PROGRAMACIÓN EN JAVA

Java Standard Edition (JSE)

Jose Luis Llorente
joseluis.llorenteperales@gmail.com

Contenido

Tema 1: Conceptos Programación orientada a objetos

Introducción

La programación orientada a Objetos básicamente define una serie de conceptos y técnicas de programación para representar acciones o cosas de la vida real basada en objetos.

A diferencia de otras formas de programación como por ejemplo la estructurada, con la POO trabajamos de manera distinta vinculando diferentes conceptos tales como clases, objetos, métodos, propiedades, estados, herencia, encapsulación entre otros.

La Metodología Orientada a Objetos, aporta un enfoque diferente para la construcción del software.

- Este enfoque debe ser capaz de manejar tanto sistemas grandes y complejos, como sistemas pequeños.
- Debe crear sistemas flexibles, mantenibles y capaces de evolucionar a los cambios inherentes a la realidad que representan.

En nuestro mundo distinguimos objetos individuales, u objetos complejos formados por otros objetos, creando una imagen mental de cada uno de ellos, de modo que podemos reconocer un objeto de nuevo después de haberlo visto una vez.

Cuando observamos un objeto del mundo real (plancha, persona, factura, cliente, ordenador, bicicleta…) normalmente nos fijamos en sus propiedades, por ejemplo, una persona puede ser alta o baja, joven o adulta, se llama Tomás o Esteban, haber nacido en Madrid o en Guadalajara, etc.

Además de las características de cada objeto, debo saber o reconocer su comportamiento. Y también observamos que determinadas operaciones de un objeto cambian el valor de sus atributos.

Por tanto, podemos decir que la Metodología Orientada a Objetos trata de trasladar el mundo real al mundo Informático.

Identifica todos y cada uno de los objetos que le son necesarios para convertir el Sistema de Información del cliente (conjunto de requisitos y reglas de negocio) en el Sistema Informático.

Clases

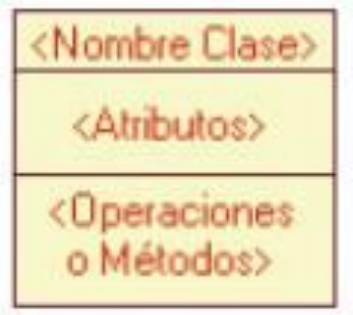
Las clases son uno de los principales componentes de un lenguaje de programación, pues en ellas ocurren todos los procesos lógicos requeridos para un sistema, en si podemos definirlas como estructuras que representan objetos del mundo real, tomando como objetos a personas, lugares o cosas, en general las clases poseen propiedades, comportamientos y relaciones con otras clases del sistema.

Todos los objetos que comparten las mismas características y comportamiento, para definirles, se utiliza el término de **CLASE**.

Por tanto, tendremos las clases Persona, Cliente, Proveedor, Suministrador, Ordenador…

Cuando describimos la Clase Persona, estamos identificando a todos y cada uno de los objetos Persona que se creen de esta clase, y que comparten los mismos nombres de atributos, y las mismas operaciones.

Una clase se compone por tres partes fundamentales:



Nombre: Contiene el Nombre de la Clase.
Atributos: Representan las propiedades que caracterizan la clase.
Métodos: Representan el comportamiento u operaciones, la forma como interactúa la clase con su entorno.

Características principales de la POO

Para todos los lenguajes orientados a objetos, su marco de referencia conceptual es el modelo de objetos (que analizaremos posteriormente).

Hay cuatro elementos fundamentales en este modelo:

- Abstracción
- Encapsulamiento
- Polimorfismo
- Herencia

Abstracción

La abstracción es el camino por el cual el ser humano combate la complejidad. Es un mecanismo, quizá innato, por el que tendemos a hacer simple aquello que por su naturaleza es complejo.

Es decir, cuando vemos un objeto, solo nos fijamos en aquellas propiedades y comportamiento que nos son útiles para el fin que perseguimos, eliminando aquellos otros que, de momento, son irrelevantes o nos distraen del problema.

Mediante el mecanismo de abstracción nos aseguraremos de que en nuestra aplicación los elementos que vamos a utilizar solo contengan aquellas características y propiedades útiles para establecer las funcionalidades establecidas.

Por ejemplo, en una aplicación bancaria para representar a un cliente serán importantes su nombre, apellidos, DNI o edad, pero no serán útiles su color de pelo o como come o duerme.

Encapsulamiento

Cada clase es una estructura compleja en cuyo interior hay datos y comportamiento, todos ellos relacionados entre sí, como si estuvieran encerrados conjuntamente en una cápsula.

Cuando se crean objetos de una clase determinada, éstos, son inaccesibles, e impiden que otros objetos, los usuarios, o incluso los programadores conozcan cómo está distribuida la información o qué información hay disponible.

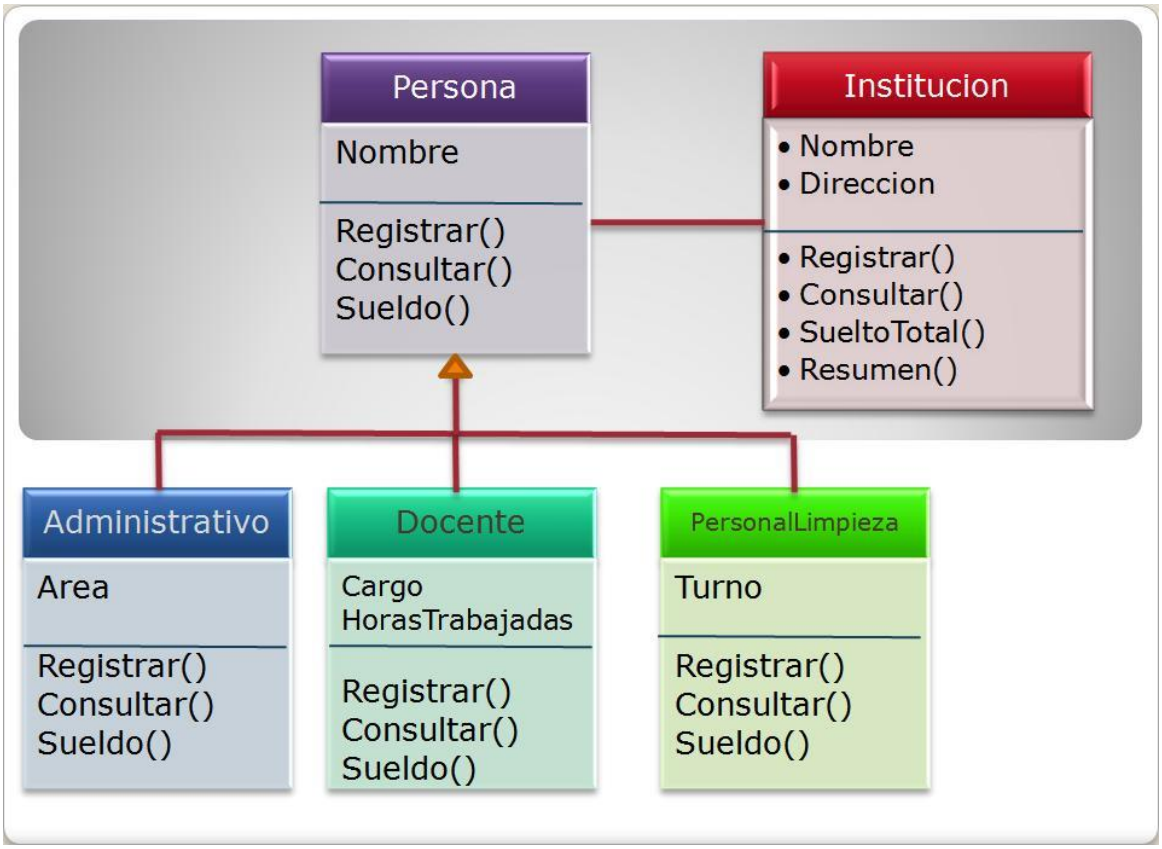
Esta propiedad de los objetos se denomina ocultación de la información. Se aplica para que nadie pueda alterar el comportamiento que tiene un programa, únicamente solicitas que se ejecute una funcionalidad y te devuelva el resultado, el como se hace no te debe afectar.

Por ejemplo, la acción de cambiar un canal en el televisor, tu quieres que cuando pulses un botón de un canal la televisión cambie de canal no te importa como lo hace, tu lo que quieres es ver el resultado. Tampoco puedes cambiar el comportamiento ya que puedes provocar que deje de funcionar.

Herencia

Otra característica del ser humano para simplificar lo complejo, es jerarquizar las cosas. Es decir, clasificar y ordenar las abstracciones, usando la generalización, es decir empezando por lo general y terminando en fases sucesivas por la especialización.

La jerarquización consiste en crear estructuras piramidales, es decir con un nodo arriba de la pirámide, que identifica a una clase que posee características y/o funcionalidades generales a todos los subtipos que hay por debajo de la estructura.



Básicamente es el mecanismo por el cual una clase determinada denomina clase derivada, sub-clase o clase hija (en el ejemplo, Administrativo, Docente y PersonalLimpieza, son sub-clases de Persona) hereda la estructura y comportamiento de una clase base, clase padre o superclase (la clase Persona es la superclase de la jerarquía).

Una subclase puede heredar la estructura de datos y los métodos, o algunos de los métodos, de su superclase.

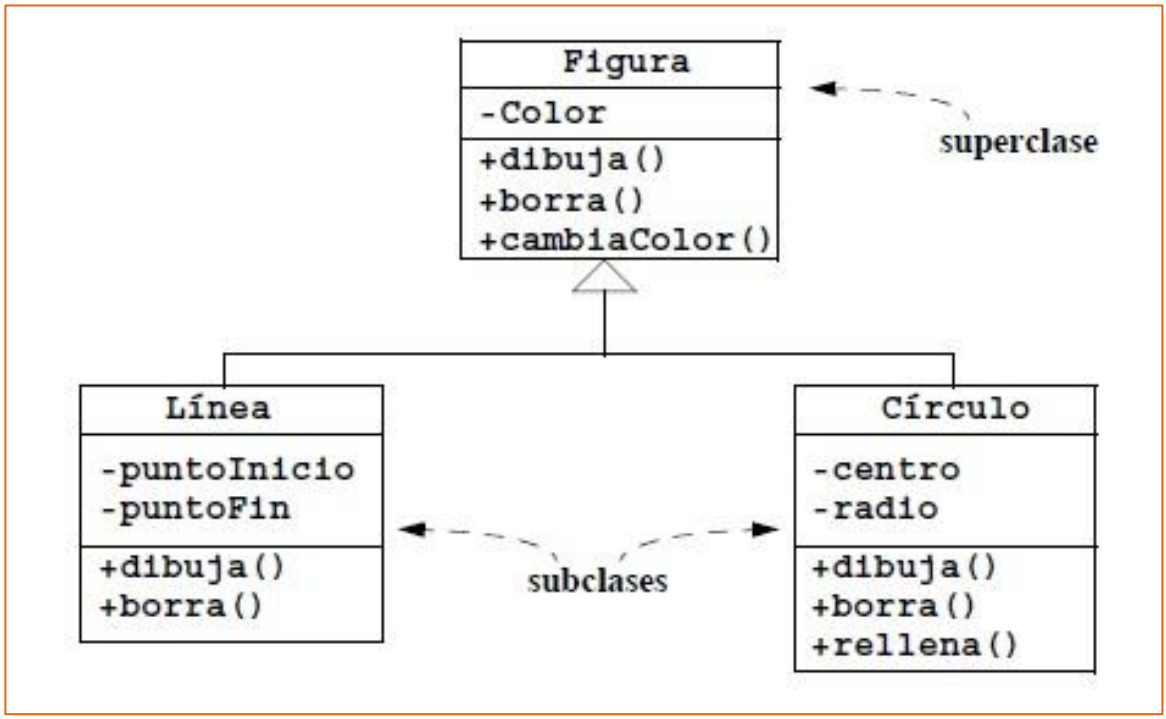
También tiene sus métodos e incluso tipos de datos propios. La herencia de la estructura de datos permite la reutilización de la estructura.

Polimorfismo

Una de las características fundamentales de la POO es el polimorfismo, usado principalmente en clases heredadas, que no es otra cosa que la posibilidad de construir varios métodos con el mismo nombre, pero con relación a la clase a la que pertenece cada uno, con comportamientos diferentes.

Esto conlleva la habilidad de enviar un mismo mensaje a objetos de clases diferentes, situados en una jerarquía. Estos objetos recibirían el mismo mensaje global, pero responderían a él de formas diferentes.

Por ejemplo, para las figuras geométricas podríamos hacer que todas tengan el método "dibuja" pero no todas las figuras geométricas se dibujaran de la misma forma.



Tema 2: Introducción a Java

Introducción

Java es un lenguaje de programación de alto nivel orientado a objetos, desarrollado por James Gosling en 1995. Se popularizó a partir del lanzamiento de su primera versión comercial de amplia difusión, la JDK 1.0 en 1996. Actualmente es uno de los lenguajes más usados para la programación en todo el mundo.

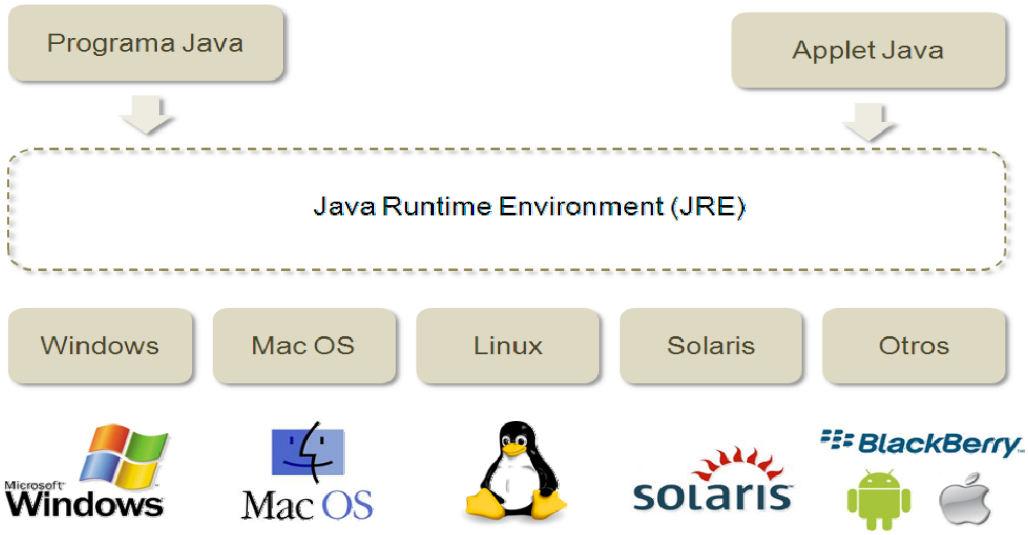
El lenguaje en sí mismo toma mucha de su sintaxis de C, Cobol y Visual Basic, pero tiene un modelo de objetos más simple y elimina herramientas de bajo nivel, que suelen inducir a muchos errores, como la manipulación directa de punteros o memoria. La memoria es gestionada mediante un recolector de basura.

