

	INGENIERÍA EN INFORMÁTICA – PLAN 2003/2014 PROGRAMACIÓN DISTRIBUIDA Y COMPONENTES – 9º CUATRIMESTRE	
	APUNTE DE JAVA	VERSIÓN: 1.1 VIGENCIA: 03-05-2014

FUNDAMENTOS DE JAVA

Origen del lenguaje

Originalmente, Java no fue creado para Internet. La primera versión de Java empezó en 1991 y fue escrita en 18 meses en Sun Microsystems. De hecho, en ese momento, ni siquiera se llamó Java: se llamó Oak y se utilizó en Sun para uso interno.

La idea original para Oak era crear un lenguaje orientado a objetos independiente de la plataforma. Por entonces, muchos programadores se limitaban a la programación del IBM PC, pero el entorno corporativo podía incluir toda clase de plataformas de programación, desde la PC hasta los grandes sistemas. Lo que había detrás de Oak era crear algo que se pudiera usar en todas las computadoras. El lanzamiento original de Oak no fue especialmente fascinante; Sun quería crear un lenguaje que se pudiera usar en electrónica.


Oak pasó a llamarse Java en 1995, cuando se lanzó para el uso público y supuso un éxito casi inmediato. En ese momento, Java había adoptado un modelo que lo hizo perfecto para Internet, el modelo *bytecode*.

Los programas Java no necesitan ser autosuficientes, y no tienen que incluir todo el código de máquina que se ejecuta en la computadora. Estos son compilados creando *bytecodes* compactos y son estos *bytecodes* lo que la máquina virtual de Java (JVM), aplicación que ejecuta un programa Java, lee e interpreta para ejecutar el programa. Cuando se descarga un applet, aplicación Java que corre en navegadores Web, lo que realmente se descarga es un archivo *bytecode*.

De esta forma, un programa Java puede ser muy pequeño, ya que todo el código máquina necesario para ejecutarlo está ya en la computadora de destino y no tiene que descargarse. Para distribuir Java entre una gran variedad de ordenadores, Sun solo tuvo que reescribir JVM para que funcionara en esos ordenadores.

Características Principales

1. **Simple:** Java es un lenguaje potente, que ofrece todas las funcionalidades de otros lenguajes como son C y C++, pero de una manera más simple. Se diseñó para ser muy similar a C++, para que su aprendizaje sea más fácil y rápido.
Java añade características muy útiles como el garbage collector (reciclador de memoria dinámica). No es necesario preocuparse de liberar memoria, el reciclador se encarga de ello y como es un thread de baja prioridad, cuando entra en acción, permite liberar bloques de memoria muy grandes, lo que reduce la fragmentación de la memoria.
2. **Orientado a objetos:** Java implementa sus datos como objetos y permite crear interfaces a esos objetos. Soporta las tres características bases del paradigma orientado a objetos, *encapsulación*, *herencia* y *polimorfismo*.
3. **Distribuido:** Java proporciona librerías y herramientas para que los programas puedan ser distribuidos. Se ha construido con extensas capacidades de interconexión TCP/IP.
4. **Robusto:** Java realiza verificaciones en busca de problemas tanto en tiempo de compilación como en tiempo de ejecución. Java proporciona, comprobación de punteros, comprobación de límites de arrays, excepciones y verificación de *bytecodes*.
5. **Arquitectura neutral:** El compilador de Java compila su código a un archivo objeto de formato independiente de la arquitectura de la máquina en que se ejecutará. Cualquier máquina que tenga el sistema de ejecución (runtime) puede ejecutar ese código objeto, sin importar en qué máquina fue generado. La única dependencia del sistema es la máquina virtual (JVM) y las librerías fundamentales.
6. **Seguro:** El código Java pasa muchos test antes de ejecutarse en una máquina. El código se pasa a través de un verificador de *bytecodes* que comprueba el formato de los

	INGENIERÍA EN INFORMÁTICA – PLAN 2003/2014 PROGRAMACIÓN DISTRIBUIDA Y COMPONENTES – 9º CUATRIMESTRE	
	APUNTE DE JAVA	VERSIÓN: 1.1 VIGENCIA: 03-05-2014

fragmentos de código y aplica un probador de teoremas para detectar fragmentos de código ilegal (código que falsea punteros, viola derechos de acceso sobre objetos o intenta cambiar el tipo o clase de un objeto).

7. **Portable:** Característica principal del lenguaje, que está fuertemente arraigada a generar una arquitectura independiente.
8. **Interpretado:** El intérprete de Java (sistema runtime) puede ejecutar directamente el código objeto. Java es más lento que otros lenguajes de programación, como C++, ya que debe ser interpretado y no ejecutado como sucede en cualquier programa tradicional. Java para conseguir ser un lenguaje independiente del sistema operativo y del procesador que incorpore la máquina utilizada, es tanto interpretado y compilado.
9. **Multihilos (multithreaded):** Java permite muchas actividades simultáneas en un programa. Los hilos (threads), también llamados procesos ligeros, son pequeños procesos o piezas independientes de un gran proceso. Al estar los threads construidos en el lenguaje, son más fáciles de usar y más robustos que sus homólogos en C o C++. El beneficio principal consiste en un mejor rendimiento interactivo y mejor comportamiento en tiempo real.
10. **Dinámico:** Java se beneficia del paradigma orientado a objetos, no intenta conectar todos los módulos que comprende una aplicación hasta el tiempo de ejecución.

Programas Java

Los programas Java son de dos tipos principales: aplicaciones y *applets*.


Los *applets* son programas Java que pueden descargarse y ejecutarse como parte de una página Web, fueron muy populares hace años y hoy ya no son prácticamente usados. Estos pueden ser ejecutados también con la herramienta appletviewer, que se encuentra en el directorio bin de Java.

A su vez, Java soporta aplicaciones que están diseñadas para ejecutarse localmente, aunque también se pueden desarrollar aplicaciones Web, estas últimas no forman parte de este curso.

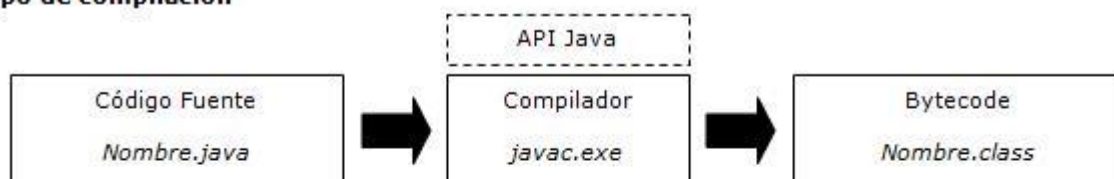
Distribuciones (APIs)

1. **Java ME (Java Platform, Micro Edition) o J2ME:** Orientada a entornos de limitados recursos, como teléfonos móviles, PDAs (Personal Digital Assistant), etc.
2. **Java SE (Java Platform, Standard Edition) o J2SE:** Para entornos de gama media, aquí se sitúa al usuario medio en una PC de escritorio.
3. **Java EE (Java Platform, Enterprise Edition) o J2EE:** Orientada a entornos distribuidos empresariales o de Internet. Incluye a J2SE.

Proceso de desarrollo y ejecución en Java

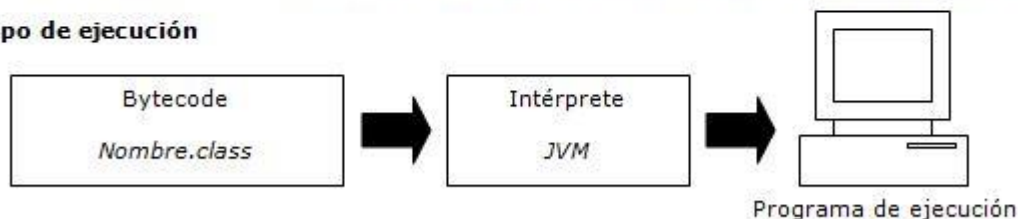
	INGENIERÍA EN INFORMÁTICA – PLAN 2003/2014 PROGRAMACIÓN DISTRIBUIDA Y COMPONENTES – 9º CUATRIMESTRE	
	APUNTE DE JAVA	VERSIÓN: 1.1 VIGENCIA: 03-05-2014

Tiempo de compilación



Nota: El archivo .class no es ejecutable, es un archivo de código intermedio que se interpreta.

Tiempo de ejecución



Nota: El código intermedio pasa por un verificador antes de su ejecución.

Mi primer programa

Nuestra primera aplicación visualizará un mensaje en consola. Para ello, debemos considerar los siguientes ítems:

1. Una aplicación Java requiere al menos una clase con acceso público. El nombre lo escribiremos en mayúscula, por una convención del lenguaje.
2. La clase debe tener el método **public static void main(String[] args)** o **public static void main(String args[])**. Este método es el que la máquina virtual de Java busca cuando inicia una aplicación.
El método *main* debe declararse con el especificador de acceso *public*, lo que quiere decir que puede llamarse desde fuera de su clase. También debe declararse como *static*, que significa que *main* es un método de una clase no un método de un objeto. Cuando se termine de ejecutar, no debe devolver ningún valor de retorno, por lo cual usamos la palabra *void*. Finalmente, el argumento *String[] args* que también puede escribirse como *String args[]* indica que al método se le pasa un *vector* cuyos elementos son cadenas de caracteres.
3. El archivo con el código fuente **debe tener el mismo nombre que la clase** con la extensión .java.

Veamos un ejemplo,

Archivo: Run.java

```

public class Run {


    public static void main(String[] args) {

        System.out.println("¡Hola Mundo :)!!!");

    }

}
  
```

La clase *System* pertenece al paquete *java.lang*, incluida en cualquier aplicación Java, lo que quiere decir que no hay que hacer nada especial para utilizarlo, a diferencia de otros paquetes.

	INGENIERÍA EN INFORMÁTICA – PLAN 2003/2014 PROGRAMACIÓN DISTRIBUIDA Y COMPONENTES – 9º CUATRIMESTRE	
	APUNTE DE JAVA	VERSIÓN: 1.1 VIGENCIA: 03-05-2014

La clase *System* incluye un miembro de datos llamado *out* que tiene un método llamado *println*, que permite que se visualice un texto.

Debemos tener en cuenta que toda línea de código termina con un punto y coma (;), característica que se hereda de C y C++.

Nuestro proyecto lo llamaremos "HolaMundo" y la clase estará almacenada dentro de un paquete cuyo nombre será "ar.edu.ubp.pdc.holamundo", nótese que los paquetes se escriben todo en minúscula.

Sintaxis de Java

Variables

Las variables pueden ser de diferentes tipos y actúan como gestores de memoria de los datos. Los diferentes tipos tienen que ver con el formato de los datos que se almacenan en ellas, así como con la memoria que es necesaria para gestionar ese dato.

Antes de usar una variable en Java, debe declarársela, especificando su tipo. La sintaxis es:

```
tipo nombre [ = valor ] [, nombre [ = valor ]... ];
```

Veamos un ejemplo,

```
public class Run {
    public static void main(String[] args) {
        int dias = 365;
        System.out.println("Cantidad de días = " + dias);
    }
}
```

Tipos de datos primitivos

Java pone mucho énfasis en sus tipos de datos. Es un lenguaje que insiste en que las variables sencillas que se declaran y se utilizan deben corresponderse con la lista establecida. Todas las variables sencillas deben tener un tipo (y de hecho, toda expresión, toda combinación de términos que Java puede evaluar para obtener un resultado, también tiene un tipo). Además, Java es muy particular para mantener la integridad de esos tipos, especialmente si se intenta asignar un valor de un tipo a una variable de otro tipo.

Grupo	Tipo de Datos	Almacenamiento en bytes	Rango de Valores
Booleanos	boolean	2	true, false
Enteros	byte	1	-128 a 127
	short	2	-32.768 a 32.767


	int	4	-2.147.483.648 a 2.147483.647
	long	8	- 9.223.372.036.854.775.808 a 9.223.372.036.854.775.807
Números con coma flotante	float	4	-3,402823E38 a - 1,401298E-45 para valores negativos 1,401298E-45 a - 3,402823E38 para valores positivos
	double	8	-1,79769313486232E308 a -4,94065645841247E-324 para valores negativos 4,94065645841247E-324 a 1,79769313486232E308 para valores positivos
Caracteres	char	2	N/A

Consideraciones en la creación de *literales*:

1. Los valores *long* pueden tener más dígitos que los *int*, por lo que Java proporciona una forma explícita de crear constantes *long*, se añade "L" al final del literal.
2. Los números en coma flotante que se utilizan como literales son de tipo *double*, no de tipo *float*. Se pueden cambiar añadiendo una "f" o "F" al final del literal para convertirlo en *float* o una "d" o "D" para pasarlo a *double*.
3. La forma básica de un literal de tipo carácter en Java es un valor que corresponde a un carácter del conjunto de caracteres Unicode. Los literales de tipo carácter son números que actúan como índices dentro del conjunto de caracteres Unicode. También, se puede referir al código Unicode para una letra con un literal de tipo carácter encerrado entre comillas simples.

