoundException{
        RandomAccessFile arch = new RandomAccessFile(ruta+"un_archivo.txt", "r");
        String linea;
        int lineas = 0;
        try {
            while((linea = arch.readLine())!=null) {
                System.out.println(linea);
                lineas ++;
            }
            arch.close();
        } catch (IOException ex) {
```

```
            System.out.println("No existe el archivo");
        }
        System.out.println("El archivo tiene " + lineas +" lineas");
    }

    static public void ejemplo3() throws FileNotFoundException, IOException{
        RandomAccessFile arch = new RandomAccessFile(ruta+"salida.txt", "rw");
        int i, f1=0, f2 = 1, f3;

        for(i=0; i<10; i++) {
            f3 = f2 + f1;
            System.out.println(f3);
            arch.writeBytes(Integer.toString(f3)+", ");
            f1 = f2;
            f2 = f3;
        }
        arch.close();
    }

    static public void ejemplo4() throws FileNotFoundException, IOException{
        RandomAccessFile arch = new RandomAccessFile(ruta+"que_es.txt", "rw");
        int i,  N = 100;
        double theta = 2.0*Math.PI/(double) N;
        double x, y;

        for(i=0; i<100; i++) {
            x = 10.0*Math.cos(theta*i);
            y = 10.0*Math.sin(theta*i);
            arch.writeBytes(x + ", " + y + "\n");
        }
        arch.close();
    }

    static public void ejemplo5() throws FileNotFoundException, IOException{
        RandomAccessFile arch = new RandomAccessFile(ruta+"que_es.txt", "rw");
        String Linea;
        StringTokenizer ficha;
        double d, r;
        int i=0;

        while((Linea=arch.readLine())!= null){
```

```
            i++;
            ficha = new StringTokenizer(Linea, ", ");
            r = 0;
            while(ficha.hasMoreTokens()) {
                d = Double.valueOf(ficha.nextToken()).doubleValue();
                r += d*d;
            }
            System.out.println(i + ".- " + Math.sqrt(r));
        }

        arch.close();

    }

    public static void main(String args[]) throws FileNotFoundException, IOException {
        ejemplo5();
    }
}
```

9.3. Uso de BufferedReader,BufferedWriter, File, FileReader, FileWriter and PrintWriter

9.4. Serialización de objetos usando las siguientes clases: DataOutputStream, FileInputStream, FileOutputStream, ObjectInputStream, ObjectOutputStream and Serializable

9.5. Uso de la clase java.util.Locale

9.6. Uso de expresiones regulares

Tareas

10.1. Tarea 1

Escribir Código Java que permita haber el redondeo de un número flotante a dos cifras significativa. Utilizar solamente operaciones y cambios de tipo para realizar la tarea.

10.2. Tarea 2

Hacer la implementación del objeto Circulo, el cual tiene como datos el centro de coordenadas y su radio. Algunos métodos relacionados con el son Área, perímetro.

El Diagrama UML luce como

```
-----------------------------------------
Circulo
-----------------------------------------
- Coordenada_x : double
- Coordenada_y : double
- Radio : double
-----------------------------------------
<<constructor>> Circulo (i : double, j : double, r : double)
+ Area() : double
+ Perimertro() : double
-----------------------------------------
```

10.3. Tarea 3

Escribir un método para convertir números enteros a su equivalente en romano utilizando las sentencia swotch

10.4. Tarea 4

Escribir un método o función recursiva para calcular x^n donde x es un número real elevado a un exponente entero n

10.5. Tarea 5

Escribir para el ejemplo de la Agenda un método que permita encontrar el domicilio y teléfono de una persona por su nombre.

Escribir las tarjetas CRC correspondientes a este ejemplo y el diagrama UML.

10.6. Tarea 6

Dado el código para imprimir una factura, hacer una clase que se llame fecha. La fecha que se genera mediante este objeto se debe insertar a la factura para tener la información del día en el que se imprime una factura cualquier. La fecha debe ser la fecha del sistema.

Probar que la clase Cliente y Cuenta funcionan haciendo una clase de prueba que tenga una función main donde se defina una variable de tipo Cliente y se pueda ver la información en cada una de las cuentas del cliente creado.

10.7. Tarea 7

Escribir el código en Java para emular un convertidos de tipos de divisas. El convertidor debe tener al menos posibilidad de hacer el cambio de 6 divisas de diferentes tipos y dos campos te texto donde se pueda dar el valor a cambiar en pesos y nos reporte el equivalente en la nueva divisa.

10.8. Tarea 8

Escribir las clases circulo y elipse para agregar al programa de dibujo que se inicio en clases.

10.9. Tarea