Lenguaje interpretado

Java es un lenguaje muy valorado porque los programas Java se pueden ejecutar en diversas plataformas con sistemas operativos como Windows, Mac OS, Linux o Solaris.

James Gosling, el director del equipo de trabajo encargado de desarrollar Java, hizo realidad la promesa de un lenguaje independiente de la plataforma. Se buscaba diseñar un lenguaje que permitiera programar una aplicación una sola vez que luego pudiera ejecutarse en distintas máquinas y sistemas operativos.

Para conseguir la portabilidad de los programas Java se utiliza un entorno de ejecución para los programas compilados. Este entorno se denomina Java Runtime Environment (JRE). Es gratuito y está disponible para los principales sistemas operativos. Esto asegura que el mismo programa Java pueda ejecutarse en Windows, Mac, OS, Linux o Solaris.



Usos

Java es un lenguaje útil para solventar casi todo tipo de funcionalidades y problemas.

Podemos citar como funcionalidades de Java varias:

1. **Aplicaciones "cliente":** son las que se ejecutan en un solo ordenador (por ejemplo el portátil de tu casa) sin necesidad de conectarse a otra máquina. Pueden servirte por ejemplo para realizar cálculos o gestionar datos.

2. **Aplicaciones "cliente/servidor":** son programas que necesitan conectarse a otra máquina (por ejemplo un servidor de datos) para pedirle algún servicio de forma más o menos continua, como podría ser el uso de una base de datos. Pueden servir por ejemplo para el teletrabajo: trabajar desde casa pero conectados a un ordenador de una empresa.

3. **"Aplicaciones web"**, que son programas Java que se ejecutan en un servidor de páginas web. Estas aplicaciones reciben "solicitudes" desde un ordenador y envían al navegador (Internet Explorer, Firefox, Safari, etc.) que actúa como su cliente páginas de respuesta en HTML.

4. **"Aplicaciones móviles"** gracias a la máquina virtual de los sistemas Android, primeramente, Dalvik y desde la versión 4.4.2 o Kitkat denominada ART.

Éstos son sólo algunos ejemplos de todo el potencial que hay detrás de Java como lenguaje para aprender y obtener muchos beneficios con su uso. Obviamente por determinados términos empleados (cliente, cliente/servidor, base de datos, HTML…), te darás cuenta de que el lenguaje Java tiene mucha potencialidad, pero también de que su conocimiento a fondo requeriría mucho tiempo.

VERSIONES Y DISTRIBUCIONES DE JAVA

Distribuciones

Existen 3 distribuciones principales de Java con ciertos aspectos comunes y ciertos aspectos divergentes.

- **Java SE**: la conocida como ***Standard Edition*** es la edición más difundida de la plataforma Java. Incorpora los elementos necesarios para crear **aplicaciones de escritorio** con o sin interfaz gráfica de usuario, acceso al sistema de archivos, comunicación a través de redes, concurrencia y otros servicios básicos.
    - **JavaFX**: originalmente JavaFX era una alternativa a Java SE para el **desarrollo de proyectos de tipo RIA** (***Rich Internet Applications***), con un núcleo más ligero y fácil de distribuir, capacidad de **aceleración 3D** aprovechando la GPU, servicios avanzados para producción de gráficos y animaciones, y un mecanismo simplificado para el diseño de interfaces de usuario. JavaFX forma parte de Java SE desde la versión 7 de dicha edición de la plataforma.
- **Java EE**: es la ***Enterprise Edition*** de la plataforma Java, dirigida al desarrollo de soluciones software que se ejecutarán en un **servidor de aplicaciones**. A las capacidades de Java SE, la edición EE agrega servicios para gestionar la persistencia de objetos en bases de datos, hacer posible la invocación remota de métodos, crear aplicaciones con interfaz de usuario web, etc.
- **Java ME**: esta edición de la plataforma, Micro Edition, está enfocada a la creación de programas que se ejecutarán en **sistemas con recursos limitados**, tales como teléfonos móviles, electrodomésticos y dispositivos de domótica o equipos para entornos empotrados como la Rasperry Pi y similares.

Nota: Java SE, EE, ME, la JVM y otras partes de Java son en realidad especificaciones abstractas de los servicios y modos de funcionamiento de los distintos elementos de la plataforma. Encontraremos múltiples implementaciones de dichas especificaciones. Por ejemplo, HotSpot es la implementación de Oracle de la JVM, existiendo implementaciones alternativas de esa misma especificación como la de OpenJDK o la de IBM.

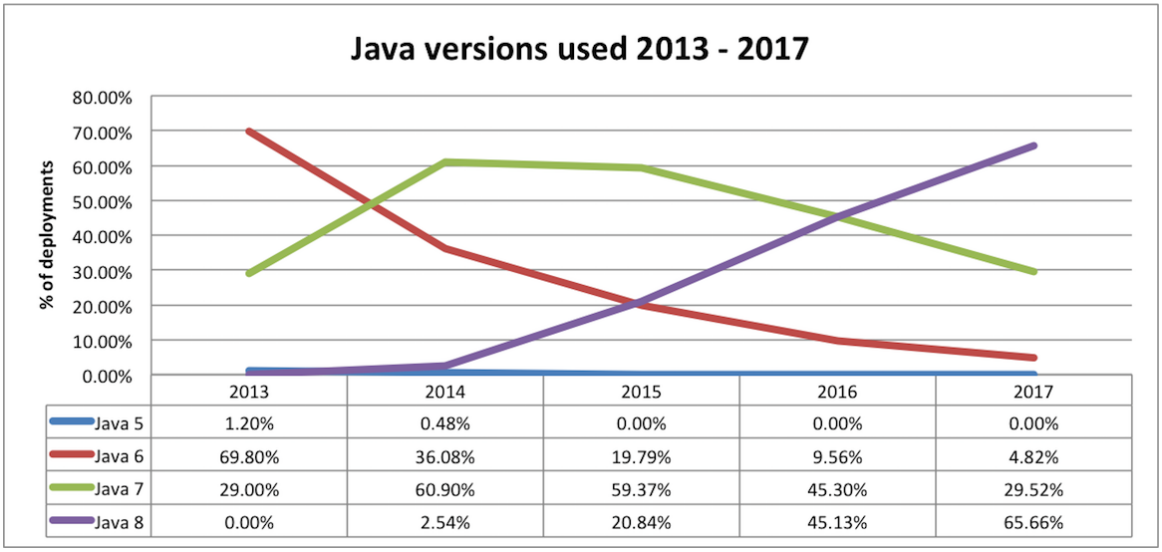
Versiones

Java, como la mayoría de los lenguajes, ha sufrido cambios a lo largo de su historia. Además, en cada momento han coexistido distintas versiones o distribuciones de Java con distintos fines.

- JDK 1.0 (23 de enero de 1996).
- JDK 1.1 (9 de febrero de 1997).
- J2SE 1.2 (8 de diciembre de 1998).
- J2SE 1.3 (8 de mayo de 2000).
- J2SE 1.4 (6 de febrero de 2002)

- J2SE 5.0 (30 de septiembre de 2004).
- Java SE 6 (11 de diciembre de 2006).
- Java SE 7 (Jul-2011)
- Java SE 8 (2013)
- Java SE 9 (Oct -2017)

En el año 2009 El grupo estadounidense de software **Oracle** adquirio Sun Microsystems, haciendo cargo así de Java.



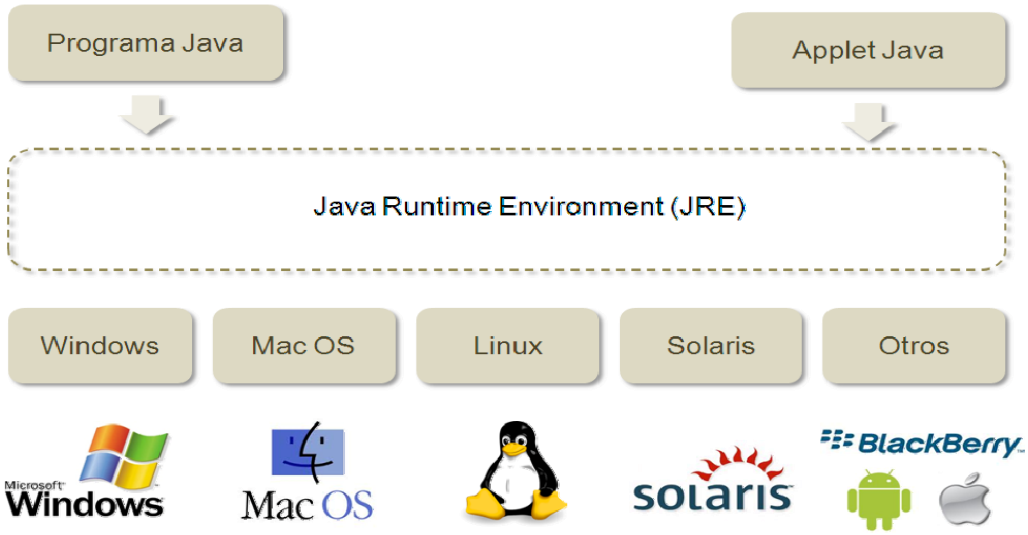
	2013	2014	2015	2016	2017
Java 5	1.20%	0.48%	0.00%	0.00%	0.00%
Java 6	69.80%	36.08%	19.79%	9.56%	4.82%
Java 7	29.00%	60.90%	59.37%	45.30%	29.52%
Java 8	0.00%	2.54%	20.84%	45.13%	65.66%

Fuente (https://plumbr.io/blog/java/java-version-and-vendor-data-analyzed-2017-edition)

La Plataforma Java

Los programas Java se compilan a un lenguaje intermedio, denominado **Bytecode**. Este código es interpretado por la máquina virtual de Java del entorno de ejecución (JRE) y así se consigue la portabilidad en distintas plataformas.

El JRE es una pieza intermedia entre el código Bytecode y los distintos sistemas operativos existentes en el mercado. Un programa Java compilado en Bytecode se puede ejecutar en sistemas operativos como Windows, Linux, Mac Os, Solaris, BlackBerry OS, iOs o Android utilizando el entorno de ejecución de Java (JRE) apropiado.



Una de las características más importantes de los lenguajes de programación modernos es la **portabilidad**. Un programa es portable cuando es independiente de la plataforma y puede ejecutarse en cualquier sistema operativo y dispositivo físico. Los programas Java son portables porque se ejecutan en cualquier plataforma. Sucede algo parecido con las fotografías o los ficheros PDF. Las fotografías con formato JPEG son portables porque un archivo JPEG lo podemos visualizar con distintos visores de fotos y en dispositivos como ordenadores, tabletas o teléfonos

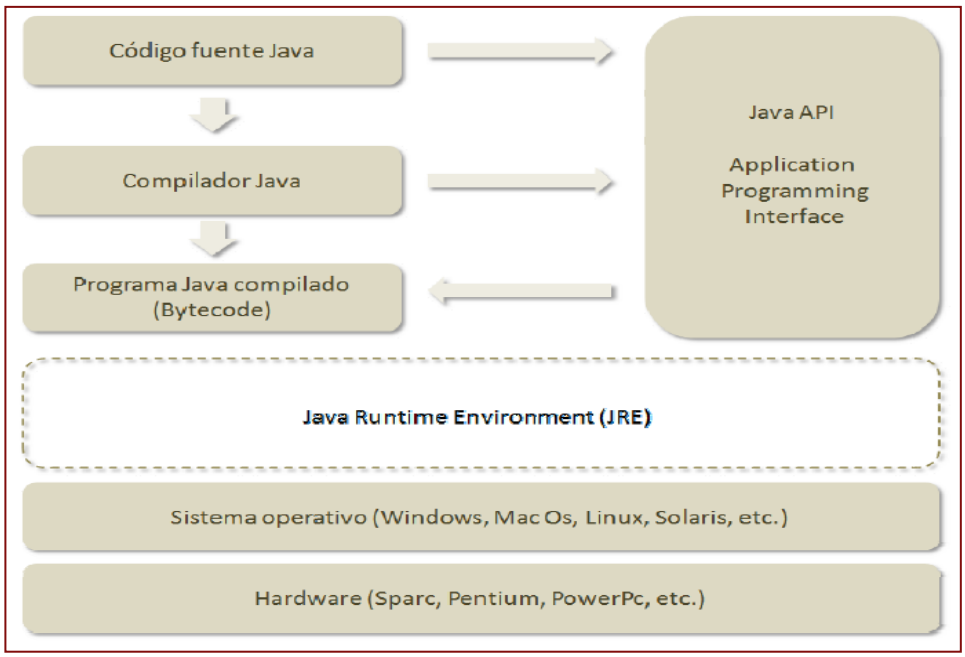
La **portabilidad de Java** ha contribuido a que muchas empresas hayan desarrollado sus sistemas de comercio electrónico y sus sistemas de información en Internet con Java. El proceso de desarrollo y de mantenimiento de los sistemas resulta menos costoso y las aplicaciones son compatibles con distintos sistemas operativos.

La evolución del lenguaje de programación Java ha sido muy rápida. La **plataforma de desarrollo de Java, denominada Java Development Kit (JDK),** se ha ido ampliando y cada vez incorpora a un número mayor de programadores en todo el mundo. En realidad, Java no solo es un lenguaje de programación. Java es un lenguaje, una plataforma de desarrollo, un entorno de ejecución y un conjunto de librerías para desarrollo de programas sofisticados.

Las librerías para desarrollo se denominan Java Application Programming Interface (Java API).

El siguiente esquema muestra los elementos de la plataforma Java, desde el código fuente, el compilador, el API de Java, los programas compilados en Bytecode y el entorno de ejecución de Java. Este entorno de ejecución (JRE) y la máquina virtual (JVM) permiten

que un programa compilado Java se ejecute en distintos sistemas operativos.



El proceso de edición y compilación

En Java, al igual que en otros lenguajes de programación, se sigue el siguiente proceso: edición del código fuente, compilación y ejecución. Los programas Java se desarrollan y se compilan para obtener un código denominado Bytecode que es interpretado por una máquina virtual de Java (Java Virtual Machine).