Existen un conjunto caracteres que comienzan por \. Las secuencias de escape son:

Caracter	Significado
\'	Comilla simple
\"	Comillas dobles
\\	Barra invertida
\b	Espacio en blanco
\ddd	Caracter octal
\f	Avance
\n	Nueva línea
\r	Retorno de carro
\t	Tabulación

	INGENIERÍA EN INFORMÁTICA – PLAN 2003/2014 PROGRAMACIÓN DISTRIBUIDA Y COMPONENTES – 9º CUATRIMESTRE	
	APUNTE DE JAVA	VERSIÓN: 1.1 VIGENCIA: 03-05-2014

Caracter	Significado
\uxxx	Carácter Unicode en hexadecimal

Veamos algunos ejemplos,

Archivo: Run.java

```
public class Run {

    public static void main(String[] args) {

        boolean b = true;
        System.out.println("Boolean = " + b);

        int i = 16;
        System.out.println("16 decimal = " + i);

        i = 020;
        System.out.println("20 Octal = " + i + " en decimal");

        i = 0x10;
        System.out.println("10 hexadecimal = " + i + " en decimal");

        long l = 1234567890123456789L;
        System.out.println("Long = " + l);

        float f = 1.5f; // o 1.5F
        System.out.println("Float = " + f);

        double d = 1.5; //o 1.5d o 1.5D
        System.out.println("Double = " + d);

        char c = 'C';
        System.out.println("La tercera letra del alfabeto = " + c);

        c = 67; //Caracter del conjunto de caracteres Unicode
        System.out.println("La tercera letra del alfabeto = " + c);

        System.out.println("\"Este es\nun texto\nen varias líneas\"");

    }

}
```

Inicialización dinámica

Así como se asignan valores constantes a variables, se puede asignar cualquier expresión, es decir, asignar cualquier combinación de términos Java que da un valor, a una variable cuando es declarada siempre que la expresión sea válida en ese momento. Es decir, que la inicialización será determinada en tiempo de ejecución, por ello, se llama inicialización dinámica. Veamos un ejemplo,

Archivo: Run.java

	INGENIERÍA EN INFORMÁTICA – PLAN 2003/2014 PROGRAMACIÓN DISTRIBUIDA Y COMPONENTES – 9º CUATRIMESTRE	
	APUNTE DE JAVA	VERSIÓN: 1.1 VIGENCIA: 03-05-2014

```

public class Run {

    public static void main(String[] args) {

        int i1 = 2, i2 = 3, i3;

        i3 = i1 * i2;
        System.out.println("Resultado = " + i3);

    }

}

```

Conversión de tipos de datos

Java es un lenguaje muy insistente con los tipos, y como consecuencia de ello, con frecuencia nos enfrentamos a la situación de asignar una variable de un tipo a una variable de otro. Hay dos formas de hacerlo: contando con una conversión automática y haciendo explícitamente un cast de tipos.

Conversiones automáticas

Cuando se asigna un tipo de dato a una variable de otro tipo, Java convertirá el dato al nuevo tipo de variable de forma automática si las dos condiciones siguientes son verdaderas:

- Los tipos de datos y de las variables son compatibles.
- El tipo de destino tiene un rango mayor que el de origen.

Este tipo de conversiones, en las que se convierte a un tipo de datos que tiene mayor rango, se llaman extensión de conversiones. En ellas, los tipos numéricos, como entero o coma flotante, son compatibles entre sí. Por otro lado, los tipos char y boolean no son compatibles entre sí y tampoco con los tipos numéricos.


Casting a nuevos tipos de datos

Si se está asignando un valor que es de un tipo con un rango mayor que la variable a la que se le está asignando, se está ejecutando lo que se denomina estrechamiento de conversiones. El compilador de Java no las ejecuta automáticamente, ya que se perdería la posibilidad de precisión.

Si se quiere hacer estrechamiento de conversiones, se debe usar explícitamente un cast, que tiene el siguiente formato:

(tipo de dato de destino) valor

Si no se hace explícitamente el cast, el compilador devolverá error, pero con el cast de tipos, no hay problema, ya que Java decide que se conoce la posibilidad de perder algunos datos cuando se introduce un valor probablemente mayor en un tipo más pequeño. Por ejemplo, cuando se pone un número en coma flotante en un *long*, la parte fraccional del número se truncará, y puede que se pierdan más datos si el valor en coma flotante está fuera del rango que un *long* puede gestionar.

	INGENIERÍA EN INFORMÁTICA – PLAN 2003/2014 PROGRAMACIÓN DISTRIBUIDA Y COMPONENTES – 9º CUATRIMESTRE	
	APUNTE DE JAVA	VERSIÓN: 1.1 VIGENCIA: 03-05-2014

Algo a tener en cuenta es que el compilador de Java convierte a un tipo de mayor precisión, si es necesario, al evaluar expresiones. Por ejemplo, vamos a considerar el siguiente código, en el que parece que solo hay bytes involucrados:

Archivo: Run.java

```
public class Run {
    public static void main(String[] args) {
        byte b1 = 100, b2 = 100, b3;

        b3 = b1 * b2 / 100; //Error ya que b3 debería ser del tipo de dato int... este
código no COMPILA!
        System.out.println("Resultado = " + b3);
    }
}
```

Como Java sabe que de la multiplicación de tipos puede resultar valores del tamaño de un entero, automáticamente convierte el resultado de $b1 * b2$ en un entero, lo que quiere decir que realmente hay que usar explícitamente un cast para mantener el tipo byte:

Archivo: Run.java

```
public class Run {
    public static void main(String[] args) {
        byte b1 = 100, b2 = 100, b3;


        b3 = (byte)(b1 * b2 / 100);
        System.out.println("Resultado = " + b3);
    }
}
```

Arrays

Los tipos sencillos son buenos para almacenar datos simples, pero con frecuencia, los datos son más complejos. Utilizando un *array*, se podrá agrupar tipos de datos sencillos en estructuras más complejas y hacer referencia a esa nueva estructura por su nombre. Lo que es más importante, mediante un índice numérico que comienza en 0 (cero), se podrá hacer referencia a los datos individuales almacenados en el *array*. Los corchetes indican que estamos definiendo una variable del tipo *array*. Veamos un ejemplo,

Archivo: Run.java

```
public class Run {
    public static void main(String[] args) {
        double [] nros;
```


	INGENIERÍA EN INFORMÁTICA – PLAN 2003/2014 PROGRAMACIÓN DISTRIBUIDA Y COMPONENTES – 9º CUATRIMESTRE	
	APUNTE DE JAVA	VERSIÓN: 1.1 VIGENCIA: 03-05-2014

```

        nros = new double[20];

        nros[7] = 7.12;
        System.out.println("Número en posición 8 = " + nros[7]);

    }

}

```

Además, se puede inicializar un *array* con valores cuando se declara, encerrando entre llaves la lista de valores. Veamos otro ejemplo,

Archivo: Run.java

```

public class Run {

    public static void main(String[] args) {

        double [] nros = {0, 2.03, 1.09, 5.67, 5.73, 6.2, 7.12, 3.04};
        System.out.println("Número posición 6 = " + nros[6]);

    }

}

```

En los ejemplos se han declarado *arrays* unidimensionales, también llamados vectores, es decir, es una lista de números que pueden ser indexados por otro número. Sin embargo, en Java, los *arrays* pueden tener muchas dimensiones, lo que significa que se pueden tener muchos índices.

Se pueden declarar *arrays* multidimensionales de la misma forma que se declaran los unidimensionales, solo con incluir un par de corchetes para cada dimensión del *array*.

tipo nombre[][][]...;


Los *arrays* multidimensionales son realmente *arrays* de *arrays*, lo que significa que si se tiene un *array* de dos dimensiones (*array*[][]), se puede tratar como un *array* de *arrays* unidimensionales, al que se puede acceder con *array*[0], *array*[1], *array*[2], y así sucesivamente.

Cadenas

En Java, las cadenas se gestionan como si fueran objetos. Una de las ventajas de esto es que un objeto de tipo *string* tiene gran variedad de métodos que se pueden usar. Hay dos clases *string* en Java: *String* y *StringBuilder*. Se utiliza la clase *String* para crear cadenas de texto que no se pueden modificar, si se quiere cambiar el texto actual de una cadena se debe utilizar la clase *StringBuilder*.

La clase *String* es una clase, no un tipo de dato intrínseco, lo que significa que se crean objetos de esa clase con constructores. Es una clase muy poderosa, ya que cuenta con un conjunto de métodos que permiten manipular ampliamente una cadena.

Por otra parte, la clase *StringBuilder* proporciona las mismas prestaciones de la clase *String*, más que permite modificar la cadena actual. Si se crea un objeto *StringBuilder* vacío este se inicializa con 16 espacios en blanco, por defecto. Veamos un ejemplo,

	INGENIERÍA EN INFORMÁTICA – PLAN 2003/2014 PROGRAMACIÓN DISTRIBUIDA Y COMPONENTES – 9º CUATRIMESTRE	
	APUNTE DE JAVA	VERSIÓN: 1.1 VIGENCIA: 03-05-2014

Archivo: Run.java

```
public class Run {

    public static void main(String[] args) {

        StringBuilder sb = new StringBuilder();
        sb.append("¡Hola a todos ");
        sb.append("desde Java!!!");
        System.out.println(sb);

    }

}
```

Operadores

Operadores matemáticos

Descripción	Símbolo
Multiplicación	*
División	/
Resto de una división entera	%
Suma	+
Resta	-
Incremento	++
Decremento	--

Operadores de comparación

Descripción	Símbolo
Igualdad	==
Desigualdad	!=
Menor que	<
Mayor que	>
Menor o igual que	<=
Mayor o igual que	>=

Operadores lógicos

Descripción	Símbolo
No lógico	!

	INGENIERÍA EN INFORMÁTICA – PLAN 2003/2014 PROGRAMACIÓN DISTRIBUIDA Y COMPONENTES – 9º CUATRIMESTRE	
	APUNTE DE JAVA	VERSIÓN: 1.1 VIGENCIA: 03-05-2014


Descripción	Símbolo
NOT unitario a nivel de bit	~
AND en cortocircuito	&&
AND a nivel de bit	&
OR en cortocircuito	
OR a nivel de bit	
XOR a nivel de bit	^

Operadores de desplazamiento

Descripción	Símbolo
Desplazamiento de bits hacia la izquierda	<<
Desplazamiento de bits hacia la derecha	>>
Desplazamiento a la derecha con relleno de ceros	>>>

Operadores de asignación

Descripción	Símbolo
Asignación de multiplicación	*=
Asignación de división	/=
Asignación de módulo	%=
Asignación de Suma	+=
Asignación de Resta	-=
Asignación de AND	&=
Asignación de OR	=
Asignación de XOR	^=
Asignación del	<<=

	INGENIERÍA EN INFORMÁTICA – PLAN 2003/2014 PROGRAMACIÓN DISTRIBUIDA Y COMPONENTES – 9º CUATRIMESTRE	
	APUNTE DE JAVA	VERSIÓN: 1.1 VIGENCIA: 03-05-2014

Descripción	Símbolo
desplazamiento de bits hacia la izquierda	
Asignación del desplazamiento de bits hacia la derecha	>>=
Asignación del desplazamiento de bits hacia la derecha con relleno de ceros	>>>=

Operador especial if-then-else

Hay un operador Java que funciona como una sentencia if-else, el operador ternario (? :). Este operador se llama ternario porque involucra tres operandos, una condición y dos valores:

valor = condición ? valor1 : valor2;

Si la condición es verdadera, el operador ?: devuelve valor1, y en caso contrario devuelve valor 2. De esta forma, la sentencia precedente funciona como la siguiente sentencia if:

```
if (condición) {
    valor = valor1;
}
else {
    valor = valor2;
}
```

Estructuras de control


Bifurcaciones condicionales

Una bifurcación condicional es una estructura que realiza una tarea u otra dependiendo del resultado de evaluar una condición. La primera que vamos a estudiar es la estructura if...else. Esta estructura es la más sencilla y antigua de todas:

```
if (condición) {
    acciones
}
[else {
    acciones
}]
```

Hay que indicar que la sentencia else es opcional.

La siguiente estructura bifurca según los distintos valores que pueda tomar una variable específica. Es la sentencia switch:

	INGENIERÍA EN INFORMÁTICA – PLAN 2003/2014 PROGRAMACIÓN DISTRIBUIDA Y COMPONENTES – 9º CUATRIMESTRE	
	APUNTE DE JAVA	VERSIÓN: 1.1 VIGENCIA: 03-05-2014

```

switch (variable) {
    case valor1:
        acciones1;
        [break;]
    case valor2:
        acciones2;
        [break;]
    .....
    .....
    case valorN:
        accionesN;
        [break;]
    default:
        acciones por defecto;
}

```

Veamos cada una de las partes:

1. *case valor1*: En el caso de que la variable tenga el valor "valor1", realizará las acciones "acciones1".
2. *break*: Si no se incluye esta sentencia después de cada "case", se realizarían todos los casos del "switch" hasta el final. De este modo, sólo se realizarán las acciones referentes al "case" concreto.
3. *default*: Si el valor de la variable no concuerda con ningún case, se realizarán las acciones de la rama "default".

Bucle

Un bucle es una estructura que permite repetir una tarea un número de veces, determinado por una condición. Para hacer bucles podemos utilizar las estructuras while y do...while. Estos bucles iteran indefinidamente mientras se cumpla una condición. La diferencia entre ellas es que la primera comprueba dicha condición antes de realizar cada iteración y la segunda lo hace después:

```

while (condición) {
    acciones
}

do {
    acciones
}
while (condición);

```

En Java, el bucle for tiene la siguiente estructura:

```

for (inicio; condición; incremento) {
    acciones
}

```

Para recorrer un *array*, rápidamente podemos usar la sentencia foreach:

```

for (variable : array) {
    acciones
}

```

	INGENIERÍA EN INFORMÁTICA – PLAN 2003/2014 PROGRAMACIÓN DISTRIBUIDA Y COMPONENTES – 9º CUATRIMESTRE	
	APUNTE DE JAVA	VERSIÓN: 1.1 VIGENCIA: 03-05-2014

Veamos un ejemplo de este último caso,

Archivo: Run.java

```
public class Run {
    public static void main(String[] args) {
        double [] nros = {0, 2.2, 4.3, 5.6, 5.7, 6.2, 7.12, 3.04};
        for(double n : nros) {
            System.out.print(n + "\t");
        }
    }
}
```

Por último, hay que decir que la ejecución de la sentencia *break* dentro de cualquier parte del bucle provoca la salida inmediata del mismo. Aunque a veces no hay más remedio que utilizarlo, es mejor evitarlo para mejorar la legibilidad del código.

A su vez, existe la sentencia *continue* para retornar al principio del bucle en cualquier momento sin ejecutar las líneas que haya por debajo de dicha sentencia.

Entrada y Salida Estándar

La entrada desde teclado y la salida a pantalla se administran a través de la clase *System*. Esta clase pertenece al paquete *java.lang*, incluida en cualquier aplicación Java.

La Clase *System* contiene entre otros dos objetos para manipular la entrada y salida de datos por consola, *out* e *in*.

System.out

Permite mostrar mensajes y resultados en la consola. Pertenece a la clase *PrintStream*. Entre sus métodos encontramos:


- *System.out.print(String)*: Muestra un mensaje en pantalla.
- *System.out.println(String)*: Igual que el anterior, pero al final agrega un salto de línea.

System.in

Permite la entrada de datos por consola. Pertenece a la clase *InputStream*. Entre sus métodos encontramos:

int System.in.read(): Lee un carácter ASCII entre 0 y 255 (1 byte) y retorna un valor del tipo *int*. Puede producir error en tiempo de ejecución ya que se depende del ingreso de datos por parte del usuario.