El primer concepto que abordar es el de **compilación**. **"Compilar"** significa traducir el código escrito en "Lenguaje entendible por humanos" (por ejemplo, Java, C, Pascal, Fortran), a un código en "Lenguaje Máquina", que entienden las máquinas, pero no entendible por nosotros.

Se hace esto porque a los humanos nos resultaría casi imposible trabajar directamente con el lenguaje de los ordenadores. Es por eso por lo que usamos un lenguaje más asequible para nosotros (en nuestro caso Java) y luego empleamos un traductor (compilador).

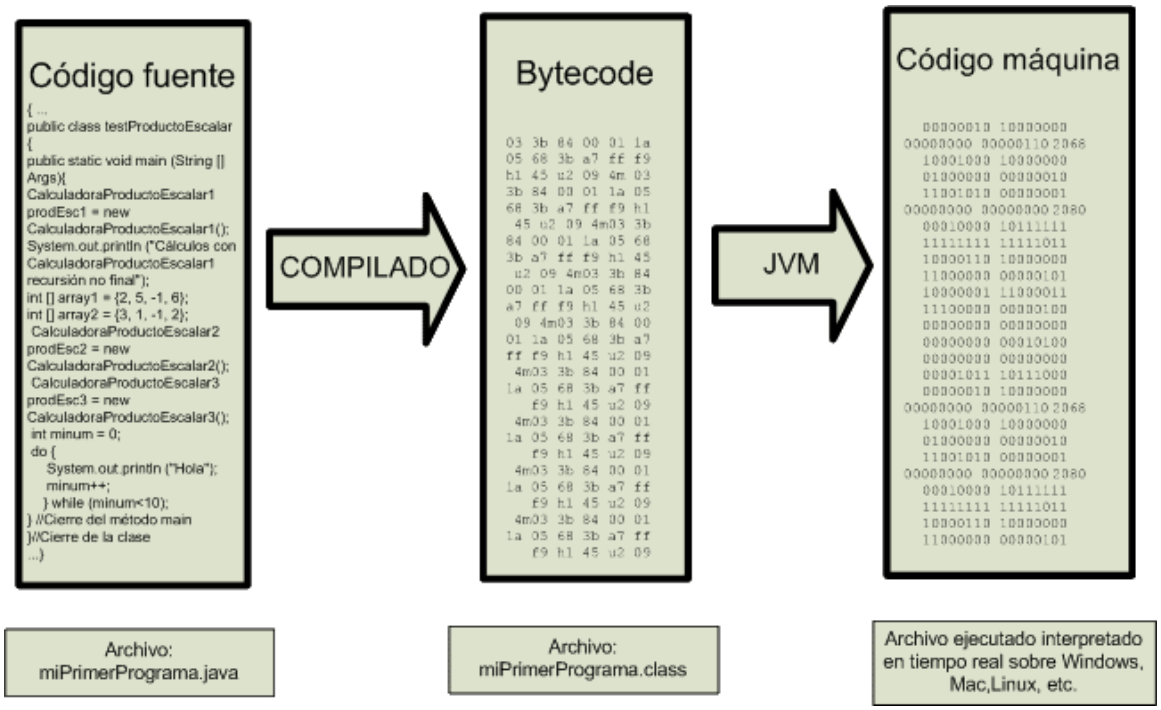
La creación de programas en muchos lenguajes se basa en el proceso:

escribir código fuente → compilar → obtener programa ejecutable.

La compilación se realiza con el compilador Java javac o utilizando un entorno integrado de desarrollo. Durante el proceso de compilación

se verifica que el código fuente cumple la definición léxica, sintáctica y semántica de Java. Esto significa que el compilador comprueba que el código fuente se compone de palabras válidas en Java y que los comandos Java tienen la forma sintáctica correcta. Si durante el proceso de compilación el compilador detecta los errores que ha cometido el programador y le informa de los problemas que ha encontrado para que pueda corregirlos.

Si durante la compilación no se detectan errores, se genera un fichero de tipo class en Bytecode. Una vez finalizado el proceso de compilación se puede ejecutar el programa. Para esto, es necesario que la máquina virtual de Java interprete el código Bytecode y ejecute la aplicación.



Esto permite que Java pueda ejecutarse en una máquina con el Sistema Operativo Unix, Windows, Linux o cualquier otro, porque en realidad no va a ejecutarse en ninguno de los sistemas operativos, sino en su propia máquina virtual que se instala cuando se instala Java. El precio a pagar o desventaja de este esquema es que todo ordenador que quiera correr una aplicación Java ha de tener instalado Java con su máquina virtual.

Tema 3: Instalacion y configuración

Instalacion

El Java Development Kit (JDK por sus siglas en inglés) se trata de un conjunto de herramientas que permiten desarrollar programas en lenguaje Java. Existen versiones del JDK que sirven para prácticamente todos los Sistemas Operativos y existen también distintos programas comerciales. Java es un lenguaje de programación y una plataforma informática comercializada. Hay muchas aplicaciones y sitios web que no funcionarán a menos que tenga Java instalado. Cada día se crean más aplicaciones con este lenguaje. Esto debido a que Java es rápido, seguro y fiable. Desde portátiles hasta centros de datos, desde consolas para juegos hasta súper computadoras, desde teléfonos móviles hasta Internet, Java está en todas partes.

Para instalarlo deberás dirigirte al siguiente enlace. Una vez allí deberás dar clic en la pestaña descargas. Entonces fíjate más abajo donde verás el botón Download debajo de donde dice JDK. Hazle clic a dicho botón como se muestra en la imagen.

Visión de conjunto | **descargas** | Documentación | Comunidad | tecnologías | Formación

Java SE Descargas

Plataforma Java (JDK) 8u121

NetBeans con JDK 8

Java SE

8u121 Java SE

Java SE 8u121 incluye importantes mejoras de seguridad. Oracle recomienda encarecidamente que todos los Java SE 8 usuarios actualizar a esta versión.
Aprende más

cambio planificado importante para los JAR-MD5 firmado

A partir de las notas de actualización crítica de abril, previstas del 18 de abril de 2017, todas las versiones de JRE tratarán JAR firmados con MD5 como sin signo. Obtenga más información y ver las instrucciones de prueba.
Para obtener más información sobre el soporte algoritmo criptográfico, por favor, compruebe el JRE y JDK Crypto Hoja de Ruta .

- Instrucciones de instalación
- Notas de la versión
- Licencia de Oracle
- Java SE Productos

JDK
DOWNLOAD

JRE del

Luego seleccionarás la opción que dice Aceptar el acuerdo de licencia en la parte superior de la ventana. Debajo verás una lista de enlaces justo al lado de los sistemas operativos. En nuestro caso debes buscar donde dice 64 bits de windows. Seguido dale clic al enlace que está a su lado, como muestra la Imagen #2.

Elegiremos la versión de java que corresponda a nuestro Sistema Operativo para comenzar la descarga. Antes tendremos que aceptar los terminos de la licencia.

8u121 Java SE Development Kit

Debe aceptar el Acuerdo de licencia del código binario de Oracle para Java SE para descargar este software.

○ **Aceptar el acuerdo de licencia** ◉ **Negar acuerdo de licencia**

Producto / Descripción del archivo	Tamaño del archivo	Descargar
Linux ARM 32 Float duro ABI	77.86 MB	jdk-8u121-linux-ARM32-VFP-hflt.tar.gz
Linux brazo 64 del flotador duro ABI	74.83 MB	jdk-8u121-linux-arm64-VFP-hflt.tar.gz
Linux x86	162.41 MB	jdk-8u121-linux-i586.rpm
Linux x86	177,13 MB	jdk-8u121-linux-i586.tar.gz
Linux x64	159.96 MB	jdk-8u121-linux-x64.rpm
Linux x64	174,76 MB	jdk-8u121-linux-x64.tar.gz
Mac OS X	223.21 MB	jdk-8u121-macosx-x64.dmg
Solaris SPARC de 64 bits	139.64 MB	jdk-8u121-solaris-sparcv9.tar.Z
Solaris SPARC de 64 bits	99.07 MB	jdk-8u121-solaris-sparcv9.tar.gz
Solaris x64	140.42 MB	jdk-8u121-solaris-x64.tar.Z
Solaris x64	96.9 MB	jdk-8u121-solaris-x64.tar.gz
x86 de windows	189.36 MB	jdk-8u121-windows-i586.exe
64 bits de windows	195.51 MB	jdk-8u121-windows-x64.exe

Imagen #2

Al darle clic al enlace de descarga de abrirá una ventana con el Explorador de archivos. Escogerás la carpeta donde deseas que se descargue el instalador y darás clic en el botón **Guardar**

Ahora hemos de ejecutar el archivo exe descargado y seguir los pasos.

En breves instantes comenzará la descarga así que espera pacientemente a los que termina. Una vez terminada la descarga, abrirás el archivo. Al hacerlo verás la ventana de bienvenida. Para comenzar la instalación debes dar clic en el botón Next, como se muestra la imagen #3.

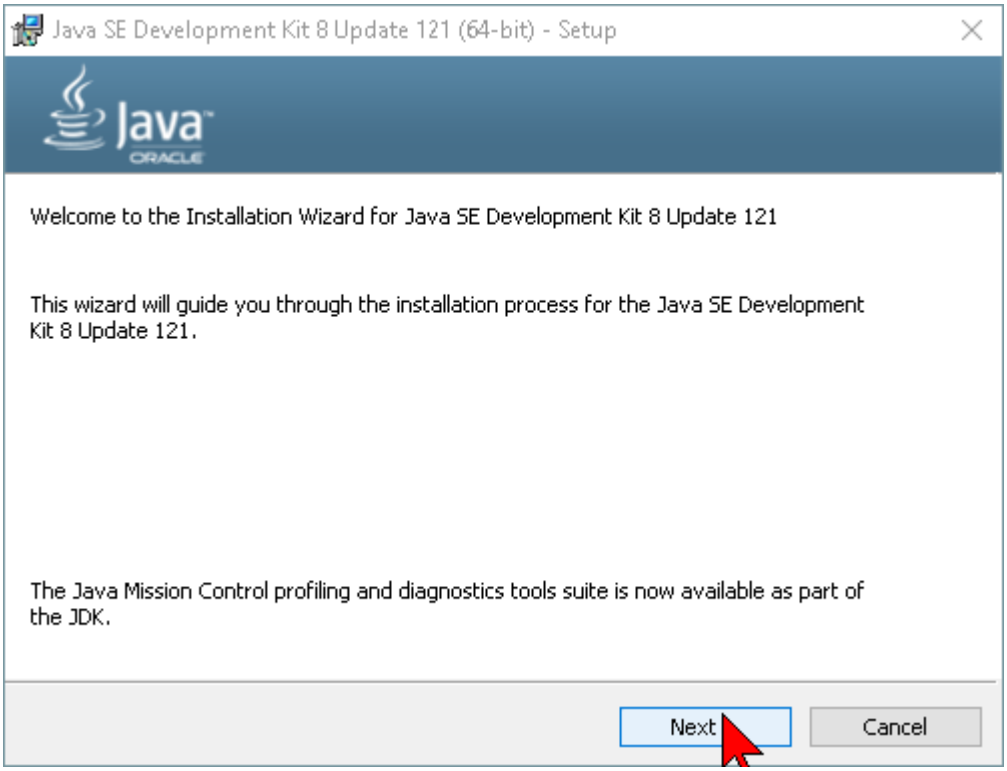


Imagen #3

Luego debes continuar dando clic en los botones **Next** como se ve en la imagen *#4*.

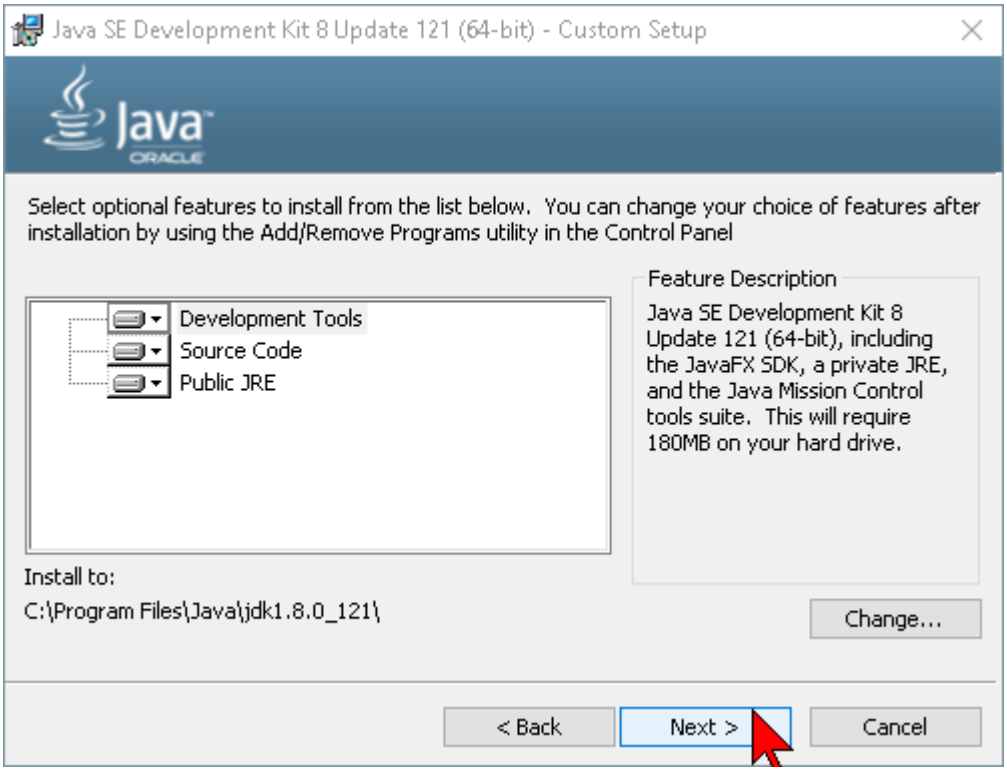


Imagen *#4*

Verás que se comienza a remover algunos archivos, nada de qué preocuparse. El ejemplo en la imagen #6 te muestra el progreso de la instalación.