En Java, a los errores en tiempo de ejecución se los conoce con el nombre de *excepciones*. Las excepciones requieren ser consideradas para tratarlas adecuadamente o, o de última, definir que

	INGENIERÍA EN INFORMÁTICA – PLAN 2003/2014 PROGRAMACIÓN DISTRIBUIDA Y COMPONENTES – 9º CUATRIMESTRE	
	APUNTE DE JAVA	VERSIÓN: 1.1 VIGENCIA: 03-05-2014

la aplicación puede lanzar una determinada excepción. Veamos el ejemplo donde se remarca que el método *main* lanza una excepción del tipo *IOException*.

Archivo: Run.java

```
public class Run {

    public static void main(String[] args) throws IOException {

        char c;
        String p = "";

        System.out.println("Ingrese una palabra: ");
        do {
            c = (char)System.in.read();
            p += c;

        } while (c != '\n');

        System.out.println("La palabra ingresada: " + p.toUpperCase());

    }

}
```

Lectura de cadenas

Para poder leer una línea completa de texto es necesario declarar un objeto del tipo *BufferedReader*. Veamos un ejemplo,

Archivo: Run.java

```
import java.io.*;

public class Run {

    public static void main(String[] args) throws IOException {

        String apellido, nombres;
        int edad;
        float peso;
        BufferedReader entrada = new BufferedReader(new InputStreamReader(System.in));

        System.out.println("Ingrese su apellido: ");
        apellido = entrada.readLine();

        System.out.println("Ingrese sus nombres: ");
        nombres = entrada.readLine();


        System.out.println("Edad: ");
        edad = Integer.parseInt(entrada.readLine());

        System.out.println("Peso: ");
        peso = Float.parseFloat(entrada.readLine());

        System.out.println("Sus datos ingresados");

    }

}
```

	INGENIERÍA EN INFORMÁTICA – PLAN 2003/2014 PROGRAMACIÓN DISTRIBUIDA Y COMPONENTES – 9º CUATRIMESTRE	
	APUNTE DE JAVA	VERSIÓN: 1.1 VIGENCIA: 03-05-2014

```

        System.out.println("Apellido: " + apellido);
        System.out.println("Nombres: " + nombres);
        System.out.println("Edad: " + edad);
        System.out.println("Peso: " + peso);

    }

}

```

Del ejemplo anterior, podemos observar que existen una serie de clases que permiten representar a los tipos primitivos y que cuentan con métodos que permiten convertir el valor contenido en un String y viceversa, además de métodos para acceder al valor representado.

Tipo de dato primitivo	Clase
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

PROGRAMACIÓN ORIENTADA A OBJETOS


Paradigma Orientado a Objetos

La programación orientada a objetos busca encapsular datos y métodos en objetos, de forma que cada objeto sea semiautónomo, encerrando métodos y datos privados (es decir, internos). Así, el objeto puede interactuar con el resto del programa por medio de una interfaz bien definida por sus métodos públicos.

La programación orientada a objetos fue creada para gestionar programas más grandes y descomponerlos en unidades funcionales. Esto nos lleva al siguiente paso, que consiste en dividir un programa en subrutinas, ya que los objetos pueden contener múltiples subrutinas y datos. El resultado de encapsular partes de un programa en un objeto es que es concebido como un elemento sencillo y no hay que tratar todo lo que el objeto hace internamente.

En Java, la programación orientada a objetos gira sobre algunos conceptos clave: clases, objetos, miembros de datos, métodos y herencia. Básicamente estos términos significan:

- *Una clase* es una planilla desde la que se pueden crear objetos. La definición de una clase incluye especificaciones formales para la clase y cualquier dato y métodos incluidas en ella.

	INGENIERÍA EN INFORMÁTICA – PLAN 2003/2014 PROGRAMACIÓN DISTRIBUIDA Y COMPONENTES – 9º CUATRIMESTRE	
	APUNTE DE JAVA	VERSIÓN: 1.1 VIGENCIA: 03-05-2014

- *Un objeto* es una instancia de una clase, al igual que una variable es una instancia de un tipo de dato. Se puede pensar en una clase como el tipo de un objeto y se puede pensar en el objeto como una instancia de una clase. Los objetos encapsulan métodos y variables de instancia.
- *Los miembros de datos*, también llamados atributos, son esas variables que forman parte de una clase y en ellas se almacenan los datos que usa el objeto. El objeto soporta variables de instancia, cuyos valores son específicos del objeto, y variables de clase, cuyos valores son compartidos entre los objetos de esa clase.
- *Un método* es una función construida en una clase u objeto. Se pueden tener métodos de instancia y métodos de clase. Con objetos, se usan los métodos de instancia, pero se puede usar un método de clase haciendo referencia, simplemente, a la clase por su nombre, sin requerir ningún objeto.
- *Herencia* es el proceso que consiste en derivar una clase, llamada clase derivada de otra llamada clase base, se pueden utilizar los métodos de la clase base en la clase derivada.

Clases

En la programación orientada a objetos, las clases proporcionan una especie de plantilla para los objetos. Es decir, se puede considerar que una clase es un tipo de objeto; se usa una clase para crear un objeto y luego se puede llamar a los métodos del objeto.

Para crear un objeto, se invoca al constructor de una clase, que es un método que se llama igual que la clase. Este constructor crea un nuevo objeto de la clase. Como ya hemos visto, para crear un programa en Java, se necesita una clase.

Objetos

Un *objeto* es simplemente una instancia de alguna clase. Para crear un objeto, se llama al constructor de una clase, que tiene el mismo nombre que ella. Por ejemplo,

```
StringBuilder s = new StringBuilder ("Hola Mundo!!!!");
```

Miembros de datos

Los miembros de datos de un objeto se llaman *miembros de datos de instancia* o *variable de instancia* o *atributo*. Los atributos permiten describir lo que un objeto es, cuando son compartidos por todos los objetos de una clase se llaman *miembros de datos de clase* o *variables de clase*. Los miembros de datos pueden hacerse accesibles desde fuera de un objeto, o se puede hacer que sean internos al objeto para usar, de forma privada, los métodos del interior del objeto.

Es importante aplicar el principio de ocultamiento para permitir el acceso a los atributos de una clase; este sugiere que al definir una clase, no se permita que los atributos sean accesibles en forma directa desde el exterior de la clase, si no que se usen métodos de la misma para consultar sus valores o modificarlos. La idea central es que el desarrollador que usa una clase no deba preocuparse por los detalles de implementación interna de la clase, sino que simplemente use los métodos que la misma provee y se maneje en un nivel de abstracción más alto.

Métodos

Los métodos son funciones de una clase, permiten describir lo que un objeto *hace*. Generalmente los métodos se dividen en aquellos que se usan internamente en la clase, llamados métodos privados (private), los que se usan fuera de la clase y sus derivadas, llamados métodos protegidos (protected).

	INGENIERÍA EN INFORMÁTICA – PLAN 2003/2014 PROGRAMACIÓN DISTRIBUIDA Y COMPONENTES – 9º CUATRIMESTRE	
	APUNTE DE JAVA	VERSIÓN: 1.1 VIGENCIA: 03-05-2014

Los métodos privados son, generalmente, llamados en el interior del objeto por otras partes del mismo.

Java soporta dos tipos de métodos: métodos de clase y métodos de instancia. Los métodos de instancia, son invocados en objetos, es decir, los objetos son instancia de una clase. Los métodos de clase, por otro lado, son invocados en una clase. Por ejemplo, la clase `java.lang.Math` tiene un método de clase llamado `sqrt` que calcula una raíz cuadrada, y se puede usar como sigue (no es necesario un objeto),

Archivo: Run.java

```
public class Run {
    public static void main(String[] args) {
        double var = 4, sqrt;
        sqrt = Math.sqrt(var);
        System.out.println("La raíz cuadrada de " + var + " es: " + sqrt);
    }
}
```

Se invoca método de la clase Math


Herencia

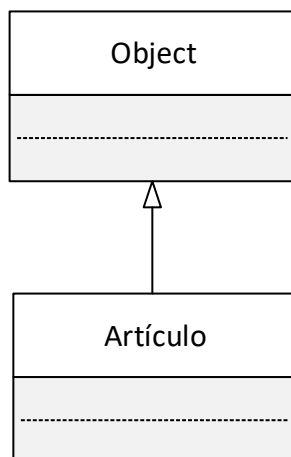
Herencia Simple

El desarrollo de software involucra un gran número de clases: muchas son variantes de otras. Para controlar la complejidad potencial resultante se necesita un mecanismo de clasificación, conocido con el nombre *herencia*. Una clase será heredera de otra si incorpora todas las características de la otra además de las propias (Un *descendiente* es un heredero directo o indirecto, la noción inversa es un *ascendente*).

La herencia genera una *jerarquía de clases*, que no es más que un conjunto de clases relacionadas por herencia. La clase desde la cual se hereda, se llama *clase base* o *super clase* o *ascendente*, y las clases que heredan desde otra se llama *subclases* o *clases derivadas* o *descendientes*. Básicamente, la clase base tiene un conjunto de características comunes a todas las clases de la jerarquía, y que por lo tanto todas ellas deberían compartir sin necesidad de hacer redeclaración de esas características. Es decir que la *clase B* hereda de la *clase A*, hace que la *clase B* incluya todos los miembros de la *clase A* como propios, a los que accederá según el calificador de acceso.

En Java, para indicar que una clase hereda de otra, se usa la palabra reservada *extends* al declarar la subclase, nombrando después de ella a la superclase. A su vez, cuando no se indica que una clase deriva de otra, entonces el lenguaje asume que esa clase hereda de la clase *Object*, que es la base de toda la jerarquía de clases. Veamos cómo se diagrama la herencia en UML (Lenguaje Unificado de Modelado),

	INGENIERÍA EN INFORMÁTICA – PLAN 2003/2014 PROGRAMACIÓN DISTRIBUIDA Y COMPONENTES – 9º CUATRIMESTRE	
	APUNTE DE JAVA	VERSIÓN: 1.1 VIGENCIA: 03-05-2014



Obsérvese que la flecha que marca la herencia entre clases tiene punta rellena y línea continua.

Herencia Múltiple

A menudo nos encontramos con la necesidad de combinar varias abstracciones. La *herencia múltiple* es la garantía de que una clase puede heredar no solo de una clase sino de tantas como esté conceptualmente justificado; permite que una *clase* herede de forma directa, y no solo transitiva, de más de una clase al mismo tiempo.


Java no soporta herencia múltiple, solo está permitida la herencia simple. Por lo tanto, al declarar una clase derivada, luego de la palabra *extends* no puede escribirse más que el nombre de una clase.

Polimorfismo

En la programación orientada a objetos existen criterios que debe respetar un lenguaje para ser considerado como tal. Cuatro de ellos están fuertemente asociados, a continuación definiremos cada uno:

1. **Comprobación estática de tipos:** Cuando la ejecución de un sistema de software causa una llamada a una cierta característica aplicada a cierto objeto, ¿cómo se puede saber que dicho objeto es capaz de admitir dicha llamada?
Para brindar una garantía de ejecución correcta, el lenguaje tiene que poseer una comprobación de tipos. Esto significa que se imponen unas pocas reglas de compatibilidad, en particular:
 - Toda entidad (es decir, todo nombre que se use en el texto de software para referirse a los objetos durante la ejecución) se declara explícitamente como perteneciente a un cierto tipo, que se deriva de una clase.
 - Toda llamada a una característica aplicada a una cierta entidad debe ser una característica de la clase correspondiente (y la característica debe estar disponible, en el sentido de ocultación de información, para la clase que contiene a la rutina que hace la llamada).
 - La asignación y el paso de argumentos están sometidos a *reglas de compatibilidad*, basadas en la herencia, que requieren que el tipo original sea compatible con el tipo de destino.

En un lenguaje que impone tal política, es posible escribir un *verificador de tipos estáticos* que aceptará o rechazará los sistemas software, garantizando que los sistemas que acepte

	INGENIERÍA EN INFORMÁTICA – PLAN 2003/2014 PROGRAMACIÓN DISTRIBUIDA Y COMPONENTES – 9º CUATRIMESTRE	
	APUNTE DE JAVA	VERSIÓN: 1.1 VIGENCIA: 03-05-2014

no causarán en ejecución errores del tipo "la característica no está disponible para el objeto".

En resumen, un sistema de tipos bien definido debería, mediante la imposición de cierto número de reglas de declaración de tipos y compatibilidad, garantizar la seguridad de tipos en ejecución de los sistemas que acepte.

2. *Redefinición*: Cuando una clase hereda de otra, quizás necesite cambiar la implementación u otras propiedades de alguna de las características heredadas. Supongamos que una clase "Sesión" que describe las sesiones de los usuarios en un sistema operativo puede tener una característica llamada *terminar* para llevar a cabo las operaciones de limpieza al finalizar cada sesión; una clase heredera podría ser "SesionRemota" que maneja sesiones iniciadas en distintas computadoras de una red. Si la terminación de una sesión remota requiere acciones suplementarias (tales como notificar a la computadora remota), la clase "SesionRemota" redefinirá la característica *terminar*. La redefinición puede afectar la implementación de una característica, su retorno (el tipo de los argumentos y del resultado) y su especificación.

En resumen, debe ser posible redefinir la especificación, el retorno y la implementación de una característica heredada.


3. *Polimorfismo*: Con la herencia en juego, la comprobación estática de tipos que se mencionó sería demasiado restrictiva si esto significase que una entidad declarada como de tipo C puede referirse solo a objetos cuyo tipo sea exactamente C. Esto significará por ejemplo que una entidad de tipo C (en un sistema de carrito compra) no podría usarse para referirse a un objeto del tipo "Lapicera" o "Conserva" suponiendo que ambas clases heredaran de una clase "Artículo".

El polimorfismo es la capacidad de una entidad consistente en poder conectarse a objetos de varios tipos. En un entorno con comprobación estática de tipos, el polimorfismo no puede ser arbitrario, sino que tiene que estar controlado por la herencia, por ejemplo, se debiera permitir que una entidad de tipo "Artículo" llegue a estar conectada a un objeto que represente a un objeto de tipo "Supermercado" que es una clase que no hereda de "Artículo".

En resumen, durante la ejecución debiera ser posible conectar entidades (nombres en los textos de software que representan objetos durante la ejecución) a objetos de distintos tipos posibles, bajo el control del sistema de tipos basado en la herencia.

4. *Ligadura dinámica*: La combinación de los últimos mecanismos mencionados, redefinición y polimorfismo, sugiere inmediatamente el siguiente. Consideremos una llamada cuyo destino es una entidad polimorfa, por ejemplo, una llamada a una característica *aplicarTasa* aplicada a una entidad declarada como de tipo "Artículo". Los diferentes descendientes de "Artículo" pueden haber redefinido esta característica de diferentes formas. Está claro que debe existir un mecanismo automático para garantizar que la versión de *aplicarTasa* será siempre la que se deduzca del tipo real del objeto, independientemente de cómo haya sido declarada la entidad. Esta propiedad recibe el nombre de ligadura dinámica.

La ligadura dinámica ejerce una influencia importante sobre la estructura de las aplicaciones orientadas a objetos, ya que permite a los desarrolladores escribir simples llamadas para denotar lo que realmente son varias posibles llamadas dependiendo de las situaciones correspondientes en la ejecución.

	INGENIERÍA EN INFORMÁTICA – PLAN 2003/2014 PROGRAMACIÓN DISTRIBUIDA Y COMPONENTES – 9º CUATRIMESTRE	
	APUNTE DE JAVA	VERSIÓN: 1.1 VIGENCIA: 03-05-2014

En resumen, la invocación a una característica sobre una entidad debe desencadenar siempre la característica correspondiente al tipo del objeto que durante la ejecución haya sido asociado a dicha entidad, y que no será necesariamente al mismo en diferentes ejecuciones de la llamada.

Programación Orientada a Objetos en Java

Declaración y creación de objetos

Antes de utilizar un objeto, es necesario declararlo. Se pueden declarar objetos de la misma forma que se declaran variables de tipos de datos sencillos, pero se puede usar la clase como tipo de objeto.

Aunque al declarar una variable simple se crea esa variable, la declaración de un objeto no la crea. Para crear el objeto se puede usar el operador *new* con el siguiente formato en el que se pasan parámetros al constructor de la clase:

```
objeto = new Clase([parámetro 1[, parámetro 2...]]);
```

A continuación veamos un ejemplo donde la referencia al objeto *s1* se copia en *s2*. En la práctica, esto significa que *s1* y *s2* se refieren al mismo objeto. Esto es importante saberlo, porque si se cambian los datos de instancia de *s1*, también se está cambiando los de *s2*, y viceversa.

Archivo: Run.java

```
public class Run {
    public static void main(String[] args) {
        StringBuilder s1 = new StringBuilder("Hola, "), s2;
        s2 = s1;
        s1 = s1.append("¿Como estás?");
        s2 = s2.append("Bien gracias!");
        System.out.println("1: " + s1);
        System.out.println("2: " + s2);
    }
}
```

s1 y s2 se refieren al mismo objeto

Declaración de acceso

Se puede usar un modificador de acceso, llamado *access* en los códigos que siguen a continuación, para fijar la visibilidad de los miembros de la clase (atributos y métodos).

En Java pueden ser cuatro: *public*, *private*, *protected* o se puede no especificar ninguno de los anteriores tomando la definición por defecto. Veamos a continuación la definición de cada uno:

- *private*: Solo es accesible desde el interior de la propia clase usando sus propios métodos.
- *protected*: Sirve en la herencia, hace que un miembro sea público para clases derivadas y para clases en el mismo paquete y privado para el resto.

- *public*: Un miembro público es accesible tanto desde el interior de la clase, como desde el exterior de la misma por otras clases de cualquier paquete.
- *sin modificador (default)*: Un miembro sin ningún modificador de acceso, asume un acceso por defecto, lo cual significa que será público para clases en el mismo paquete y privado para el resto.

Modificador	La misma clase	Otra clase del mismo paquete	Subclase de otro paquete	Otra clase de otro paquete
private	X			
protected	X	X	X	
public	X	X	X	X
"default"	X	X		

Declarar y definir clases

En Java, la creación de una clase se realiza en dos pasos: la declaración y su definición. Esta es la forma general de la declaración de una clase:

```
[access] class NombreDeLaClase [extends...] [implements...] {
    //Aquí va la definición de la clase
}
```


La implementación de la clase es lo que se llama definición de clase, y se hace en el cuerpo de la declaración. Esta es la forma general de una definición y declaración de la clase:

```
[access] class NombreDeLaClase [extends...] [implements...] {
    [access] [static] tipo variableDeInstancia1;
    ...
    [access] [static] tipo variableDeInstanciaN;

    [access] [static] tipo metodo1 ([tipo parámetro 1[, tipo parámetro 2...]]) {
        ...
    }

    [access] [static] tipo metodoN ([tipo parámetro 1[, tipo parámetro 2...]]) {
        ...
    }
}
```

Aquí la palabra reservada *static* convierte variables en variables de clases o métodos en métodos de clase en contraposición a las variables y métodos de instancia. El término *access* especifica la accesibilidad de la clase o de un atributo o método. Se usan las palabras reservadas *extends* e *implements* con la herencia.

	INGENIERÍA EN INFORMÁTICA – PLAN 2003/2014 PROGRAMACIÓN DISTRIBUIDA Y COMPONENTES – 9º CUATRIMESTRE	
	APUNTE DE JAVA	VERSIÓN: 1.1 VIGENCIA: 03-05-2014

Por convención, el nombre de la clase debe tener la primera letra de cada palabra en mayúscula y el resto en minúscula, si quiero declarar una clase con el nombre "carrito de compras" escribiré "CarritoDeCompras".

NOTA: Crear un archivo por cada "clase", recordando que el archivo debe tener el mismo nombre que la clase con extensión .java.

Crear variables de instancia

En la clase, se pueden almacenar datos de dos formas, como variables de instancia o como variables de clase. Las variables de instancia son específicas para los objetos; si se tienen dos objetos, es decir, dos instancias de una clase, las variables de instancia de cada objeto son independientes de las variables de instancia del otro objeto.

Por convención, la primera letra de la variable debe ser minúscula y el resto de las palabras deben tener su primera letra en mayúscula y el resto en minúscula, es decir, si yo quiero declarar una variable con el nombre "apellido persona" esta debo especificarla como "apellidoPersona".

```
[access] class NombreDeLaClase [extends...] [implements...] {
    [access] tipo variableDeInstancia1;
    ...
    [access] tipo variableDeInstanciaN;
}
```

Veamos un ejemplo,

Archivo: Mensaje.java

```
public class Mensaje {
    public String msj = "Este es un mensaje inicial";
}
```


Variable de instancia = Atributo del objeto

Crear variables de clase

El valor de una variable de clase es compartido por todos los objetos de esa clase, lo que significa que será el mismo para todos los objetos. Una variable se declara como estática con la palabra reservada *static*.

```
[access] class NombreDeLaClase [extends...] [implements...] {
    [access] static tipo variableDeInstancia1;
    ...
    [access] static tipo variableDeInstanciaN;
}
```

Si se quiere inicializar variables estáticas, se puede especificar el valor al declarar la variable o se puede declarar un bloque de código estático, que se etiqueta con la palabra reservada *static*; este código se ejecuta solo una vez, cuando la clase se carga por primera vez. Veamos un ejemplo,

	INGENIERÍA EN INFORMÁTICA – PLAN 2003/2014 PROGRAMACIÓN DISTRIBUIDA Y COMPONENTES – 9º CUATRIMESTRE	
	APUNTE DE JAVA	VERSIÓN: 1.1 VIGENCIA: 03-05-2014

Archivo: Datos.java

```
public class Datos {

    public static int dato1 = 1, dato2, dato3;

    static {
        Datos.dato2 = Datos.dato1 * 2;
        Datos.dato3 = 3;
    }

}
```

Variable estática =
Atributo de la clase

Bloque de inicialización

Crear constantes

La palabra reservada *final* permite declarar constantes las cuales no pueden cambiar su valor en tiempo de ejecución; estas pueden ser declaradas sin ser inicializadas, pero al momento que se especifique un valor este no podrá ser cambiando durante la ejecución del programa.

Por convención, el nombre de estas variables debe ser escrito "todo" en mayúscula separando cada palabra con guión bajo, es decir, si yo quiero declarar una constante con el nombre "ruta de acceso" esta debo especificarla como "RUTA_DE_ACCESO".

```
[access] class NombreDeLaClase [extends...] [implements...] {

    [access] final tipo CONSTANTE_1;
    ...
    [access] final tipo CONSTANTE_N;

}
```

Veamos un ejemplo,

Archivo: Run.java

```
public class Run {

    public static void main(String[] args) {

        final String MENSAJE_INICIAL = "Este es un mensaje que no puede ser
modificado";
        System.out.println(MENSAJE_INICIAL);


    }

}
```

Crear métodos

Un método es un bloque de código al que se puede transferir el control y por lo tanto, ejecutar ese código. Así es cómo se crean métodos en una clase:

```
[access] class NombreDeLaClase [extends...] [implements...] {
```


	INGENIERÍA EN INFORMÁTICA – PLAN 2003/2014 PROGRAMACIÓN DISTRIBUIDA Y COMPONENTES – 9º CUATRIMESTRE	
	APUNTE DE JAVA	VERSIÓN: 1.1 VIGENCIA: 03-05-2014

```

[access] tipo metodo1 ([tipo parámetro 1[, tipo parámetro 2...]]) {
    ...
}

[access] tipo metodoN ([tipo parámetro 1[, tipo parámetro 2...]]) {
    ...
}

}

```

Para declarar y definir un método, se puede usar un modificador de acceso y especificar el tipo de retorno del método si se quiere que devuelva un valor. Ejemplos de ellos son *int*, *float*, tipo de *object* o *void*, si el método no devuelve ningún valor. Se da el nombre al método y se sitúa la lista de parámetros que se le quieren pasar después de ese nombre, separándolos con coma y poniéndolos entre paréntesis. El cuerpo actual del método, el código que se ejecutará cuando se le llame, está encerrado en un bloque de código que sigue a la declaración del método.

Por convención, el nombre de los métodos se escribe con la primera letra en minúscula y el resto de las palabras con la primera letra en mayúscula y el resto en minúscula, es decir, si mi método se llama "insertar dato" lo escribiré como "insertarDato".

En un método se utiliza la sentencia *return* para devolver un valor desde un método y en la declaración del método se indica el tipo de valor de retorno.

El tipo de retorno puede ser cualquier tipo de los que Java reconoce, por ejemplo, *int*, *float*, *double*, el nombre de una clase que se ha definido, *int[]* para devolver un *array* de enteros, etc.

Crear métodos de clase

Para hacer que un método sea un método de clase, se debe utilizar la palabra reservada *static*:

```

[access] class NombreDeLaClase [extends...] [implements...] {

    [access] static tipo metodo1 ([tipo parámetro 1[, tipo parámetro 2...]]) {
        ...
    }

    [access] static tipo metodoN ([tipo parámetro 1[, tipo parámetro 2...]]) {
        ...
    }


}

```

Si se declara un método estático, internamente solo puede llamar a otros métodos estáticos y acceder a datos estáticos. Además, no puede usar las palabras reservadas *this* y *super*, que hacen referencia al objeto actual y a su padre, respectivamente.

Crear métodos de acceso a datos

Para poder acceder a los atributos privados de una clase, se requiere un método de consulta y otro de actualización, siempre que el desarrollador considere que se permiten estas operaciones sobre el atributo.

	INGENIERÍA EN INFORMÁTICA – PLAN 2003/2014 PROGRAMACIÓN DISTRIBUIDA Y COMPONENTES – 9º CUATRIMESTRE	
	APUNTE DE JAVA	VERSIÓN: 1.1 VIGENCIA: 03-05-2014

La convención utilizada en Java es que el método de consulta comience con la palabra *get* seguida del nombre del atributo, tiene retorno del tipo de dato del atributo y no recibe parámetros. Mientras que el método de actualización debe comenzar con *set* seguido del nombre del atributo, no tiene retorno y recibe un único parámetro del mismo tipo de dato que el atributo al que se asignará el valor. Veamos un ejemplo,

Archivo: Mensaje.java

```
public class Mensaje {
    private String msj;

    public void setMsj(String m) {
        this.msj = m;
    }

    public String getMsj() {
        return this.msj;
    }
}
```

Método para setear el valor a un atributo

Método para obtener el valor de un atributo

Sobrecarga de métodos

La sobrecarga de métodos es una técnica de la orientación a objetos que permite definir diferentes versiones de un método, todos con el mismo nombre pero con diferentes listas de parámetros. Cuando se usa un método sobrecargado, el compilador de Java sabrá cuál es el que se quiere utilizar por el número y/o tipo de parámetros que se le pasen, y buscará la versión del método con la lista de parámetros correcta.

Para sobrecargar un método, solo hay que definirlo más de una vez, especificando una nueva lista de parámetros en cada una de ellas. Cada lista de parámetros debe ser diferente de cualquier otra de alguna forma, ya sea por el número de parámetros o el tipo de uno o más de ellos.

Es importante tener en cuenta, que no se toma como sobrecarga cambiar el tipo de salida del método. Veamos un ejemplo,


Archivo: Calculos.java

```
public class Calculos {
    public static double sumar(double op1, double op2) {
        return op1 + op2;
    }

    public static double sumar(double op1, double op2, double op3) {
        return op1 + op2 + op3;
    }

    public static float sumar(float op1, float op2) {
        return op1 + op2;
    }

    public static float sumar(float op1, float op2, float op3) {
        return op1 + op2 + op3;
    }
}
```

	INGENIERÍA EN INFORMÁTICA – PLAN 2003/2014 PROGRAMACIÓN DISTRIBUIDA Y COMPONENTES – 9º CUATRIMESTRE	
	APUNTE DE JAVA	VERSIÓN: 1.1 VIGENCIA: 03-05-2014

```

    }

    public static int sumar(int op1, int op2) {
        return op1 + op2;
    }

    public static int sumar(int op1, int op2, int op3) {
        return op1 + op2 + op3;
    }

    public static int sumar(short op1, short op2) {
        return op1 + op2;
    }

    public static int sumar(short op1, short op2, short op3) {
        return op1 + op2 + op3;
    }
}

```

Archivo: Run.java

```

public class Run {

    public static void main(String[] args) {

        int int1 = 1, int2 = 2;
        float float1 = 1.2f, float2 = 1.24f;
        double double1 = 2.03, double2 = 4.18;

        System.out.println("Suma (int): " + Calculos.sumar(int1, int2));
        System.out.println("Suma (float): " + Calculos.sumar(float1, float2));
        System.out.println("Suma (double): " + Calculos.sumar(double1, double2));

    }
}

```


Crear constructores

Crear un constructor para una clase es fácil, basta con añadir un método a una clase con el mismo nombre que la clase, sin ningún tipo de retorno (ni siquiera void), con la posibilidad de recibir parámetros como cualquier otro método. El constructor tiene como objetivo inicializar los atributos de la clase. Es invocado automáticamente al crear una instancia de la clase con el operador *new*.