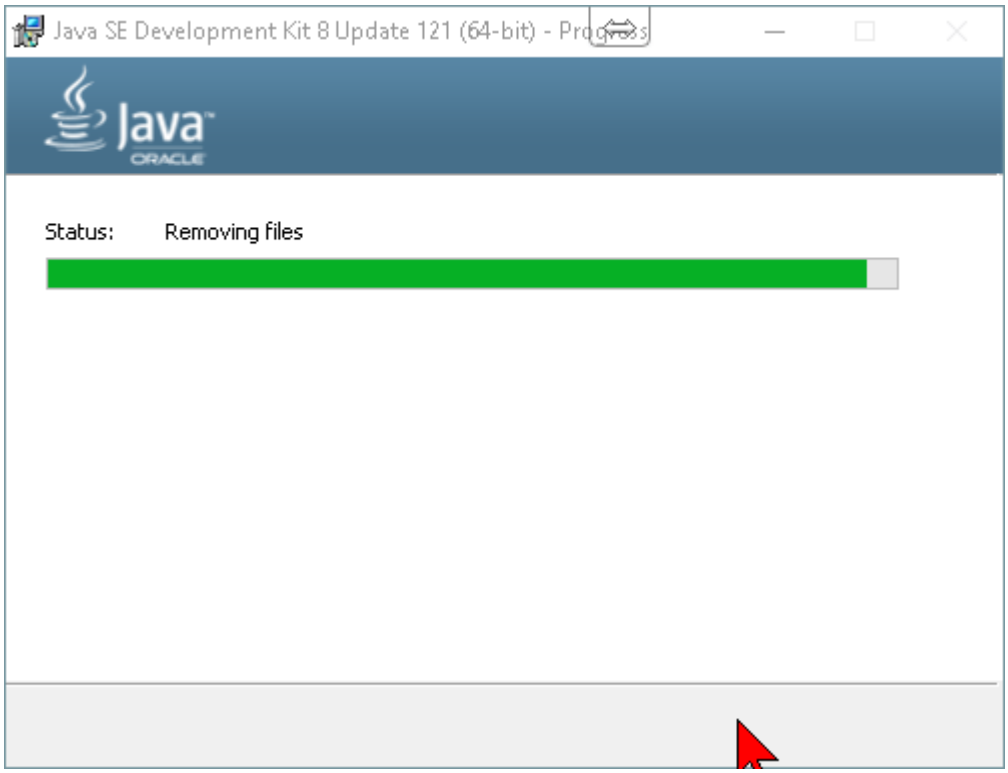


Imagen *#5*

El Java JDK incluye también el Java Runtime Environment (JRE por sus siglas en inglés). Éste es el programa que se usa para ejecutar aplicaciones escritas en el lenguaje Java. En otras palabras, con el JDK las desarrollas, con el JRE las ejecutas. Entonces también debes instalarlo y por ello te saldrá una ventana preguntándote donde quieres instalarlo.

Puedes ver esta ventana en la Imagen #6. Es recomendable que dejes la carpeta por defecto para que funcionen bien el JDK con el JRE. Después de seleccionar la carpeta donde deseas que se instale el JRE dale clic al botón **Siguiente**. El ejemplo está en la imagen #6.

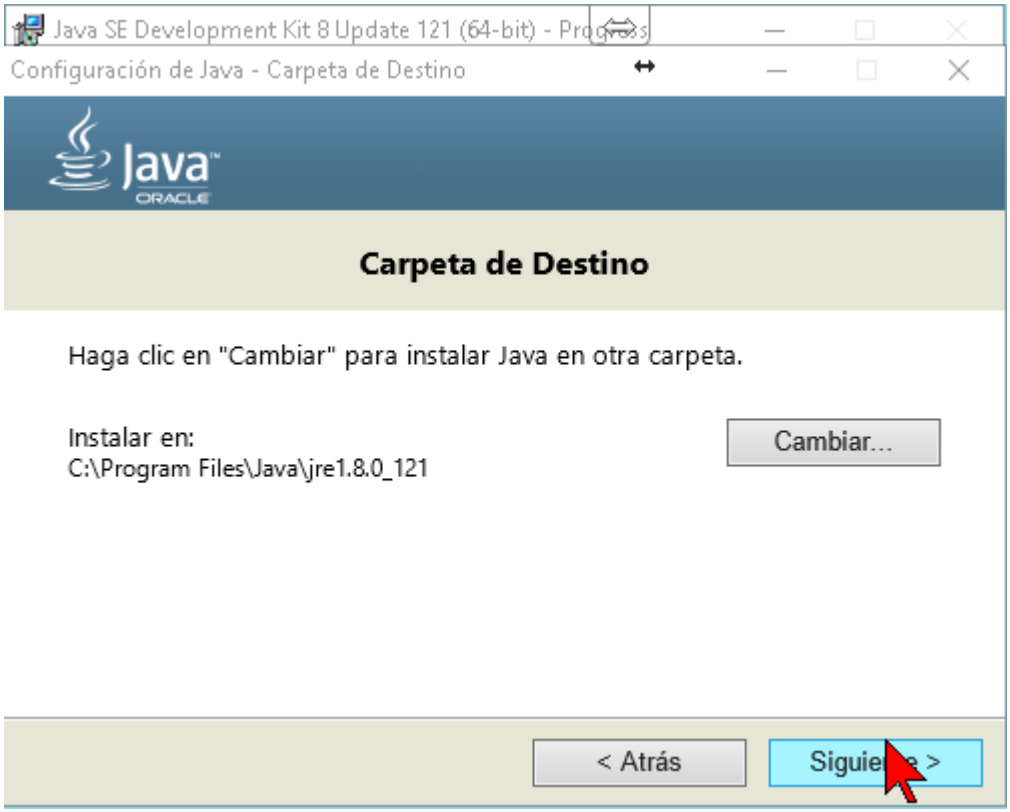


Imagen #6

Automáticamente comenzará la instalación del Java JRE también. Verás el progreso de la instalación en una ventana similar a la que se muestra en la imagen #7.

Imagen #7

Verás que la instalación está instalada correctamente cuando veas la ventana como la que se muestra a continuación. Para terminar con la misma dar clic en el botón **Close** como está en el ejemplo de la imagen *#8*.

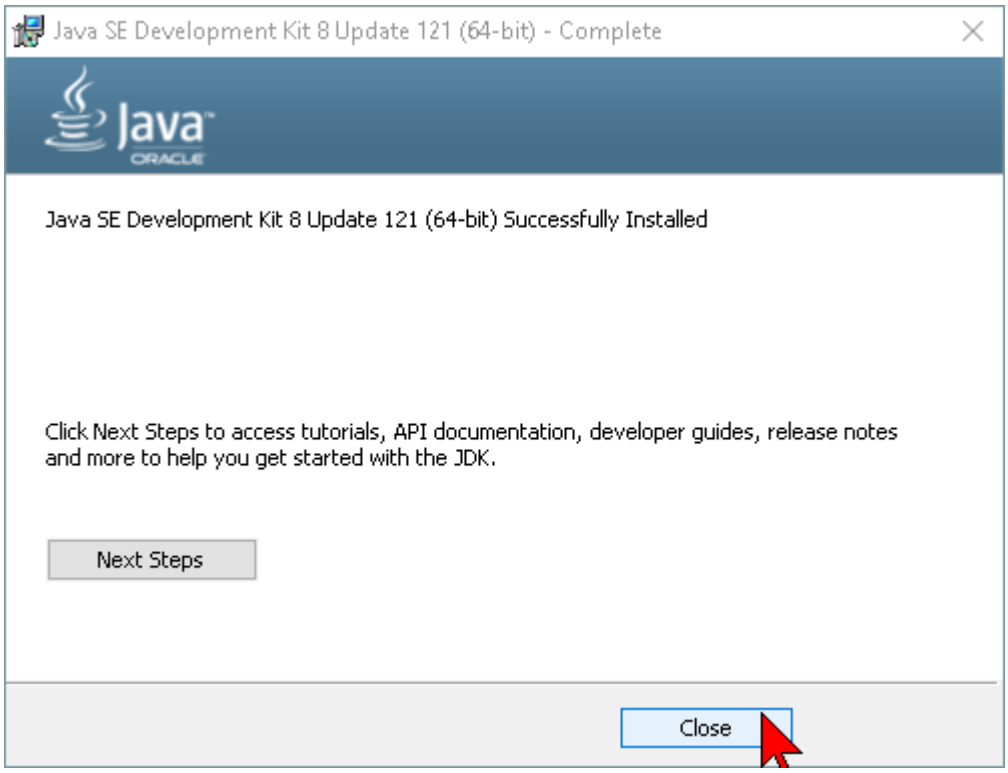
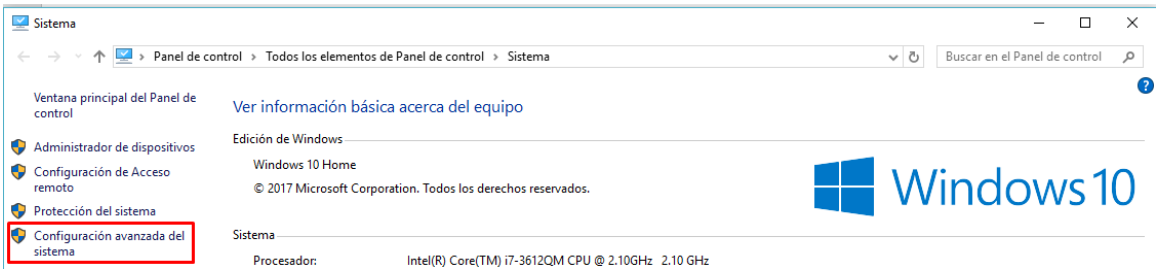


Imagen *#8*

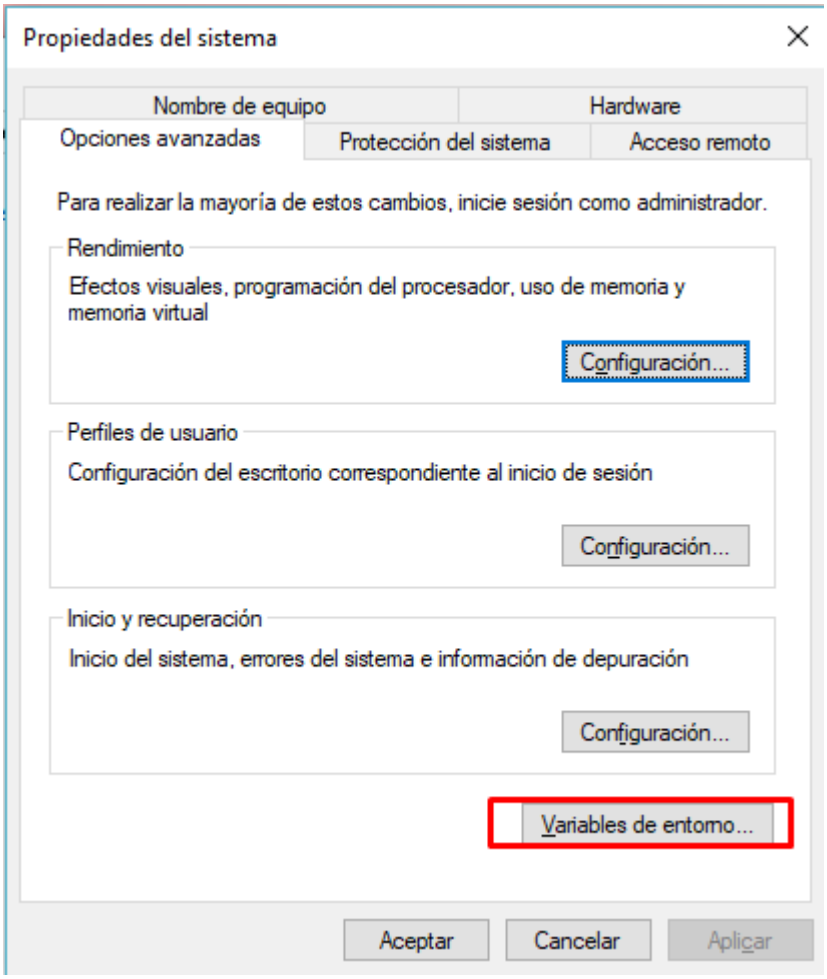
Configuración de variables de entorno

Ahora vamos a configurar las variables de entorno necesarias para trabajar, para ello seguimos estos pasos:

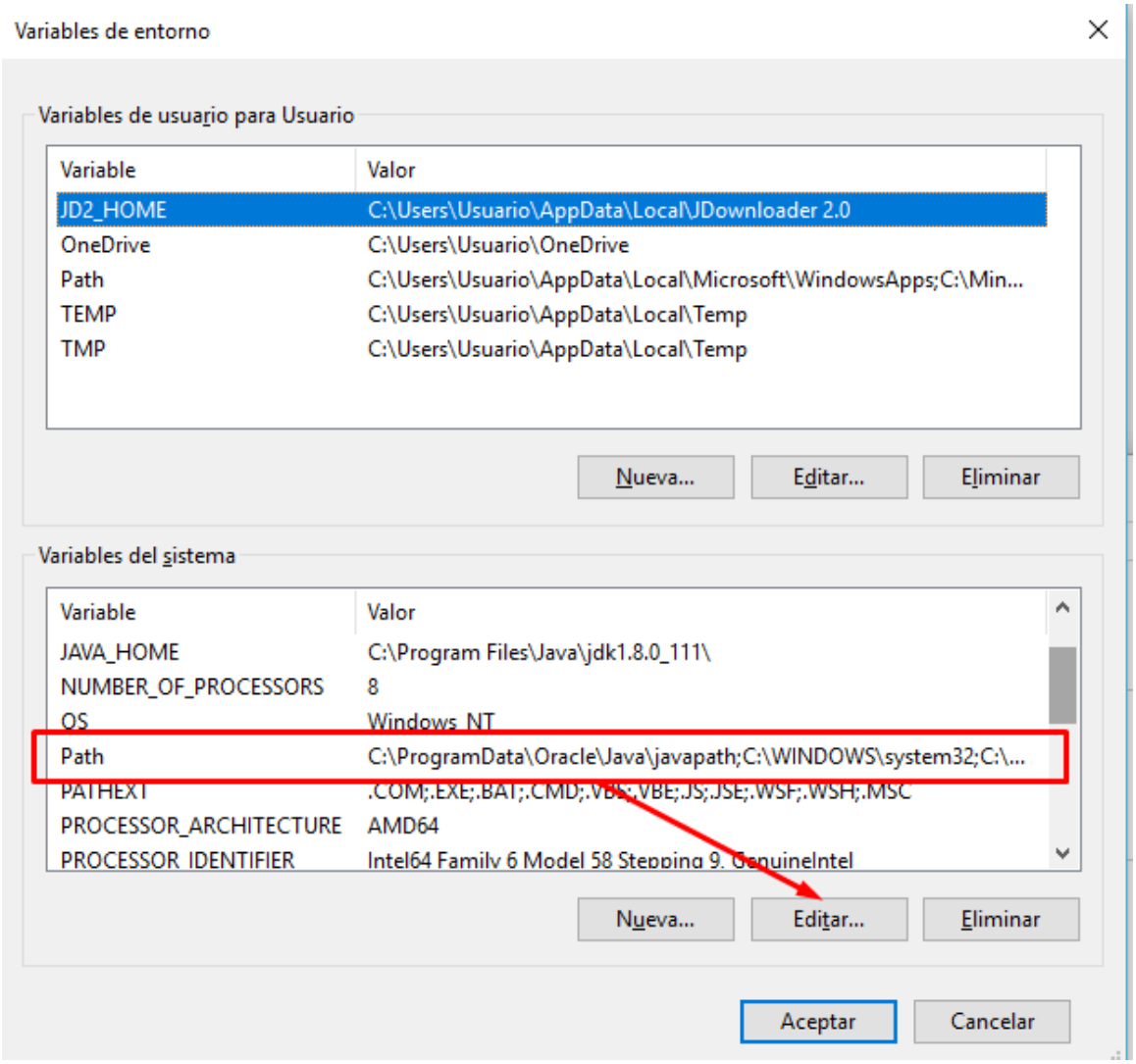
Menú Inicio → Clic derecho en Equipo → Propiedades



→ **Configuración avanzada del sistema**



→ **Variables de entorno.**



Buscamos la variable de entorno path y añadimos la ruta a la carpeta bin de donde se ha instalado el jdk

;C:\Program Files\Java\jdk1.8.0_121\bin

Esta ruta puede variar según el equipo, habrá que comprobar antes. El punto y coma es para separar del resto de las rutas incluidas anteriormente en el Path.