Si el desarrollador no declara ningún constructor el compilador siempre crea un constructor por defecto sin parámetros (constructor nulo). Si este decidiera incluir algún constructor que no sea el nulo, entonces el compilador no incluirá este último en el archivo bytecode y cualquier creación de una instancia sin especificar parámetros al constructor no compilará.

Sobrecarga de constructores

La sobrecarga de constructores funciona como la sobrecarga de métodos, solo hay que definir el constructor un número de veces, cada una con una lista de parámetros diferentes.

	INGENIERÍA EN INFORMÁTICA – PLAN 2003/2014 PROGRAMACIÓN DISTRIBUIDA Y COMPONENTES – 9º CUATRIMESTRE	
	APUNTE DE JAVA	VERSIÓN: 1.1 VIGENCIA: 03-05-2014

Usar la palabra reservada *this*

Cuando un objeto invoca a un método propio, este está presente como *parámetro implícito*, no es necesario nombrarlo para acceder a sus atributos o invocar métodos que se apliquen sobre él. Pero es conveniente por cuestiones de claridad, utilizar la palabra reservada *this*, que hace referencia a ese objeto implícito u objeto actual.

Si al usar la palabra reservada *this* le sigue un punto, se busca acceder a un miembro de la clase y si luego de *this* se abre un paréntesis y se envían parámetros, como si fuera un método, entonces se está invocando a algún constructor de la clase.

Veamos un ejemplo que resume alguno de los temas tratados al momento,

Archivo: Artículo.java

```
public class Artículo {

    private String nomArticulo;
    private String unidadMedida;
    private double importeUnitario;
    private int cantidadSolicitada;

    public Artículo() {
        this.nomArticulo = null;
        this.unidadMedida = "Unidad";
        this.importeUnitario = 1.000;
        this.cantidadSolicitada = 0;
    }

    public Artículo(String nomArticulo, String unidadMedida, double importeUnitario,
int cantidadSolicitada) {
        this.nomArticulo = nomArticulo;
        this.unidadMedida = unidadMedida;
        this.importeUnitario = importeUnitario;
        this.cantidadSolicitada = cantidadSolicitada;
    }

    public Artículo(String nomArticulo, String unidadMedida, double importeUnitario)
    {
        this(nomArticulo, unidadMedida, importeUnitario, 0);
    }

    public Artículo(String nomArticulo, String unidadMedida)
    {
        this(nomArticulo, unidadMedida, 1.0000);
    }


    public Artículo(String nomArticulo, double importeUnitario)
    {
        this(nomArticulo, "Unidad", importeUnitario);
    }

    public void setNomArticulo(String nom) {
        this.nomArticulo = nom;
    }
}
```

Variables de instancia

Constructores

Métodos de acceso a
datos o métodos
accesores

	INGENIERÍA EN INFORMÁTICA – PLAN 2003/2014 PROGRAMACIÓN DISTRIBUIDA Y COMPONENTES – 9º CUATRIMESTRE	
	APUNTE DE JAVA	VERSIÓN: 1.1 VIGENCIA: 03-05-2014

```

public void setUnidadMedida(String uni) {
    this.unidadMedida = uni;
}

public void setImporteUnitario(double imp) {
    this.importeUnitario = imp;
}

public void setCantidadSolicitada(int c) {
    this.cantidadSolicitada = c;
}

public String getNomArticulo() {
    return this.nomArticulo;
}

public String getUnidadMedida() {
    return this.unidadMedida;
}

public double getImporteUnitario() {
    return this.importeUnitario;
}

public int getCantidadSolicitada() {
    return this.cantidadSolicitada;
}

@Override
public String toString() {
    return "Nombre: " + this.nomArticulo + "\n" +
        "Unidad: " + this.unidadMedida + "\n" +
        "Imp. Unit: " + this.importeUnitario + "\n" +
        "Cant. Solicitada: " + this.cantidadSolicitada + "\n";
}
}

```

Método heredado de la clase Object

Archivo: Run.java

```

public class Run {

    public static void main(String[] args) {


        Articulo a, b, c;

        a = new Articulo();
        b = new Articulo("Cepillo", 2.34);
        c = new Articulo("Leche", "Litros");

        System.out.println("Artículo A:\n" + a);
        System.out.println("Artículo B:\n" + b);
        System.out.println("Artículo C:\n" + c);

    }
}

```

	INGENIERÍA EN INFORMÁTICA – PLAN 2003/2014 PROGRAMACIÓN DISTRIBUIDA Y COMPONENTES – 9º CUATRIMESTRE	
	APUNTE DE JAVA	VERSIÓN: 1.1 VIGENCIA: 03-05-2014

Si observamos el ejemplo, veremos que se sobrescribió el método *toString()* perteneciente a la clase *Object*.

Toda clase en Java en última instancia hereda de *Object*, la cual provee un conjunto de métodos entre los cuales encontramos el método *toString()*.

Este método sirve para retornar una cadena de caracteres con el contenido del objeto; si no se sobrescribe retorna una cadena con el nombre de la clase a la cual pertenece el objeto, más la dirección de memoria de ese objeto en formato hexadecimal.

Herencia, clases internas e interfaces

Crear una subclase

La palabra reservada *extends* permite establecer que una clase deriva de otra. Veamos algunos ejemplos,

Archivo: Lapicera.java

```
public class Lapicera extends Artículo {

    private String color;
    private String trazo;

    public Lapicera() {
        super("Lapicera", 2.03);
        this.color = null;
        this.trazo = null;
    }

    public Lapicera(String color, String trazo) {
        this();
        this.color = color;
        this.trazo = trazo;
    }

    public void setColor(String color) {
        this.color = color;
    }

    public void setTrazo(String trazo) {
        this.trazo = trazo;
    }

    public String getColor() {
        return this.color;
    }

    public String getTrazo() {
        return this.trazo;
    }


    @Override
    public String toString() {
        String s = super.toString();
        s += "Color:" + this.color + "\n";
        s += "Trazo:" + this.trazo + "\n";
    }
}
```

Se hereda de la clase "Artículo"

Se invoca al constructor de la superclase

Se invoca al constructor de la clase

Se invoca al método "toString" de la superclase

	INGENIERÍA EN INFORMÁTICA – PLAN 2003/2014 PROGRAMACIÓN DISTRIBUIDA Y COMPONENTES – 9º CUATRIMESTRE	
	APUNTE DE JAVA	VERSIÓN: 1.1 VIGENCIA: 03-05-2014

```

        return s;
    }

}

```

Archivo: Conserva.java

```

public class Conserva extends Artículo {

    private String tipo;

    public Conserva() {
        super("Conserva", 6.35);
        this.tipo = null;
    }

    public Conserva(String tipo) {
        this();
        this.tipo = tipo;
    }

    public void setTipo(String tipo) {
        this.tipo = tipo;
    }

    public String getTipo() {
        return this.tipo;
    }

    @Override
    public String toString() {
        return "Nombre: " + this.getNomArticulo() + (this.tipo == null ? "" : " de " +
this.tipo) + "\n" +
            "Unidad: " + this.getUnidadMedida() + "\n" +
            "Imp. Unit: " + this.getImporteUnitario() + "\n" +
            "Cant. Solicitada: " + this.getCantidadSolicitada() + "\n";
    }

}

```

Se hereda de la clase "Artículo"

Se invoca al constructor de la superclase

Se invoca al constructor de la clase

Archivo: Run.java

```

public class Run {

    public static void main(String[] args) {

        Conserva t;
        Lapidera l;


        t = new Conserva("Tomate");
        l = new Lapidera("Azul", "Fino");

        System.out.println("Conserva:\n" + t);
        System.out.println("Lapidera:\n" + l);

    }

}

```

	INGENIERÍA EN INFORMÁTICA – PLAN 2003/2014 PROGRAMACIÓN DISTRIBUIDA Y COMPONENTES – 9º CUATRIMESTRE	
	APUNTE DE JAVA	VERSIÓN: 1.1 VIGENCIA: 03-05-2014

Si vemos el ejemplo anterior, observaremos que en el constructor de la subclase, hemos invocado al constructor de la superclase, utilizando la palabra reservada *super*. Nótese también que cuando los atributos de la clase base son privados, no pueden ser accedidos directamente por la clase derivada, sí solo a través de los métodos de acceso a datos. Ahora bien, si estos fueran declarados con el modificador *protected*, la clase derivada podría accederlos como si fueran públicos.

Usar la palabra reservada *super*

En la herencia, al invocar un constructor de una clase derivada, Java espera que ese constructor invoque a su vez a algún constructor de la clase base. Esa invocación debería ser agregada como primera línea del bloque de acciones del constructor de la subclase. Como hemos visto la palabra reservada utilizada es *super*. Si el constructor de la clase derivada no incluye una llamada explícita a algún constructor de la base, entonces Java impone una llamada al constructor por defecto de la clase base. Lo mismo es recomendable, siempre definirlo aunque parezca que no tiene sentido.

Si al usar la palabra reservada *super* se abre un paréntesis y se envían parámetros, como si fuera un método, entonces se está invocando a algún constructor de la clase base; ahora bien, si le sigue un punto, se busca acceder a un miembro de la superclase.

Sobrescritura de métodos

Así como se puede sobrecargar un método especificando diferente lista de parámetros, también se puede sobrescribir métodos que se heredan de una superclase, lo que quiere decir que los sustituye con una nueva versión; no se cambia la declaración del método sino que se modifica el bloque de acción del mismo al ser invocado. Al sobrescribir un método en la clase derivada solo es válido el redefinido y no el heredado.

Si queremos acceder al método original sobrescrito podemos usar la palabra reservada *super* que se refiere siempre a la superclase, como ya hemos visto.

Usar la palabra reservada *final*


Como ya hemos visto, la palabra reservada *final* nos permite definir constantes pero en Java tiene dos usos más.

Si queremos evitar que un método sea sobrescrito se le agrega la palabra reservada *final* lo que impedirá que sea redefinido.

```
[access] class NombreDeLaClase [extends...] [implements...] {
    [access] final tipo metodoN ([tipo parámetro 1[, tipo parámetro 2...]]) {
        ...
    }
}
```

A su vez, si queremos que una clase no pueda ser heredada, al declarar la misma se debe agregar la palabra reservada *final*.

```
[access] final class NombreDeLaClase [extends...] [implements...] {
    //Aquí va la definición de la clase
}
```


	INGENIERÍA EN INFORMÁTICA – PLAN 2003/2014 PROGRAMACIÓN DISTRIBUIDA Y COMPONENTES – 9º CUATRIMESTRE	
	APUNTE DE JAVA	VERSIÓN: 1.1 VIGENCIA: 03-05-2014

}

Un ejemplo es la clase String de Java, la cual no permite ser heredada por usar la palabra final en su declaración.

Polimorfismo

Un aspecto de la programación orientada a objetos es que se puede asignar una referencia a un objeto de subclase a un tipo de variable de una superclase. Esto es posible, ya que Java soporta el polimorfismo en tiempo de ejecución.

El polimorfismo en tiempo de ejecución, llamada en Java, *dispatch* dinámico de métodos, permite esperar hasta que el programa se esté realmente ejecutando para especificar el tipo de objeto que estará en una variable particular del objeto.

Usando el polimorfismo en tiempo de ejecución, se puede escribir código que funcionará con diferentes tipos de objetos y se decidirá el tipo de objeto actual en tiempo de ejecución.

Veamos un ejemplo tomando como base la superclase "Artículo" y sus subclases "Lapicera" y "Conserva",

Archivo: Run.java

```
public class Run {
    public static void main(String[] args) {
        Artículo a1, a2;

        a1 = new Conserva("Atún");
        a2 = new Lapicera("Rojo", "Grueso");


        System.out.println("Conserva:\n" + a1);
        System.out.println("Lapicera:\n" + a2);
    }
}
```

Se declaran variables del tipo "Artículo"

Se crean objetos polimórficos del tipo "Conserva" y "Lapicera"

Veamos el ejemplo a continuación, si para la variable "articulo" deseamos invocar un método de la superclase "Artículo", sin importar el tipo real del objeto apuntado por "articulo", no habrá ningún tipo de inconveniente de usar el método `la.setImporteUnitario()`.

Ahora bien, si para la variable "articulo" deseamos invocar un método definido en la clase derivada, clase real del objeto apuntado por "articulo", se deberá moldear (casting) la referencia antes de poder hacerlo, para evitar error en tiempo de compilación,

	INGENIERÍA EN INFORMÁTICA – PLAN 2003/2014 PROGRAMACIÓN DISTRIBUIDA Y COMPONENTES – 9º CUATRIMESTRE	
	APUNTE DE JAVA	VERSIÓN: 1.1 VIGENCIA: 03-05-2014

Archivo: Run.java

```
public class Run {

    public static void main(String[] args) {
        Artículo articulo;
        Lapicera aux;

        articulo = new Lapicera();
        articulo.setImporteUnitario(6.35);

        aux = (Lapicera) articulo;
        aux.setColor("Rojo");

        ((Lapicera) articulo).setTrazo("Fino");
        System.out.println("Lapicera:\n" + articulo);
    }
}
```

Se crea polimórficamente el objeto "articulo"

Se invoca un método de la superclase

Para invocar un método de la clase derivada debemos moldear el objeto

Otra forma de moldear sin usar un objeto auxiliar, IMPORTANTE: Los paréntesis externos son fundamentales al moldear la variable

Veamos otro ejemplo en este caso crearemos una clase "Carrito" que permite usar objetos de la clase "Artículo",

Archivo: Carrito.java

```
public class Carrito {


    private Artículo art[];
    private int tamActual, tamMaximo;

    public Carrito() {
        this.tamActual = 0;
        this.tamMaximo = 100;
        this.art = new Artículo[tamMaximo];
    }

    public Carrito(int tamaño) {
        this.tamActual = 0;
        this.tamMaximo = tamaño;
        this.art = new Artículo[tamMaximo];
    }

    public boolean add(Artículo a) {
        boolean res = false;
        if (this.tamActual == 0 || (this.tamActual > 0 && this.tamActual <
this.tamMaximo)) {
            this.art[this.tamActual] = a;
            this.tamActual++;
            res = true;
        }
        return res;
    }
}
```

Se declara método polimórfico

 UBP UNIVERSIDAD BLAS PASCAL	INGENIERÍA EN INFORMÁTICA – PLAN 2003/2014 PROGRAMACIÓN DISTRIBUIDA Y COMPONENTES – 9º CUATRIMESTRE
	APUNTE DE JAVA

VERSIÓN: 1.1
 VIGENCIA: 03-05-2014

```

public Artículo get (int i) {
    if (i >= 0 && i < this.tamActual)
    {
        return this.art[i];
    }
    return null;
}

public double getImporteAPagar() {
    double importeTotal = 0;
    for(int i = 0; i < this.tamActual; i++) {
        importeTotal += this.art[i].getImporteUnitario() *
this.art[i].getCantidadSolicitada();
    }
    return importeTotal;
}

@Override
public String toString() {
    StringBuilder bf = new StringBuilder("Artículos Ingresados\n");
    for(int i = 0; i < this.tamActual; i++) {
        bf.append(this.art[i]);
    }
    bf.append("Total a Pagar: ");
    bf.append(this.getImporteAPagar());
    return bf.toString();
}
}
  
```

Se declara método
 polimórfico

Observemos que los métodos "add" y "get" están definidos polimórficamente, vamos a poder agregar objetos del tipo "Artículo", "Lapicera" y "Conserva" sin problemas, y luego retornarlos.

Ahora bien, si en algún momento queremos determinar la clase a la que pertenece el objeto apuntado por una referencia polimórfica, tenemos dos vías:

1. *Operador instanceof*: Para determinar si la variable "a" pertenece a la clase "Conserva", escribiremos la siguiente condición:

```
if(a instanceof Conserva)
```

2. *Método getClass*: Método de la clase "Object", retorna un objeto de la clase "Class". Los objetos de la clase "Class" representan a las clases de los objetos de la aplicación en curso. Si tenemos dos referencias "a" y "b", ya sean polimórficas o no, la siguiente condición determina si los objetos apuntados son de la misma clase real:

```
if(a.getClass() == b.getClass())
```

Veamos un ejemplo donde queremos determinar si en nuestro "Carrito" se incorporaron objetos "Conserva",

Archivo: Run.java

```
public class Run {
```

	INGENIERÍA EN INFORMÁTICA – PLAN 2003/2014 PROGRAMACIÓN DISTRIBUIDA Y COMPONENTES – 9º CUATRIMESTRE	
	APUNTE DE JAVA	VERSIÓN: 1.1 VIGENCIA: 03-05-2014

```

public static void main(String[] args) {

    Artículo a;
    int i = 0;
    boolean compro = false;
    Carrito carrito = new Carrito();

    carrito.add(new Lapicera("Roja", "Fino"));
    carrito.add(new Conserva("Tomate"));
    carrito.add(new Artículo("Otro", "Unidad", 5.00, 3));

    while((a = carrito.get(i)) != null && !compro) {
        if(a instanceof Conserva) {
            compro = true;
        }
        i ++;
    }

    if(!compro) {
        System.out.println("No compró conservas");
    }
    else {
        System.out.println("Compró conservas");
    }
}
}

```

Bloque de acción para determinar si un objeto pertenece a una clase

Clases abstractas

Cuando se escriben clases, se pueden ejecutar casos donde solo se pueda proporcionar código general, y es necesario que el desarrollador que crea la subclase de la clase las personalice. Para asegurarse de que el desarrollador personalice el código, se puede hacer que el método sea abstracto, lo que significa que el desarrollador tendrá que sobrescribir el método, de lo contrario, la nueva clase no compilará. Para hacer un método abstracto, se usa la palabra reservada *abstract*. Si se hace algún método abstracto en una clase, también deberá serlo la clase. Es decir, que las clases abstractas nos permiten agrupar características, facilitar la herencia y posibilitar el polimorfismo.

Este tipo de clases no pueden ser instanciadas ya que provocan error de compilación, es decir, si nosotros declaramos nuestra clase "Artículo" como abstracta, luego no podremos crear ninguna variable de instancia de su tipo,

```
Articulo a = new Articulo(); //Si la clase es abstracta ya no compila
```

Supongamos que en nuestra clase "Artículo" contamos con un nuevo método *aplicarTasa* que posibilita aplicar una bonificación o recargo al importe del artículo, como puede ser distinto según el artículo, dicho método lo declararemos abstracto para que sea codificado en las clases derivadas. Cada una de las clases derivadas está obligada a sobrescribir el método o a declararlo nuevamente abstracto y por lo tanto convertirse en ella misma en una nueva clase abstracta.

Veamos la redefinición de nuestra clase "Artículo",

Archivo: Articulo.java

```
public abstract class Articulo {

    private String nomArticulo;
    private String unidadMedida;
    private double importeUnitario;
    private int cantidadSolicitada;

    public Articulo() {
        this.nomArticulo = null;
        this.unidadMedida = "Unidad";
        this.importeUnitario = 1.000;
        this.cantidadSolicitada = 0;
    }

    public Articulo(String nomArticulo, String unidadMedida, double importeUnitario,
int cantidadSolicitada) {
        this.nomArticulo = nomArticulo;
        this.unidadMedida = unidadMedida;
        this.importeUnitario = importeUnitario;
        this.cantidadSolicitada = cantidadSolicitada;
    }

    public Articulo(String nomArticulo, String unidadMedida, double importeUnitario)
    {
        this(nomArticulo, unidadMedida, importeUnitario, 0);
    }

    public Articulo(String nomArticulo, String unidadMedida)
    {
        this(nomArticulo, unidadMedida, 1.0000);
    }

    public Articulo(String nomArticulo, double importeUnitario)
    {
        this(nomArticulo, "Unidad", importeUnitario);
    }

    public void setNomArticulo(String nom) {
        this.nomArticulo = nom;
    }


    public void setUnidadMedida(String uni) {
        this.unidadMedida = uni;
    }

    public void setImporteUnitario(double imp) {
        this.importeUnitario = imp;
    }

    public void setCantidadSolicitada(int c) {
        this.cantidadSolicitada = c;
    }

    public String getNomArticulo() {
        return this.nomArticulo;
    }
}
```

Se declara la clase como abstracta

	INGENIERÍA EN INFORMÁTICA – PLAN 2003/2014 PROGRAMACIÓN DISTRIBUIDA Y COMPONENTES – 9º CUATRIMESTRE	
	APUNTE DE JAVA	VERSIÓN: 1.1 VIGENCIA: 03-05-2014

```

    public String getUnidadMedida() {
        return this.unidadMedida;
    }

    public double getImporteUnitario() {
        return this.importeUnitario + (this.importeUnitario * this.aplicarTasa());
    }

    public int getCantidadSolicitada() {
        return this.cantidadSolicitada;
    }

    public abstract double aplicarTasa();

    @Override
    public String toString() {
        return "Nombre: " + this.nomArticulo + "\n" +
            "Unidad: " + this.unidadMedida + "\n" +
            "Imp. Unit: " + this.importeUnitario + "\n" +
            "Cant. Solicitada: " + this.cantidadSolicitada + "\n";
    }
}

```

Se declara método abstracto, por lo que la clase debe ser abstracta

Ahora, veamos la codificación de alguna de las clases derivadas de "Artículo",

Archivo: Lapicera.java

```

public class Lapicera extends Articulo {

    private String color;
    private String trazo;

    public Lapicera() {
        super("Lapicera", 2.03);
        this.color = null;
        this.trazo = null;
    }


    public Lapicera(String color, String trazo) {
        this();
        this.color = color;
        this.trazo = trazo;
    }

    public void setColor(String color) {
        this.color = color;
    }

    public void setTrazo(String trazo) {
        this.trazo = trazo;
    }

    public String getColor() {
        return this.color;
    }
}

```

	INGENIERÍA EN INFORMÁTICA – PLAN 2003/2014 PROGRAMACIÓN DISTRIBUIDA Y COMPONENTES – 9º CUATRIMESTRE	
	APUNTE DE JAVA	VERSIÓN: 1.1 VIGENCIA: 03-05-2014

```

    public String getTrazo() {
        return this.trazo;
    }

    @Override
    public double aplicarTasa() {
        if(this.getCantidadSolicitada() >= 2) {
            return -0.10;
        }
        return 1;
    }

    @Override
    public String toString() {
        String s = super.toString();
        s += "Color:" + this.color + "\n";
        s += "Trazo:" + this.trazo + "\n";
        return s;
    }
}

```

Se sobrescribe método declarado como abstracto en la superclase

Interfaces

Las interfaces son algo así como declaraciones de funcionalidades que tienen que cubrir las clases que implementan las interfaces; se puede pensar en ellas como la declaración de una clase.

En una interfaz se definen habitualmente un juego de métodos que deben codificar las clases que implementan dicha interfaz. De modo que, cuando una clase implementa una interfaz, podremos estar seguros que en su código están definidos los métodos que incluía la misma.

En Java son utilizadas para aplicar herencia múltiple, ya que una clase puede implementar más de una interfaz a la vez, es decir, una clase puede heredar de una única clase pero implementar tantas interfaces como se desee.

En su definición se utiliza la palabra reservada *interface*. A su vez, para poder definir que una clase implementa una determina interfaz, se utiliza la palabra reservada *implements*.


Otra característica importante es que facilitan el polimorfismo, por lo que las instancias de las clases que implementen la interface, por ejemplo, "Comparable" de Java podrán apuntar a una referencia polimórfica del tipo:

Comparable c;

Cuando se define una clase de interfaz, se declaran una serie de métodos sin especificar un bloque de acciones. Luego, las clases que implementen esa interfaz serán las encargadas de proporcionar un código a los métodos que contiene esa interfaz. Es importante considerar que si una clase implementa una interfaz, debería declarar todos los métodos de la clase de interfaz; si no tenemos código fuente para alguno de esos métodos, por lo menos debemos declararlos como abstractos y, por tanto, la clase también tendrá que declararse como abstracta.

En resumen, una clase de interfaz puede ser considera como una clase abstracta que solo provee métodos abstractos.

A la hora de programar un aplicativo, podemos contar con objetos que son muy diferentes y que por tanto no pertenecen a la misma jerarquía de herencia, pero que deben realizar algunas acciones comunes. Un ejemplo, sería un avión y un televisor son clases muy distintas que no pertenecen a la misma jerarquía de herencia, pero pueden encenderse y apagarse. En este caso,

	INGENIERÍA EN INFORMÁTICA – PLAN 2003/2014 PROGRAMACIÓN DISTRIBUIDA Y COMPONENTES – 9º CUATRIMESTRE	
	APUNTE DE JAVA	VERSIÓN: 1.1 VIGENCIA: 03-05-2014

podríamos construir una interfaz llamada "Encendible", que incluiría las funcionalidades de encender y apagar.

Comúnmente, las interfaces se declaran usando nombres terminados en "able" que sugieran que el objeto que implemente sus métodos adquiere esa propiedad.

Veamos la declaración de la interfaz "Encendible",

Archivo: Encendible.java

```
public interface Encendible {

    public String encender();
    public String apagar();

}
```

Se declara la interfaz y sus métodos

Veamos las clases "Televisor" y "Avión",

Archivo: Televisor.java

```
public class Televisor implements Encendible, Comparable {

    private int nroSerie;
    private String estado;

    public Televisor() {
        this.nroSerie = 0;
        this.estado = "apagado";
    }

    public Televisor(int nroSerie) {
        this();
        this.nroSerie = nroSerie;
    }

    public void setNroSerie(int nroSerie) {
        this.nroSerie = nroSerie;
    }

    public int getNroSerie() {
        return this.nroSerie;
    }

    @Override
    public String encender() {
        if(this.estado.equals("apagado")) {
            this.estado = "encendido";
            return "Encendido";
        }
        else {
            return "Ya estoy en funcionamiento";
        }
    }

    @Override
    public String apagar() {
```

Se implementan varias interfaces

Se sobrescriben métodos de la interfaz "Encendible"


```

        if(this.estado.equals("encendido")) {
            this.estado = "apagado";
            return "Apagado";
        }
        else {
            return "Ya estoy apagado";
        }
    }

    @Override
    public int compareTo(Object o) {
        if(o instanceof Televisor) {
            Televisor t = (Televisor) o;
            if(this.getNroSerie() == t.getNroSerie()) {
                return 0; //Los objetos son iguales
            }
            else if(this.getNroSerie() > t.getNroSerie()) {
                return 1; //El objeto implícito tiene un número mayor
            }
        }
        return -1; //El objeto implícito tiene un número menor o no es del tipo
esperado
    }
}

```

Se sobrescribe método de la interfaz "Comparable"

Archivo: Avion.java

```

public class Avion implements Encendible {

    private int combustible;
    private int bateria;
    private String estado;

    public Avion() {
        this.combustible = 0;
        this.bateria = 0;
        this.estado = "apagado";
    }

    public Avion(int combustible, int bateria) {
        this();
        this.combustible = combustible;
        this.bateria = bateria;
    }

    public void setBateria(int bateria) {
        this.bateria = bateria;
    }

    public void setCombustible(int combustible) {
        this.combustible = combustible;
    }

    public int getBateria() {
        return bateria;
    }
}

```

Se implementa la interfaz "Encendible"

```

public int getCombustible() {
    return combustible;
}

@Override
public String encender() {
    if(this.estado.equals("apagado")) {
        if(this.bateria > 0) {
            if(this.combustible > 0) {
                this.estado = "encendido";
                this.bateria --;
                return "Encendido";
            }
            else {
                return "No tengo combustible";
            }
        }
        else {
            return "No tengo batería";
        }
    }
    else {
        return "Ya estoy en funcionamiento";
    }
}

@Override
public String apagar() {
    if(this.estado.equals("encendido")) {
        this.estado = "apagado";
        return "Apagado";
    }
    else {
        return "Ya estoy apagado";
    }
}
}