Ahora vamos a abrir una ventana de símbolo del sistema:

Inicio → Accesorios → Símbolo del sistema

Accederemos a la ruta de nuestra carpeta de trabajo (cualquier carpeta en tu equipo) usando el comando "cd".

Creamos un archivo de texto con el bloc de notas llamado HolaMundo.java (es muy importante respetar las mayúsculas y minúsculas y la extensión con la que se guarda).

Escribimos nuestro primer programa en el archivo:

```
public class HolaMundo {
    public static void main(String args[]) {
        System.out.println("Hello World!!!");
    }
}
```

Lo guardamos en nuestra carpeta de trabajo

Desde el símbolo del sistema escribimos:

```
C:\Users\Curso\Documents\CursoJava>javac HolaMundo.java
```

Comando javac utilizado para compilar nuestro programa, si no hay fallos genera el fichero HolaMundo.class en el mismo directorio de trabajo

```
C:\Users\Curso\Documents\CursoJava>java HolaMundo
```

Comando java para ejecutar lo que hará que el programa generado se ejecute y muestre por consola

```
Hello World!!!
```

Para programas más largos, este sistema es demasiado laborioso, pero es útil para comenzar a comprender el mundo de java.

En el siguiente tema aprenderemos a utilizar un entorno de desarrollo más cómodo que el bloc de notas y el símbolo del sistema.

Tema 4: Entornos de desarrollo (IDEs)

¿Qué es un IDE?

Un Entorno de Desarrollo Integrado (IDE) es una aplicación informática que reúne un editor de código fuente, compilador, constructor de entorno gráfico (GUI), depurador, etc. Además, hay diferentes herramientas o plugins que pueden integrarse después en un IDE dependiendo siempre de nuestras necesidades.

En cualquier caso un IDE pretende ayudar al programador en la tarea de creación de software al reunir en un mismo entorno las herramientas necesarias.

Componentes de un IDE

Los componentes imprescindibles de un completo entorno de programación son:

Editor de Texto, Intérprete, Compilador, Herramientas de Automatización y Depurador.

Además, debe incluir las siguientes características:

- Ser compatible con varias plataformas
- Soportar varios lenguajes de programación
- Ser capaz de integrarse con un sistema de control de versiones
- Reconocimiento de sintaxis
- Tener la posibilidad de añadir extensiones y otros componentes, además de poder integrar algunos de los framework más conocidos.

Casi todas las IDE's tienen autocompletado inteligente de código, lo que facilita enormemente el trabajo al no tener que recordar la estructura exacta de todas las funciones.

Un componente esencial del IDE y que va a parte es Java SE, una colección de APIs imprescindibles para poder programar en JAVA. Podréis descargar el Kit de desarrollo Java SE desde la web de Java (Ver Tema 3).

IDEs principales del mercado

Existen varios IDEs para desarrollar con java. Entre los más famosos están:

- Eclipse
- Netbenas
- InelliJ Idea

Eclipse

Eclipse es uno de los **IDE más usados por desarrolladores**, además de Java, también soporta otros lenguajes.

El entorno de desarrollo Eclipse fue creado inicialmente por IBM, pero actualmente es propiedad de la Fundación Eclipse, entidad sin ánimo de lucro que facilita la participación haciendo de Eclipse una aplicación de código abierto.

Eclipse, a diferencia de otros IDE's que incluyen todas las funciones, emplea plug-in que cada usuario va instalando según necesite.

Dispone de un editor de texto, y compilación en tiempo real.

Trabaja perfectamente en las plataformas Linux, Windows y Mac, pudiendo descargarlo totalmente gratis desde la web oficial de Eclipse.

La desventaja que puede tener es que a mayor plug-in instalados mayor consumo de recursos, haciendo de este IDE uno de los más pesados.

Netbeans

Netbeans trabaja diferentes lenguajes de programación, aunque inicialmente fue desarrollado para Java, y ha ido añadiendo otros como C/C++, HTML5 o PHP. Además a través de plug-in se pueden añadir python y algún otro lenguaje.

Uno de sus puntos fuertes es la gran facilidad para crear una interfaz gráfica sin tener que escribir el código, sólo arrastrando componentes.

Por supuesto es multiplataforma, funcionando perfectamente en Linux, Windows o Mac.

Dispone de una completa comunidad de colaboración en la que se pueden encontrar tutoriales para aprender su funcionamiento o herramientas para crear nuestros propios plug-in.

Una desventaja que algunos destacan de Netbeans es que cuantos más proyectos estén abiertos a la vez más lento se hace el IDE.

A la hora de instalarlo se recomienda no instalar la versión que incluye todos los lenguajes, pues consumiría mucha RAM, sino instalar el IDE con lenguaje que vayamos a necesitar únicamente.

IntelliJ IDEA

IntelliJ IDEA es un entorno de desarrollo integrado creado por JetBrains compatible con Windows, OS X y Linux. Este entorno es considerado por muchos, el **mejor IDE de Java**, por encima de Eclipse.

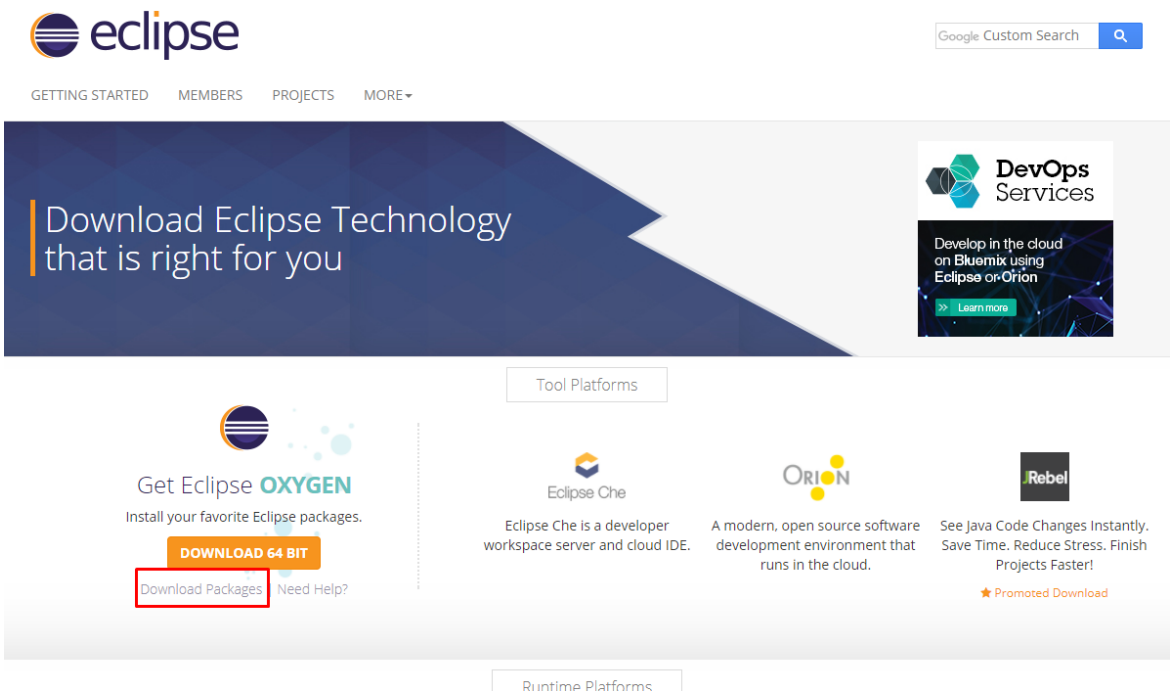
La aplicación IntelliJ IDEA contiene un editores incorporados que es compatible con la mayoría de los lenguajes de programación. Es compatible con Java EE, Android, JavaScript, etc… Incluye la posibilidad de añadir también algunos lenguajes por medio de plug-in.

Dispone de una versión gratis (IntelliJ IDEA Community) y otra de pago (IntelliJ IDEA Ultimate), siendo esta última más profesional y para uso comercial.

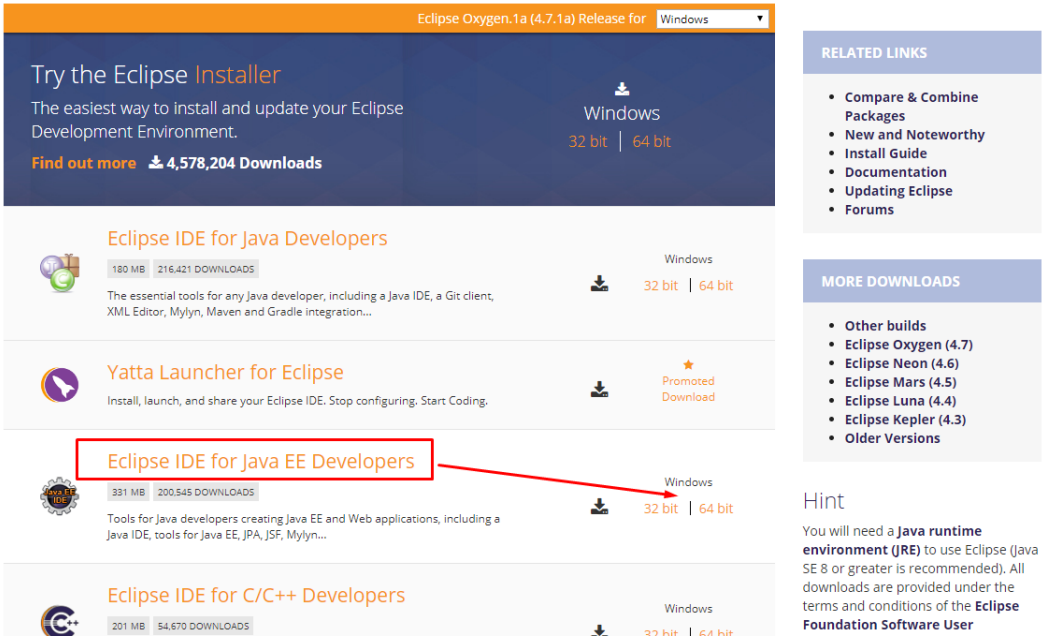
Instalacion Eclipse

Para descargar eclipse debemos acceder a la siguiente web:

http://www.eclipse.org/downloads/

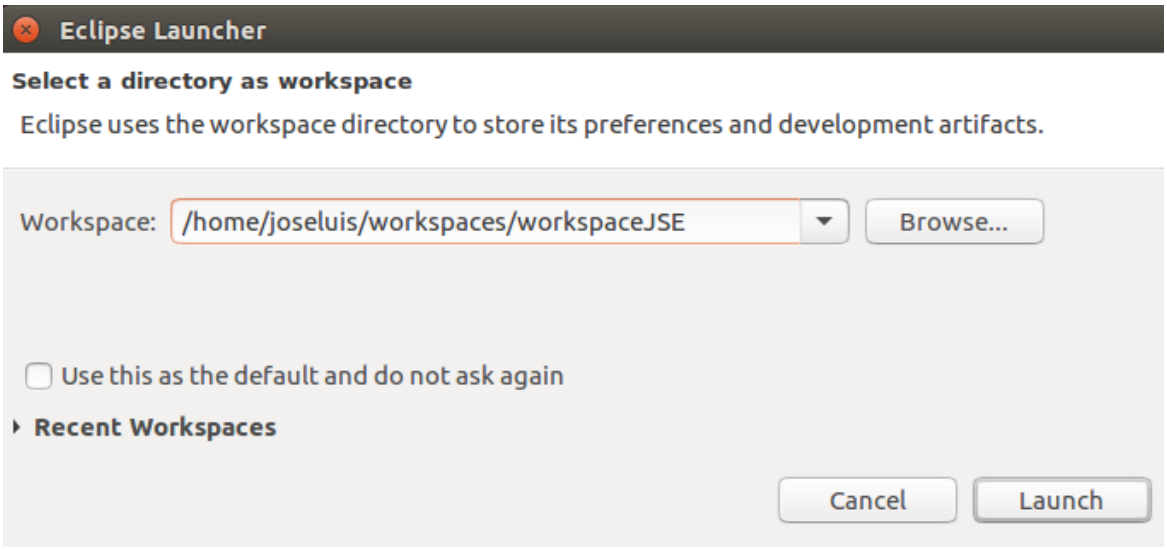


Debemos elegir "Download Packages"