```

Se sobrescriben métodos
de la interfaz
"Encendible"

Archivo: Run.java

```

public class Run {


    public static void main(String[] args) {
        Encendible tele1, tele2, tele3, avion1, avion2;

        tele1 = new Televisor(876);
        tele2 = new Televisor(123);
        tele3 = tele1;

        avion1 = new Avion(1000, 10);
        avion2 = new Avion();

        System.out.println("Televisor 1 " + tele1.encender());
        System.out.println("Televisor 2 " + tele2.apagar());
    }
}

```

	INGENIERÍA EN INFORMÁTICA – PLAN 2003/2014 PROGRAMACIÓN DISTRIBUIDA Y COMPONENTES – 9º CUATRIMESTRE	
	APUNTE DE JAVA	VERSIÓN: 1.1 VIGENCIA: 03-05-2014

```

        System.out.println("Avión 1 " + avion1.encender());
        System.out.println("Avión 2 " + avion2.encender());

        System.out.println("Televisor 1 es " + (((Comparable) tele1).compareTo(tele2)
== 0 ? "igual" : "distinto") + " al Televisor 2");
        System.out.println("Televisor 1 es " + (((Comparable) tele1).compareTo(tele3)
== 0 ? "igual" : "distinto") + " al Televisor 3");
    }

}

```

Interfaz *Comparable*

Interfaz incluida en el paquete *java.lang* que se carga automáticamente y está a disposición del desarrollador, es decir, que no requiere ser definida por este.

```

public interface Comparable
{
    int compareTo (Object o);
}

```

El método a sobrescribir es *compareTo* que retorna un valor numérico como resultado de la comparación entre los objetos:

`== 0`: Los objetos son iguales

`> 0`: El objeto que invoca el método es mayor al parámetro

`< 0`: El objeto que invoca el método es menor al parámetro

Clases Internas

Una *clase interna* está diseñada para auxiliar a otra clase; es por ello que accede a los atributos de esta última, también son llamadas *inner class* (*inner* no es una palabra reservada). Esta se declara dentro de otra llamada en este contexto *clase externa* o *outer class* (nuevamente *outer* no es una palabra reservada).

Si una clase "B" es clase interna de otra clase "A", entonces "B" tiene acceso a los miembros de "A" aún cuando sean privados, para "B" el contenido de "A" es público.

Una clase "B" de otra "A", tiene una referencia implícita al interior de la clase externa, y por ello puede usar todos los miembros de "A" sin problemas, incluida la referencia *this* de la clase "A".

Al compilar el proyecto se genera un archivo *.class* por cada clase que se defina separada de la principal, aunque estén en el mismo archivo fuente. Pero en el caso de las clases internas el *.class* lleva por nombre un identificador compuesto por el nombre de la clase externa, el signo \$, y el nombre de la clase interna, por ejemplo, *A\$B.class*


Veamos un ejemplo sencillo,

Archivo: A.java

```

public class A {

```

	INGENIERÍA EN INFORMÁTICA – PLAN 2003/2014 PROGRAMACIÓN DISTRIBUIDA Y COMPONENTES – 9º CUATRIMESTRE	
	APUNTE DE JAVA	VERSIÓN: 1.1 VIGENCIA: 03-05-2014

```

private B interna;
private String mensaje;

public A() {
    this.mensaje = "Este es un mensaje de la clase externa... ";
    this.interna = new B();
}

public void getMensajes() {
    System.out.println(this.mensaje);
    System.out.println(interna.getMiMensaje());
    System.out.println(interna.getMensajeExterno());
}

public class B {

    public B() {}

    public String getMiMensaje() {
        return "Soy una clase interna";
    }

    public String getMensajeExterno() {
        return A.this.mensaje + "usado por la clase interna";
    }

}

```

Se declara una variable de instancia que hace referencia a la clase interna

Se crea la instancia de la clase interna

Se declara la clase interna dentro de la clase externa

Archivo: Run.java

```

public class Run {

    public static void main(String[] args) {
        A a1 = new A();
        a1.getMensajes();
    }

}

```

Si observamos el método *getMensajeExterno* de la clase "B" veremos que accede a un miembro de la clase "A" a través del objeto implícito de esta última con la palabra reservada *this*, pero esto no es estrictamente necesario, así que podemos cambiar este método de la siguiente manera,

```

public String getMensajeExterno() {
    return A.this.mensaje + "usado por la clase interna";
}

```

Clases Internas Anónimas

Una forma corta que es útil para trabajar con la gestión de eventos, es usar clases internas anónimas. Este tipo de clase *no tiene nombre* y se crea en el momento que se necesita con el objetivo de contener, por lo general, un único método simple.

	INGENIERÍA EN INFORMÁTICA – PLAN 2003/2014 PROGRAMACIÓN DISTRIBUIDA Y COMPONENTES – 9º CUATRIMESTRE	
	APUNTE DE JAVA	VERSIÓN: 1.1 VIGENCIA: 03-05-2014

El compilador genera un *.class* para la clase anónima con un nombre compuesto por el nombre de la clase externa, el signo \$, y un número de orden que será correlativo por orden de aparición de cada clase anónima. Por ejemplo, *A\$1.class*

La sintaxis de creación es,

```
new SuperTipo(parámetros del constructor) {
    //métodos y datos
}
```

Tomemos la definición de la clase "A" para incluir una clase anónima,

Archivo: A.java

```
public class A {

    private B interna;
    private String mensaje;

    public A() {
        this.mensaje = "Este es un mensaje de la clase externa... ";
        this.interna = new B();
    }

    public void getMensajes() {
        System.out.println(this.mensaje);
        System.out.println(interna.getMiMensaje());
        System.out.println(interna.getMensajeExterno());

        (new C() {
            public void getMensajeCA() {
                System.out.println("Mensaje desde una clase anónima");
            }
        }).getMensajeCA();
    }

    public class B {

        public B() {}

        public String getMiMensaje() {
            return "Soy una clase interna";
        }

        public String getMensajeExterno() {
            return A.this.mensaje + "usado por la clase interna";
        }


    }

    private class C {}
}
```

Se crea la clase anónima

Se declara la clase interna dentro de la clase externa

Archivo: Run.java

	INGENIERÍA EN INFORMÁTICA – PLAN 2003/2014 PROGRAMACIÓN DISTRIBUIDA Y COMPONENTES – 9º CUATRIMESTRE	
	APUNTE DE JAVA	VERSIÓN: 1.1 VIGENCIA: 03-05-2014

```
public class Run {

    public static void main(String[] args) {
        A a1 = new A();
        a1.getMensajes();
    }

}
```

Enums

Es una estructura que almacena una lista de constantes predefinidas. Se considera un tipo de dato definido por el desarrollador. Se pueden declarar dentro o fuera de una clase pero nunca dentro de un método.

La sintaxis de creación es,

```
[access] enum Nombre {
    CONSTANTE_1, CONSTANTE_2,..., CONSTANTE_N
}
```

Veamos un ejemplo,

```
public enum Prioridad {
    ALTA, MEDIA, BAJA
}
```

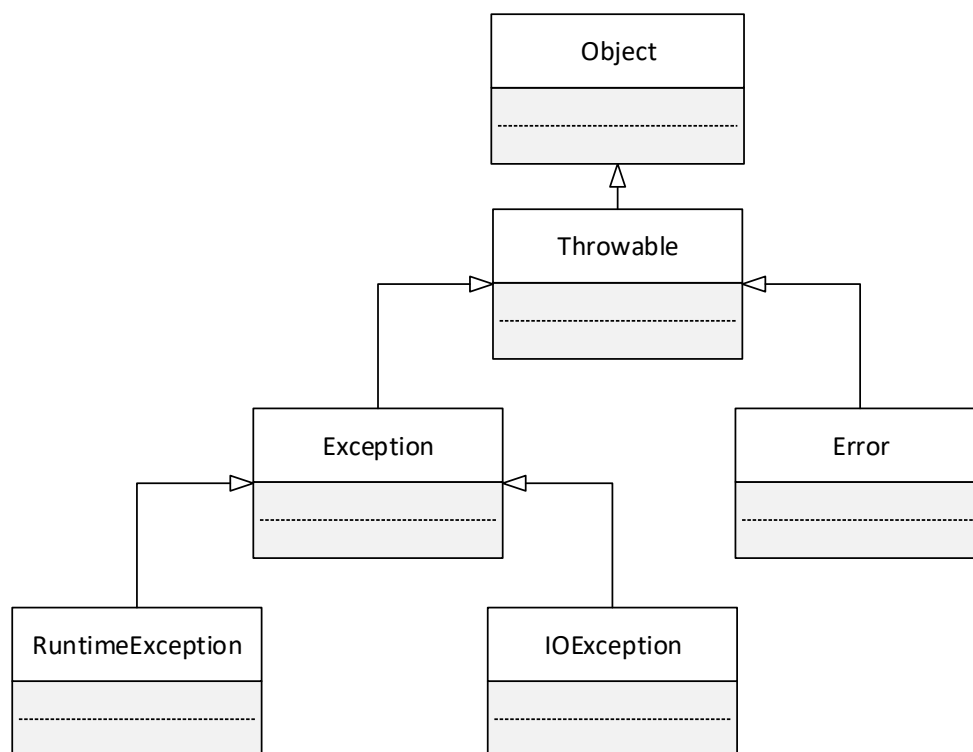
Manejo de Excepciones

Cuando un programa Java viola las restricciones semánticas del lenguaje, es decir, se produce un error, la Máquina Virtual Java (JVM) comunica este hecho al programa mediante una excepción. Por tal crea un objeto de la clase Exception o Error y notifica el hecho al sistema de ejecución. Se dice que se ha lanzado una excepción "*Throwing Exception*". El programador pueda (si lo desea) intervenir en esta situación y eventualmente recuperarse de la misma, incluso sin que la aplicación finalice de forma abrupta.

Un método se dice que es capaz de tratar una excepción "*Catch Exception*" si ha previsto el error que se ha producido y prevé también las operaciones a realizar para "recuperar" el programa de ese estado de error.

En el momento en que es lanzada una excepción, la JVM recorre la pila de llamadas de métodos en busca de alguno que sea capaz de tratar la clase de excepción lanzada. Para ello, comienza examinando el método donde ese ha producido la excepción; si este método no es capaz de tratarla, examina el método desde el que se realizó la llamada al método donde se produjo la excepción y así sucesivamente hasta llegar al último de ellos. En caso de que ningún o de los métodos de la pila sea capaz de tratar la excepción, la JVM muestra un mensaje de error y el programa termina.

Las excepciones forman parte de una jerarquía de clases cuya clase base es la clase **Throwable**:




Los errores representados por la clase **Error** son errores graves de hardware o de sistema para los que el programador no puede realizar ninguna acción adicional, solo darse por notificado del hecho. Son manejados automáticamente por la JVM. Los errores representados por la clase **Exception** son errores comunes de programación, algunos más graves que otros, para los cuales el programador **puede estar obligado** a escribir código de respuesta de esas excepciones, pues de lo contrario el programa no compilará. En ese sentido, las excepciones pueden clasificarse en "chequeadas" (**checked**) y en "no chequeadas" (**unchecked**). Los errores que pertenecen a cada grupo son:

Chequeadas (Checked)	No Chequeadas (No Checked)
IOException Excepciones programadas	Error RuntimeException

Si el código programado puede llegar a lanzar una excepción del tipo **checked**, entonces el compilador obliga a que el programador a escribir código para tratar esa posible excepción, aunque luego en la práctica la misma no llegue a lanzarse.

Si la excepción es "no chequeada", el programador **no está obligado** a escribir código de respuesta, y es solo su decisión de hacerlo o no. Todas las clases de excepción derivadas desde **RuntimeException** son no chequeadas, y también las derivadas de **Error**. Si el programador no incluye ningún código de tratamiento para estas clases de excepciones y alguna llega a producirse, simplemente el programa o el método finalizará mostrando un mensaje de error acorde a la excepción producida, pero no habrá problemas de compilación previa.

A continuación se muestra una lista de alguno de los errores más comunes:

	INGENIERÍA EN INFORMÁTICA – PLAN 2003/2014 PROGRAMACIÓN DISTRIBUIDA Y COMPONENTES – 9º CUATRIMESTRE	
	APUNTE DE JAVA	VERSIÓN: 1.1 VIGENCIA: 03-05-2014

- Error
 - OutOfMemoryError
 - InternalError
 - UnknownError
 - Etc.
- Exception
 - ClassNotFoundException
 - CloneNotSupportedException
 - IllegalAccessException
 - InstantiationException
 - InterruptedException
 - NoSuchMethodException
 - RuntimeException
 - ArithmeticException
 - StringIndexOutOfBoundsException
 - ArrayIndexOutOfBoundsException
 - ArrayStoreException
 - ClassCastException
 - IllegalMonitorStateException
 - NegativeArraySizeException
 - NullPointerException
 - SecurityException
 - IndexOutOfBoundsException
 - StringIndexOutOfBoundsException
 - ArrayIndexOutOfBoundsException
 - IllegalArgumentException
 - NumberFormatException
 - IllegalThreadStateException
 - Etc.
 - IOException
 - EOFException
 - FileNotFoundException
 - InterruptedIOException
 - ObjectStreamException
 - Etc.