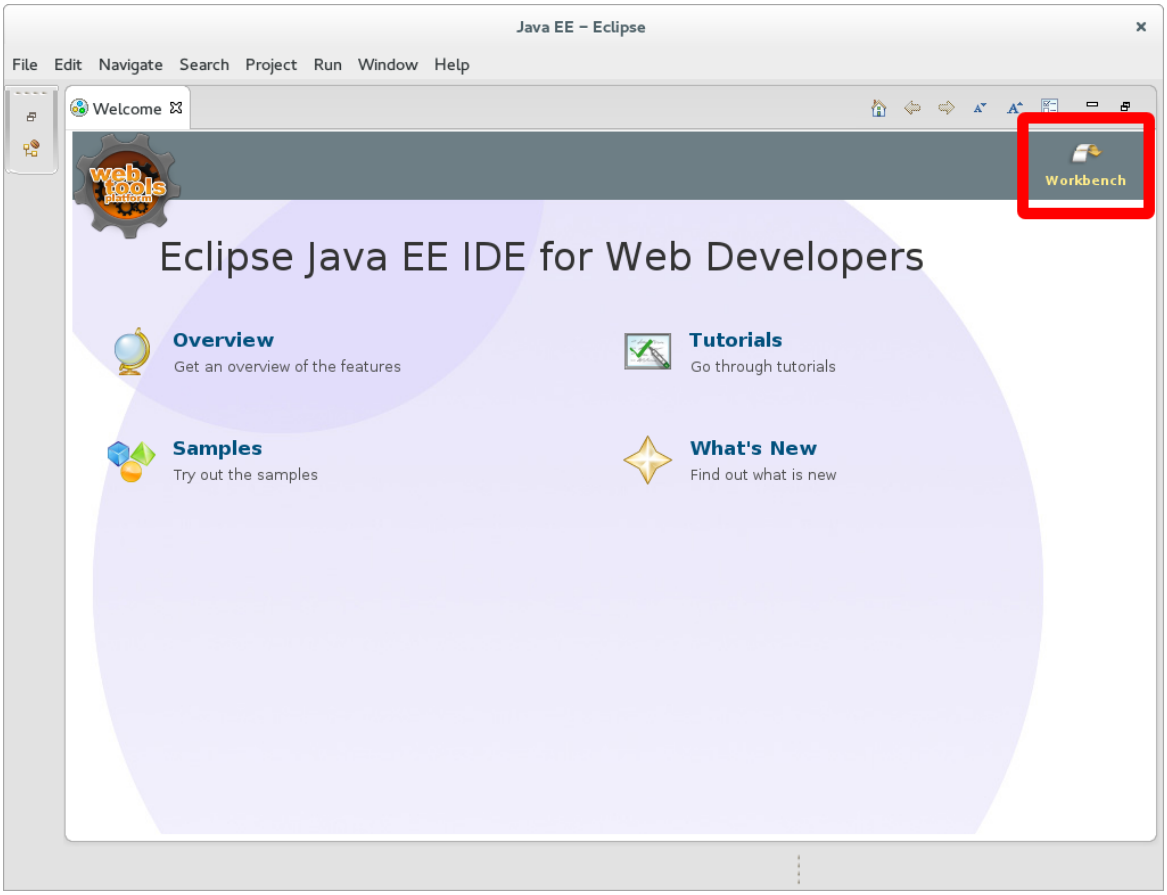


Para comenzar a trabajar con Eclipse, basta con descomprimir el archivo .zip descargado en la ubicación deseada y ejecutar el archivo Eclipse.exe.

Una vez ejecutado nos pedirá seleccionar un Workspace (espacio de trabajo)



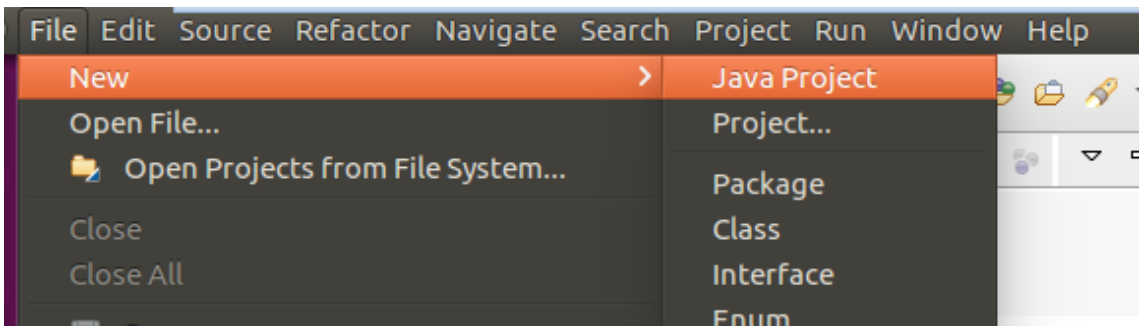
que no es más que la carpeta donde queremos ubicar nuestro trabajo. Pulsaremos el botón Browse y seleccionaremos la carpeta creada.

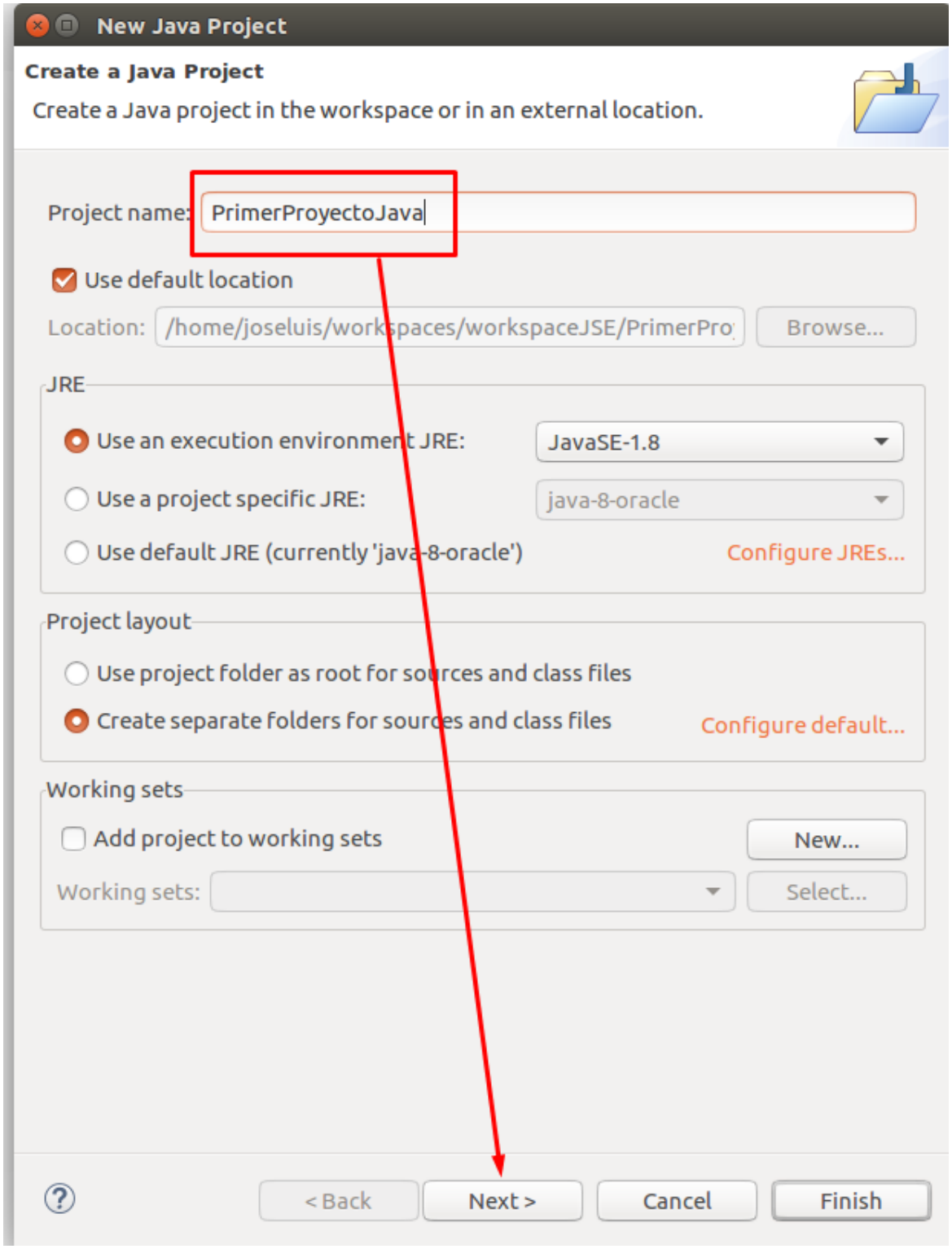


Cerraremos la ventana de bienvenida.

Ahora vamos a crear el primer proyecto, un proyecto puede estar formado por varios archivos (clases, recursos, ficheros xml, etc.), para ello:

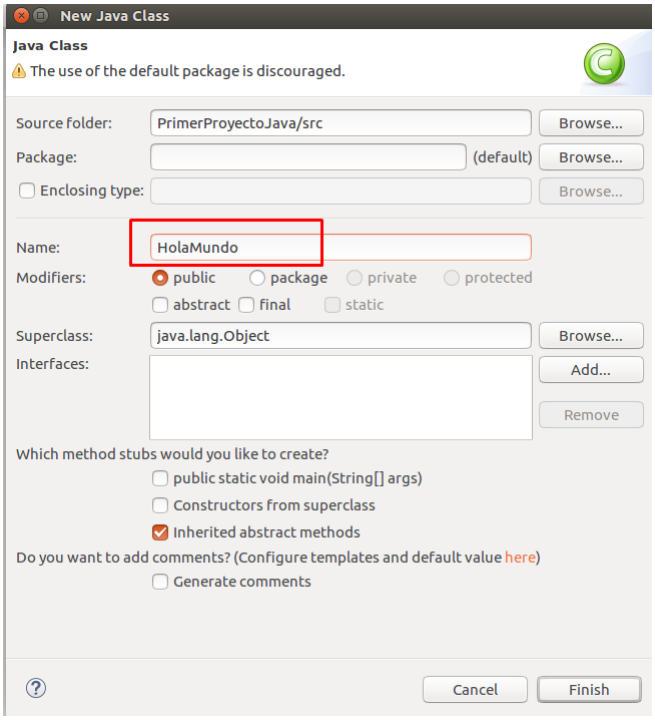
File → New → Project → Java Project →




New Java Project
Create a Java Project
Create a Java project in the workspace or in an external location.
Project name: PrimerProyectoJava
Use default location
Location: /home/joseluis/workspaces/workspaceJSE/PrimerPro
Browse...
JRE
Use an execution environment JRE:
JavaSE-1.8
Use a project specific JRE:
java-8-oracle
Use default JRE (currently 'java-8-oracle')
Configure JREs...
Project layout
Use project folder as root for sources and class files
Create separate folders for sources and class files
Configure default...
Working sets
Add project to working sets
New...
Working sets:
Select...
< Back
Next >
Cancel
Finish

Observar que en vuestra carpeta de trabajo se ha creado la carpeta PrimerProyectoJava y dentro de esta las carpetas src y bin para alojar el código fuente (archivo .java) y el codebyte (archivo .class)

Para crear nuestra primera clase, debemos hacer clic derecho sobre el nombre de nuestro proyecto y luego en el menú ir a New → Class para crear un nuevo programita con el nombre HolaMundo.



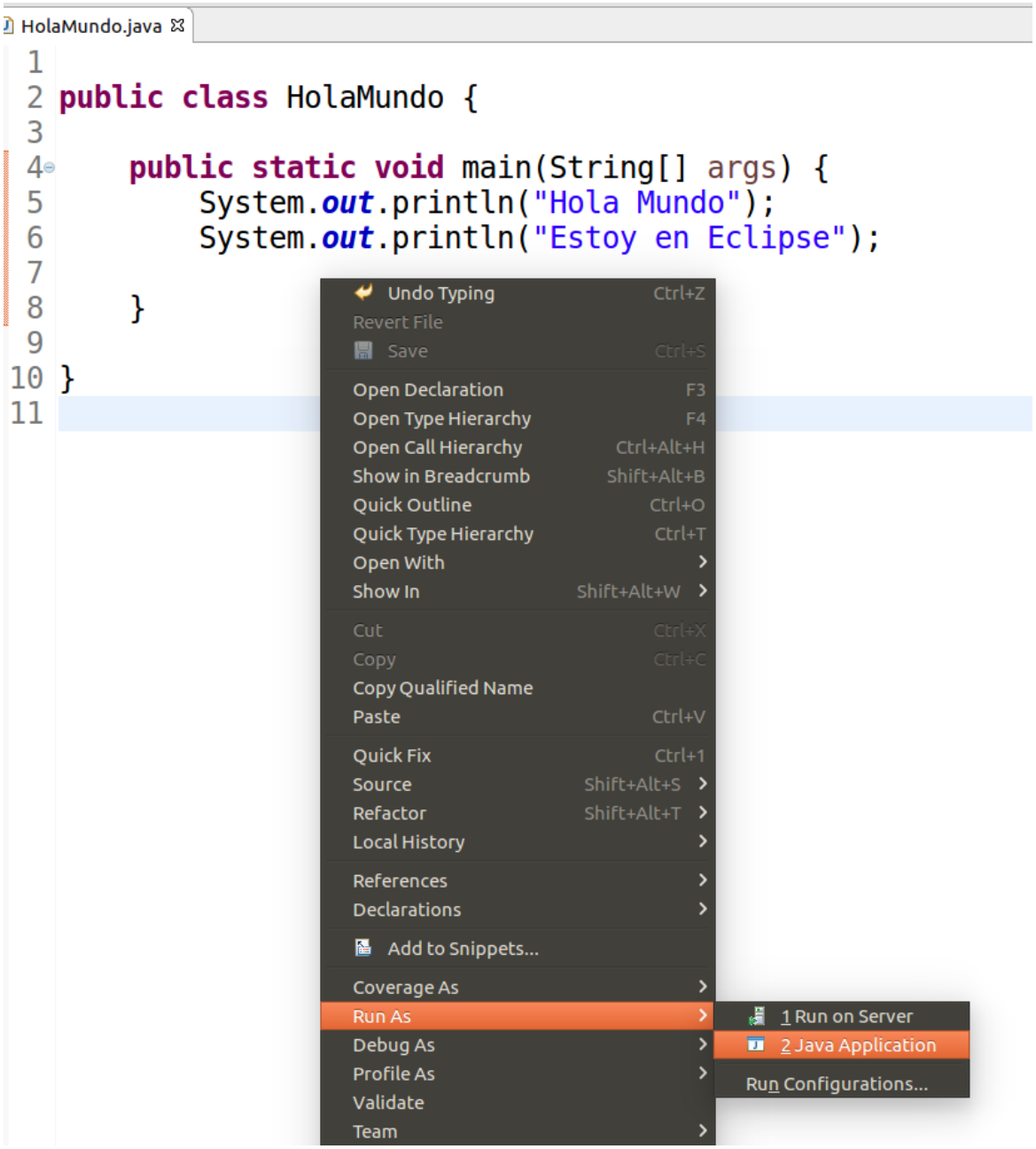
Es imprescindible que el nombre del archivo coincida exactamente con el nombre de la clase respetando las mayúsculas y minúsculas.

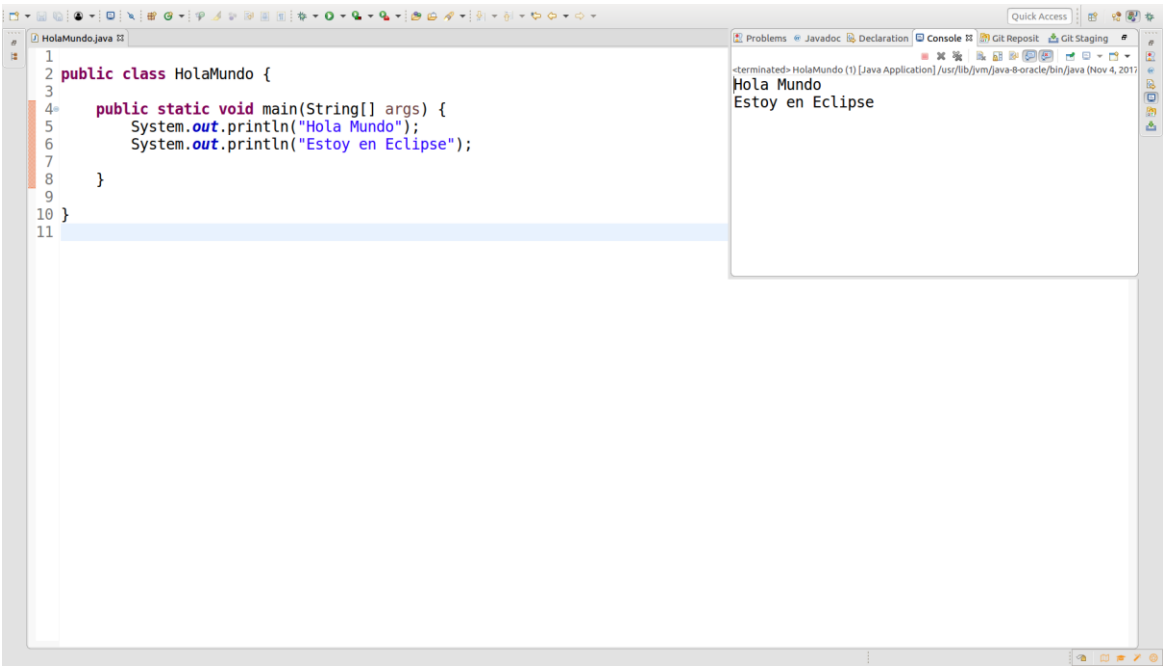
Escribiremos un ejemplo muy parecido al anterior, pero añadiremos más líneas:

HolaMundo.java

```
1
2  public class HolaMundo {
3
4      public static void main(String[] args) {
5          System.out.println("Hola Mundo");
6          System.out.println("Estoy en Eclipse");
7
8      }
9
10 }
11
```

Para ejecutarlo, basta con hacer clic con el botón derecho del ratón, seleccionar la opción "run as" y seleccionar Java Aplication.




HolaMundo.java
public class HolaMundo {
public static void main(String[] args) {
System.out.println("Hola Mundo");
System.out.println("Estoy en Eclipse");
}
}
Problems
Javadoc
Declaration
Console
Git Reposit
Git Staging
<terminated> HolaMundo (1) [Java Application] /usr/lib/jvm/java-8-oracle/bin/java (Nov 4, 2017
Hola Mundo
Estoy en Eclipse

Tema 5: Empezando con Java

Sintaxis del lenguaje

Estructura general de un programa

Java está estructurado en bloques y cada bloque comienza y termina con un símbolo de llave.

```
public class HolaMundo {

    public static void main(String[] args) {
        System.out.println("Hola Mundo");
        System.out.println("Estoy en Eclipse");

    }

}
```

Dentro de los bloques tenemos las sentencias. Una sentencia java es una expresión terminada en ";". Todas las sentencias que escribimos se compilaran y ejecutaran.

Comentarios:

Cuando escribimos código en general es útil realizar comentarios explicativos. Los comentarios no tienen efecto como instrucciones para el ordenador, simplemente sirven para que cuando un programador lea el código pueda comprender mejor lo que lee.

```
/*

  * Este es el primer programa en un IDE del curso Java
  * Creado en Noviembre de 2017

*/

// A continuación el código del programa
public class HolaMundo {

    public static void main(String[] args) {
        System.out.println("Hola Mundo");
        System.out.println("Estoy en Eclipse"); //Usamos esta sintaxis para mostrar mensajes por pantalla/consola

    }

}
```

La clase principal

Al contrario de C++ que es un lenguaje híbrido, en java todo el código del programa debe estar contenido en alguna clase. Hasta el programa más sencillo debe tener una clase principal con la siguiente estructura:

```
class <Nombre clase> {
        <cuerpo de la clase>
}
```

En programas mayores habrá varias clases, una de las cuales será la principal que contendrá la función de inicio del programa.

La función main:

El punto de inicio de cualquier programa es la función main definida en la clase principal.

Esta función siempre tendrá la siguiente forma:

```
public static void main(String args[])  {
       <grupo de sentencias de la función principal>
}
```

Los argumentos de dicha función permiten recibir valores desde la línea de comandos y procesarlos durante el programa.

```
// Esto es un ejemplo de argumentos
en línea de comandos public class
Datos {
    public static void main(String args[]) {
        System.out.println("Se han recibido "+args.length+"
        argumentos");
        for (int
            i=0;i<args.le
            ngth;i++)
            System.out.pr
            intln(args[i]
            );
    }
}
```

Este ejemplo permite recibir argumentos en línea de comandos y los muestra en pantalla.

Si se ejecuta el programa así:

```
java Datos "AMELIA GONZALEZ" MIGUEL ROSA
```

El programa dará el siguiente resultado:

```
Se han recibido 3 argumentos
AMELIA GONZALEZ
MIGUEL
ROSA
```

Al escribir parámetros entrecomillados, estos se consideran como uno sólo, aunque se introduzcan espacios en blanco.

Para especificar los argumentos desde eclipse, tenemos que hacer clic derecho en el nombre de la clase desde el explorador de paquetes (Package Explorer) y seleccionar "propiedades" del menú contextual. En Run/Debug settings seleccionamos nuestra clase y hacemos clic en el botón Edit, abrimos la ficha Arguments y escribimos los argumentos deseados en el primero cuadro.

Tipos de datos en Java

Los primeros lenguajes de programación no usaban objetos, solo variables. Una variable podríamos decir que es un espacio de la memoria del ordenador a la que asignamos un contenido que puede ser un valor numérico (sólo números, con su valor de cálculo) o de tipo carácter o cadena de caracteres (valor alfanumérico que constará sólo de texto o de texto mezclado con números).

Como ejemplo podemos definir una variable a que contenga 32 y esto lo escribimos como a = 32. Posteriormente podemos cambiar el valor de a y hacer a = 78. O hacer "a" equivalente al valor de otra variable "b" así: a = b.

Dado que antes hemos dicho que un objeto también ocupa un espacio de memoria: ¿en qué se parecen y en qué se diferencia un objeto de una variable? Consideraremos que las variables son entidades elementales: un número, un carácter, un valor verdadero o falso… mientras que los objetos son entidades complejas que pueden estar formadas por la agrupación de muchas variables y métodos. Pero ambas cosas ocupan lo mismo: un espacio de memoria (que puede ser más o menos grande).

En los programas en Java puede ser necesario tanto el uso de datos elementales como de datos complejos. Por eso en Java se usa el término "Tipos de datos" para englobar a cualquier cosa que ocupa un espacio de memoria y que puede ir tomando distintos valores o características durante la ejecución del programa. Es decir, en vez de hablar de tipos de variables o de tipos de objetos, hablaremos simplemente de tipos de datos. Sin embargo, a veces "coloquialmente" no se utiliza la terminología de forma estricta: puedes encontrarte textos o páginas web donde se habla de una variable en alusión a un objeto.

En Java diferenciamos dos tipos de datos: por un lado, **los tipos primitivos**, que se corresponden con los tipos de variables en lenguajes como C y que son los datos elementales que hemos citado. Por otro lado, **los tipos objeto** (que normalmente incluyen métodos).

TIPOS DE DATOS EN JAVA		NOMBRE	TIPO	OCUPA	RANGO APROXIMADO
	TIPOS PRIMITIVOS *(sin métodos; no son objetos; no necesitan una invocación para ser creados)*	*byte*	*Entero*	*1 byte*	*-128 a 127*
		short	*Entero*	*2 bytes*	*-32768 a 32767*
		int	*Entero*	*4 bytes*	*-2.147.483.648 a 2.147.483.647*
		long	*Entero*	*8 bytes*	*9.223.372.036.854.775.808 a 9.223.372.036.854.775.807*
		float	*Decimal simple*	*4 bytes*	*± 3,40E38*
		double	*Decimal doble*	*8 bytes*	*± 1,797E308*

<table>
<tr><td rowspan="11"></td><td rowspan="2"></td><td>char</td><td>Carácter simple</td><td>2 bytes</td><td>---</td></tr>
<tr><td>boolean</td><td>Valor true o false</td><td>1 byte</td><td>---</td></tr>
<tr><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td rowspan="9">TIPOS OBJETO
(con métodos, necesitan una invocación para ser creados)</td><td>Tipos de la biblioteca estándar de Java</td><td colspan="3">String (cadenas de texto)
Muchos otros (p.ej. Scanner, TreeSet, ArrayList…)</td></tr>
<tr><td>Tipos definidos por el programador / usuario</td><td colspan="3">Cualquiera que se nos ocurra, por ejemplo, Taxi, Autobus, Tranvia</td></tr>
<tr><td>arrays</td><td colspan="3">Serie de elementos o formación tipo vector o matriz. Lo consideraremos un objeto especial que carece de métodos.</td></tr>
<tr><td rowspan="7">Tipos envoltorio o wrapper (Equivalentes a los tipos primitivos, pero como objetos.)</td><td colspan="3">Byte</td></tr>
<tr><td colspan="3">Short</td></tr>
<tr><td colspan="3">Integer</td></tr>
<tr><td colspan="3">Long</td></tr>
<tr><td colspan="3">Float</td></tr>
<tr><td colspan="3">Double</td></tr>
<tr><td colspan="3">Character</td></tr>
<tr><td colspan="3">Boolean</td></tr>
</table>

- Un objeto es una cosa distinta a un tipo primitivo, aunque "porten" la misma información. Tener siempre presente que los objetos en Java tienen un tipo de tratamiento y los tipos primitivos, otro. Que en un momento dado contengan la misma información no significa en ningún caso que sean lo mismo. Iremos viendo las diferencias entre ambos poco a poco. De momento, recuerda que el tipo primitivo es algo elemental y el objeto algo complejo. Supón una cesta de manzanas en la calle: algo elemental. Supón una cesta de manzanas dentro de una nave espacial (considerando el conjunto nave + cesta): algo complejo. La información que portan puede ser la misma, pero no son lo mismo.
- ¿Para qué tener esa aparente duplicidad entre tipos primitivos y tipos envoltorio? Esto es una cuestión que atañe a la concepción del lenguaje de programación. Tener en cuenta una cosa: un tipo primitivo es un dato elemental y carece de métodos, mientras que un objeto es una entidad compleja y dispone de métodos. Por otro

lado, de acuerdo con la especificación de Java, es posible que necesitemos utilizar dentro de un programa un objeto que "porte" como contenido un número entero. Desde el momento en que sea necesario un objeto habremos de pensar en un envoltorio, por ejemplo Integer. Inicialmente nos puede costar un poco distinguir cuándo usar un tipo primitivo y cuándo un envoltorio en situaciones en las que ambos sean válidos. Seguiremos esta regla: usaremos por norma general tipos primitivos. Cuando para la estructura de datos o el proceso a realizar sea necesario un objeto, usaremos un envoltorio.

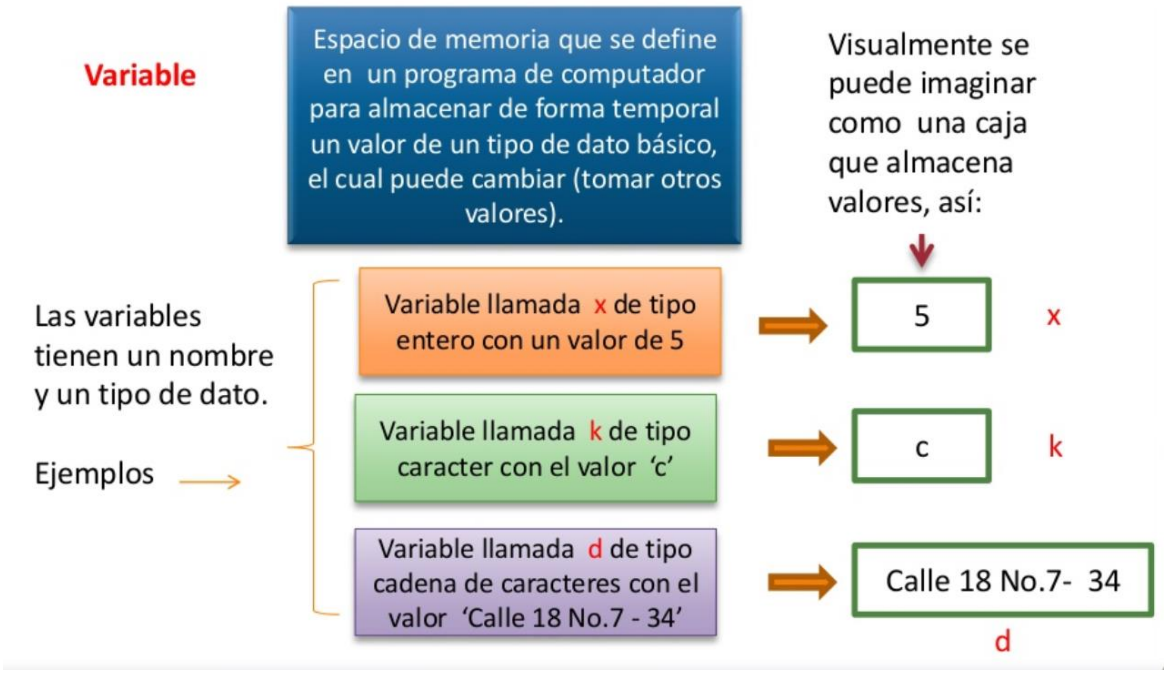
- Los nombres de tipos primitivos y envoltorio se parecen mucho. En realidad, excepto entre int e Integer y char y Character, la diferencia se limita a que en un caso la inicial es minúscula (por ejemplo double) y en el otro es mayúscula (Double). Esa similitud puede confundirnos inicialmente, pero hemos de tener muy claro qué es cada tipo y cuándo utilizar cada tipo.