Existen dos maneras de responder a una excepción. Si la excepción es chequeada, el programador está obligado a decidirse por alguna de estas dos formas. Si la excepción es no chequeada, el programador puede optar por no usar ninguna y simplemente ignorarla, o puede tratarla tal como se presenta a continuación:

1. Declarar la posibilidad de lanzamiento de excepción en la cabecera del método que posea el bloque de código. Veamos un ejemplo,


Archivo: Run.java

```
import javax.swing.JOptionPane;

public class Run {

    public static void main(String[] args) throws ArrayIndexOutOfBoundsException,
        NumberFormatException, ArithmeticException {
        JOptionPane.showMessageDialog(null, "Resultado: " +
            dividir(Double.parseDouble(args[0]), Double.parseDouble(args[1])));
    }
}
```

Palabra reservada para indicar que se lanzará una excepción

	INGENIERÍA EN INFORMÁTICA – PLAN 2003/2014 PROGRAMACIÓN DISTRIBUIDA Y COMPONENTES – 9º CUATRIMESTRE	
	APUNTE DE JAVA	VERSIÓN: 1.1 VIGENCIA: 03-05-2014

```

private static double dividir(double op1, double op2) {
    return op1 / op2;
}

```

2. Utilizar un manejador de excepciones, el cual es una porción de código que se va a encargar de tratar las posibles excepciones que se puedan generar. Los manejadores de excepciones se estructuran en tres bloques:
 - a. El bloque **try**: Encierra el conjunto de instrucciones susceptibles de lanzar una excepción. En el momento en que se produzca la excepción, se abandona el bloque try y, por lo tanto, las instrucciones que sigan al punto donde se produjo la excepción no serán ejecutadas. Cada bloque try debe tener asociado al menos un bloque catch.
 - b. El bloque **catch**: Por cada bloque try pueden declararse uno o varios bloques catch, cada uno de ellos capaz de tratar un tipo u otro de excepción. Para declarar el tipo de excepción que es capaz de tratar un bloque catch, se declara un objeto cuya clase es la clase de la excepción que se desea tratar o una de sus superclases.
 - c. El bloque **finally**: El bloque finally se utiliza para ejecutar un bloque de instrucciones sea cual sea la excepción que se produzca. Este bloque se ejecutará en cualquier caso, incluso si no se produce ninguna excepción. Sirve para no tener que repetir código en el bloque try y en los bloques catch.

Del ejemplo anterior,

Archivo: Run.java

```


import javax.swing.JOptionPane;

public class Run {

    public static void main(String[] args) {
        try {
            JOptionPane.showMessageDialog(null, "Resultado: " +
dividir(Double.parseDouble(args[0]), Double.parseDouble(args[1])));
        }
        catch(ArrayIndexOutOfBoundsException ex) {
            JOptionPane.showMessageDialog(null, "No se especificaron los
argumentos");
        }
        catch(NumberFormatException ex) {
            JOptionPane.showMessageDialog(null, "Se produjo un error al parsear los
argumentos. Detalle: " + ex.getMessage());
        }
    }

    private static double dividir(double op1, double op2) {
        try {
            return op1 / op2;
        }
        catch(ArithmeticException ex) {
            JOptionPane.showMessageDialog(null, "Se produjo un error al dividir.
Detalle: " + ex.getMessage());
            return 0;
        }
    }
}

```

	INGENIERÍA EN INFORMÁTICA – PLAN 2003/2014 PROGRAMACIÓN DISTRIBUIDA Y COMPONENTES – 9º CUATRIMESTRE	
	APUNTE DE JAVA	VERSIÓN: 1.1 VIGENCIA: 03-05-2014

```

    }
}
}

```

Si el bloque try llega a disparar una excepción del tipo `NumberFormatException`, automáticamente la JVM creará un objeto de esa clase y buscará el bloque catch que tenga un parámetro `NumberFormatException`. Si lo encuentra, pasará el objeto creado a ese bloque, y se ejecutará el código especificado en el bloque catch. Luego de ello, el programa seguirá ejecutando las instrucciones que se encuentran debajo del bloque catch. Como se ve, no necesariamente el programa termina si una excepción se produce y la misma es capturada. La decisión de terminarlo es del programador. El objeto generado por la JVM para representar la excepción dispone de una serie de métodos que permiten que el programador tenga mayor conocimiento del error producido. Uno de esos métodos es `getMessage()` (heredado desde `Throwable`) que retorna un `String` con una descripción del error que provocó la excepción.

Creación de Excepciones

Se pueden definir excepciones propias en Java, no hay por qué limitarse a las predefinidas, bastará con que la clase extienda de la clase `Exception` y proporcionar la funcionalidad extra que requiera el tratamiento de esa excepción. Dicha clase será una excepción "chequeada", y el compilador obligará a tratarla.

Veamos un ejemplo de declaración,

Archivo: `MyException.java`

```

public class MyException extends Exception {

    private static final long serialVersionUID = -9070240172452055818L;
    private String message;

    public MyException() { }

    public MyException(String message) {
        this.message = message;
    }

    @Override
    public String getMessage() {
        return this.message;
    }


}

```

Colección Garbage y Gestión de Memoria

En algunos lenguajes, como C++, se usa el operador *new* para alocar memoria y luego se usa el operador *delete* para liberarla cuando no se necesita más. Sin embargo, Java no tiene un operador delete.

En Java, hay que contar con un proceso ya construido llamado *Colección Garbage*. Este proceso es automático, aunque no se pueda predecir cuándo va a tener lugar. Java dispondrá de la memoria alocada que no tenga más referencias. Para que funcione la *Colección Garbage*, se puede poner una variable a *null* (aunque hacer esto tampoco permite predecir cuándo, si llega el caso, la *Colección Garbage* empezará a funcionar al ejecutar el programa).

	INGENIERÍA EN INFORMÁTICA – PLAN 2003/2014 PROGRAMACIÓN DISTRIBUIDA Y COMPONENTES – 9º CUATRIMESTRE	
	APUNTE DE JAVA	VERSIÓN: 1.1 VIGENCIA: 03-05-2014

Un punto fundamental, es evitar la referencia circular en las que un objeto hace referencia a otro y el segundo al primero. Cuando se liberan todas las referencias a estos objetos en un programa, cada uno todavía tiene una referencia interna al otro, lo que significa que la *Colección Garbage* no puede actuar en el otro objeto. Peor todavía es que, como no hay referencias externas al objeto, no se puede llegar a ese objeto para cambiar la situación. Ambos objetos estarán en memoria, consumiendo recursos, hasta que el programa finalice.

Cuando un objeto está dentro del proceso de *Garbage*, este recolector llama a un método del objeto llamado *finalize*, si existe. En este método se puede ejecutar el código de limpieza y, con frecuencia, es buena idea liberar cualquier referencia a otros objetos que tenga el objeto actual para eliminar la posibilidad de referencias circulares.

Dicho método se hereda de la clase `Object`, y es invocado antes de eliminar de memoria a un objeto desreferenciado.

```
protected void finalize() throws Throwable {
    System.out.println("Este objeto ya está disponible para la Colección Garbage");
}
```

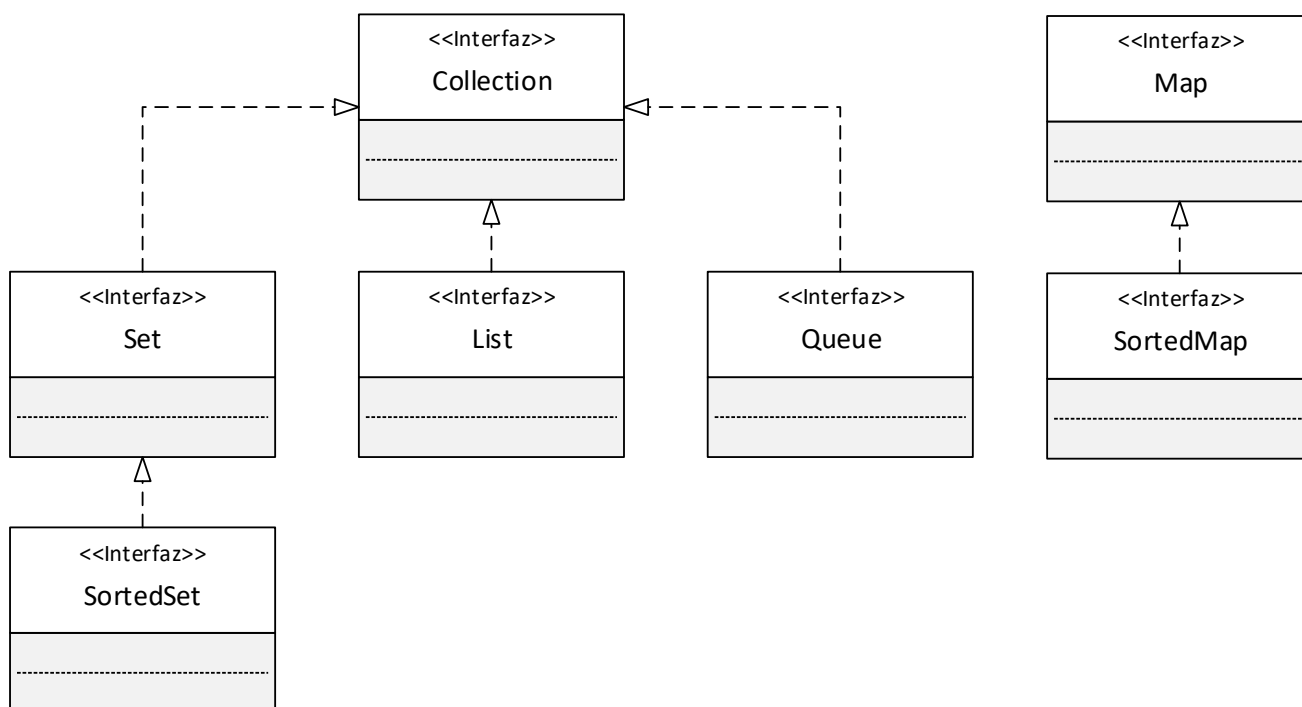
Colecciones

Una colección es un objeto que sirve para almacenar objetos, generalmente, del mismo tipo. Es un contenedor que almacena múltiples elementos en un única unidad.

Interfaces	Collection Map Queue List Set Etc.
Implementaciones	HashMap LinkedHashMap ArrayList LinkedList TreeSet Etc.
Algoritmos	Sorting Shuffling Searching

El Framework Java Collections forma parte del paquete **java.util**, por lo que se debe importar cada vez que se haga uso de alguna de sus colecciones. Veamos el propósito de alguna de sus interfaces:

- *Lists*: Lista de elementos.
- *Maps*: Pares de elementos con clave única.
- *Queues*: Elementos ordenados de acuerdo a la forma en que deben ser procesados.
- *Sets*: Conjunto de elementos únicos.



Generics

Los **generics** o tipos parametrizados nos permiten manejar las colecciones de forma segura con respecto a los tipos de datos que estas almacenan.

El tipo se señala a la derecha del nombre de la clase entre signos <,> y debe colocarse tanto en la declaración como en la instanciación de la colección antes de la llamada del constructor. Por ejemplo,

```
List<Articulo> list;
list = new LinkedList<Articulo>();
```

Convención de Nombres

- **E** → Element: Elemento de una colección
- **K** → Key: Elementos utilizados como claves
- **N** → Number
- **T** → Type: Se usa para métodos y clases con generics
- **V** → Value: Elemento usado como valores


Recorrido de colecciones

Iterator

Iterator<E> iterator(): Retorna un objeto iterador para recorrer la colección. Veamos un ejemplo,

Archivo: Run.java

```
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;
```

 UBP UNIVERSIDAD BLAS PASCAL	INGENIERÍA EN INFORMÁTICA – PLAN 2003/2014 PROGRAMACIÓN DISTRIBUIDA Y COMPONENTES – 9º CUATRIMESTRE	
	APUNTE DE JAVA	VERSIÓN: 1.1 VIGENCIA: 03-05-2014

```
public class Run {

    public static void main(String[] args) {
        List<Articulo> list;
        Iterator<Articulo> iterator;
        list = new LinkedList<Articulo>();

        list.add(new Lapicera("Roja", "Fino"));
        list.add(new Lapicera("Negra", "Grueso"));
        list.add(new Lapicera("Azul", "Fino"));

        iterator = list.iterator();
        while(iterator.hasNext()) {
            System.out.print(iterator.next().toString());
        }
    }

}
```

For Each

Útil para recorridos NO para modificaciones en la estructura de la colección. Implícitamente utiliza un iterador. Veamos el ejemplo anterior,

Archivo: Run.java

```
import java.util.LinkedList;
import java.util.List;

public class Run {

    public static void main(String[] args) {
        List<Articulo> list;
        list = new LinkedList<Articulo>();

        list.add(new Lapicera("Roja", "Fino"));
        list.add(new Lapicera("Negra", "Grueso"));
        list.add(new Lapicera("Azul", "Fino"));

        for(Articulo item : list) {
            System.out.print(item.toString());
        }
    }

}
```

Interfaz List

Son colecciones ordenadas, que pueden contener elementos duplicados (incluyendo el elemento null). Entre sus prestaciones permiten:

- Acceder a sus elementos de acuerdo a su posición numérica (índice) dentro de la lista.
- Buscar un elemento específico y retornar su índice.
- Realizar operaciones de rangos. Vistas de rango.

	INGENIERÍA EN INFORMÁTICA – PLAN 2003/2014 PROGRAMACIÓN DISTRIBUIDA Y COMPONENTES – 9º CUATRIMESTRE	
	APUNTE DE JAVA	VERSIÓN: 1.1 VIGENCIA: 03-05-2014

Finalmente, extienden el concepto de iterador para aprovechar la naturaleza de las listas secuenciales.

Sus implementaciones de propósito general son:

- **ArrayList:** Es la implementación de una lista “sobre” un arreglo. Intenta simular un arreglo *dinámico*, que aumenta o disminuye su tamaño cuando se agregan o eliminan elementos, respectivamente.
El dinamismo del arreglo es relativo, ya que lo que sucede en realidad es la creación de un nuevo arreglo en memoria, cargado con los datos del anterior y la reasignación a la referencia de la colección.
El tamaño por defecto es 10, por tal al agregar un elemento 11, se crea un nuevo objeto tal como se explicó anteriormente.
- **LinkedList:** Es la implementación de una lista basada en una lista doblemente enlazada.

Si comparamos ambas implementaciones, podemos concluir que un ArrayList es una estructura muy eficiente cuando se pretende tener acceso a los elementos, ya que está implementado sobre un arreglo. Mientras que un LinkedList será apropiado para realizar cambios en la estructura de la lista, todo lo contrario con un ArrayList que al redimensionar su arreglo interno si se cambia la estructura, pierde eficiencia.

Interfaz Set

Son colecciones que no permiten elementos duplicados. Si se intenta insertar un elemento que ya existía el Set se mantiene sin cambios.

Se considera que dos elementos son iguales si se cumple: Sean e1 y e2 elementos, e1.equals(2) == true

Representan el concepto matemático de conjunto.

Sus implementaciones de propósito general son:

- **TreeSet**
- **HashSet:** Es la implementación de un Set sobre una tabla Hash (HashMap). No garantiza el orden de los elementos.
- **LinkedHashSet**