- Una cadena de caracteres es un objeto. El tipo String en Java nos permite crear objetos que contienen texto (palabras, frases, etc.). El texto debe ir siempre entre comillas. Muchas veces se cree erróneamente que el tipo String es un tipo primitivo por analogía con otros lenguajes donde String funciona como una variable elemental. En Java no es así.

- Hay distintos tipos primitivos enteros. ¿Cuál usar? Por norma general usaremos el tipo int. Para casos en los que el entero pueda ser muy grande usaremos el tipo long. Los tipos byte y short los usaremos cuando tengamos un mayor dominio del lenguaje.

Declaración e inicialización de variables

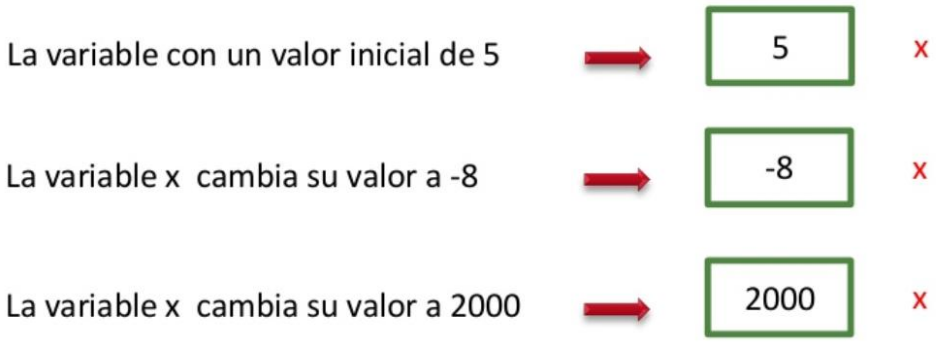
El hecho de **declarar una variable implica que se reserva un espacio de memoria para ella**, pero no que ese espacio de memoria esté ocupado, aunque pueda tener un contenido por defecto. Hay que tener en cuenta que en Java no puedes aplicar algunas normas que rigen en otros lenguajes, como que al declarar una variable entera ésta contendrá por defecto el valor cero. En Java esta situación puede dar lugar a errores de compilación: una variable entera no debemos suponer que contenga nada. Para que contenga algo debemos asignarle un contenido.

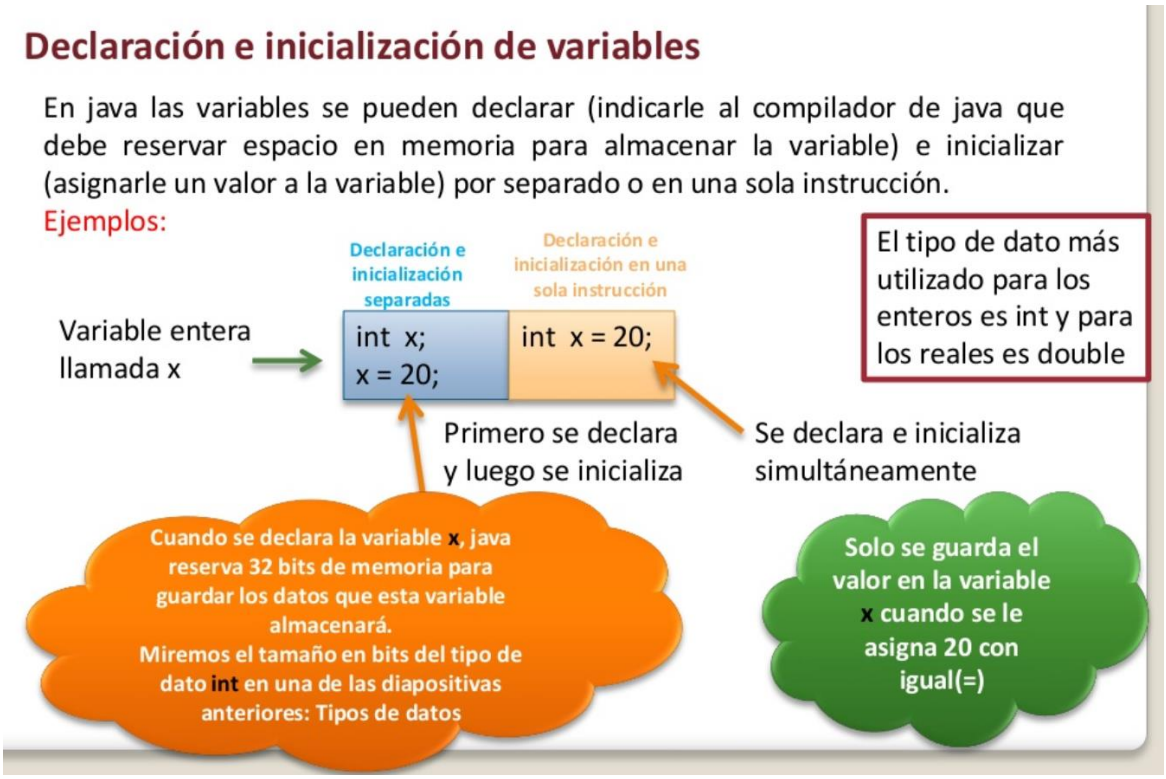


Java hace distinción entre mayúsculas y minúsculas, por lo tanto, los identificadores *Total, total y TOTAL*, harían referencia a tres variables distintas. Se recomienda que las variables de memoria comiencen por una letra minúscula y que, si están formadas por varias palabras, se utilice mayúscula al inicio de cada palabra a excepción de la primera.

- Los valores de las variables pueden cambiar mientras se está ejecutando un programa de computador.
- Las variables en Java solo aceptan valores de un tipo de dato

Si continuamos con nuestro ejemplo de la variable x de tipo entero, observemos que puede tomar diferentes valores, pero solo de tipo ENTERO.





La inicialización es un paso importante de cara a permitir un uso seguro de una variable. Es tan importante, que en general plantearemos que se haga como paso previo a cualquier otra cosa. Por ejemplo, si pensamos utilizar una variable denominada precio lo primero que haremos será establecer un valor de precio o, si no lo conocemos o lo vamos a establecer más adelante, estableceremos explícitamente un valor por defecto: por ejemplo precio = - 99; ó precio = 0; Utilizar una variable sin haberla inicializado es una práctica no recomendada en Java (mal estilo de programación) que puede dar lugar a errores o al malfuncionamiento de los programas.

Constantes de caracteres

Una constante de caracteres es un carácter encerrado entre comillas simples o apóstrofes, por ejemplo 'T' o '3'. Estos son en realidad caracteres simbólicos que representan el valor numérico de los caracteres del conjunto de caracteres Unicode.

```
public class Caracter {
    public static void main(String args[]) {
       char c='A'; System.out.println("Caracter:
       "+c);
      System.out.println("Equivale
        ncia Unicode: "+(int)c);
    }
```

```
}
```

Existen además unos cuantos caracteres que tienen nombres estándar. Estos caracteres se denominan secuencias de escape y se representan con el carácter diagonal invertida seguido de una letra o código que representa dicha secuencia de escape.

\n	Nueva línea
\t	Tabulador
\b	Retroceso
\r	Retorno de carro
\f	Avance de pagina
\\	Barra invertida
\'	Apóstrofe
\"	Comilla
\uhhhh	Carácter Unicode

Literales de cadena

Una constante literal de cadena es una secuencia de caracteres encerrada entre comillas como la siguiente:

"Aprendo java porque me gusta"

En las constantes literales se pueden emplear los caracteres de escape mencionados anteriormente como se muestra en el ejemplo:

```
public class Escape {
    public static void main(String args[]) {
        System.out.println("Primavera\nVerano\nOtoño\nInvierno");
        System.out.println("Lun\tMar\tMie\tJue\tVie");
        System.out.println("Lun\\Mar\\Mie\\Jue\\Vie");
        System.out.println("\"Cocodrilo\"");
        System.out.println("\'Hipopotamo\'");
        System.out.println("\u0001 Caracter Unicode");
    }
}
```

Declaración de datos de tipo constantes

Para declarar un dato de tipo constante se utiliza la palabra reservada final como en el siguiente ejemplo:

```
final float PI=3.1416F;
```

Cualquier intento de modificar el valor de PI producirá un error de compilación.

Operadores Aritméticos

En Java disponemos de los operadores aritméticos habituales en lenguajes de programación como son suma, resta, multiplicación, división y operador que devuelve el resto de una división entre enteros (en otros lenguajes denominado operador mod o módulo de una división):

OPERADOR	*DESCRIPCIÓN*
+	*Suma*
–	*Resta*
*	*Multiplicación*
/	*División*
%	*Resto de una división entre enteros (en otros lenguajes denominado mod)*

Las operaciones con operadores siguen un **orden de prelación o de precedencia** que determinan el orden con el que se ejecutan. Si existen expresiones con varios operadores del mismo nivel, la operación se ejecuta de izquierda a derecha. Para evitar resultados no deseados, en casos donde pueda existir duda se recomienda el uso de paréntesis para dejar claro con qué orden deben ejecutarse las operaciones. Por ejemplo, si dudas si la expresión 3 * a / 7 + 2 se ejecutará en el orden que tú deseas, especifica el orden deseado utilizando paréntesis:

por ejemplo 3 * ((a / 7) + 2).

También tenemos los operadores **++ y --** que lo que hacen es aumentar o disminuir en uno la variable sobre la que se ha asignado.

Ejemplo:

```
int num = 0;

num++;//Aumentará el valor de num en 1, es como poner num = num+1;

num--;//Disminuye el valor de num en 1, es como poner num = num-1;
```

Los **operadores de sentencia de asignación +=** se denomina operador de asignación compuesta y sirve para sumarle una cantidad

al valor de una variable. También se admite el uso del operador **-=** que en vez de sumar lo que hace es restar. El uso de los operadores de asignación compuesta es opcional: hay programadores que los usan y otros que no.

Ejemplo:

```
int num = 0;
num+=2;//Es como poner num = num+2;
```

Operadores Lógicos

En Java disponemos de los operadores lógicos habituales en lenguajes de programación como son "es igual", "es distinto", menor, menor o igual, mayor, mayor o igual, and (y), or (o) y not (no). La sintaxis se basa en símbolos como veremos a continuación y cabe destacar que hay que prestar atención a no confundir == con = porque implican distintas cosas.

OPERADOR	*DESCRIPCIÓN*
==	*Es igual*
!=	*Es distinto*
<, <=, >, >=	*Menor, menor o igual, mayor, mayor o igual*
&&	*Operador and (y)*
||	*Operador or (o)*
!	*Operador not (no)*

El operador || se obtiene en la mayoría de los teclados pulsando ALT GR + 1, es decir, la tecla ALT GR y el número 1 simultáneamente.

Los operadores && y || se llaman operadores en cortocircuito porque si no se cumple la condición de un término no se evalúa el resto de la operación. Por ejemplo: (a == b && c != d && h >= k) tiene tres evaluaciones: la primera comprueba si la variable a es igual a b. Si no se cumple esta condición, el resultado de la expresión es falso y no se evalúan las otras dos condiciones posteriores.

En un caso como (a < b || c != d || h <= k) se evalúa si a es menor que b. Si se cumple esta condición el resultado de la expresión es verdadero y no se evalúan las otras dos condiciones posteriores.

El operador ! recomendamos no usarlo hasta que se tenga una cierta destreza en programación. Una expresión como (!esVisible) devuelve false si (esVisible == true), o true si (esVisible == false). En general existen expresiones equivalentes que permiten evitar el uso de este operador cuando se desea.

Tema 6: Sentencias de control

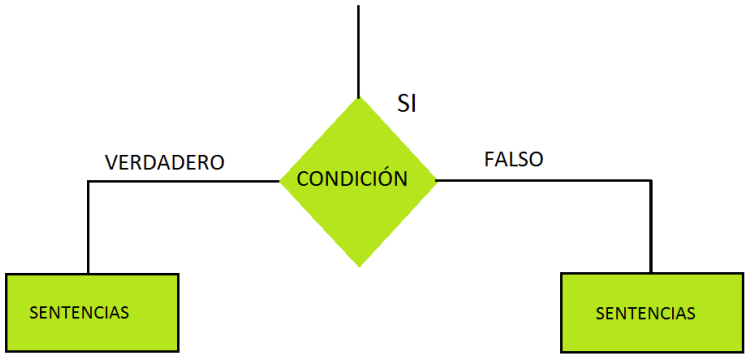
En Java al igual que todos los lenguajes de programación proporciona un conjunto de estructuras de control que permiten manejar el código de nuestro programa, inicialmente realizamos operaciones básicas como sumas, restas, multiplicaciones, pero en la gran parte de nuestros programas tendremos que utilizar condiciones, recorrer conjuntos de datos, validaciones etc. Y es en esas situaciones no tendremos más remedio que utilizar las estructuras de control.

Vamos a ver las **estructuras de control condicionales**, este tipo de estructuras permiten cambiar el flujo normal de un programa de acuerdo a una condición siempre y cuando esta sea verdadera; en palabras no técnicas permiten elegir un opción de entre muchas de acuerdo a una condición siempre que esta sea condición sea verdadera.

Estrucutras de decisión if else, if else if.

La instrucción if … else permite controlar qué procesos tienen lugar, típicamente en función del valor de una o varias variables, de un valor de cálculo o booleano, o de las decisiones del usuario. La sintaxis a emplear es:

```
if (condición) {
    instrucciones
} else {
    instrucciones
 }
```



Ejemplos:

```
// if: varias líneas en cada alternativa.
  public class Condicional {
        public static void main(String args[]) {
              int edad=25;
              if (edad<18) {
                    System.out.println("Manor de edad");
                    System.out.println("Todavía no puedes entrar a la disco de noche");
              }
              else {
                    System.out.println("Mayor de edad");
                    System.out.println("Ya puedes ir sólo por la vida, si te atreves");
              }
        }
}
```

La cláusula else (no obligatoria) sirve para indicar instrucciones a realizar en caso de no cumplirse la condición. Java admite escribir un else y dejarlo vacío: else { }. El else vacío se interpreta como que contemplamos el caso pero no hacemos nada en respuesta a él. Un else vacío no tiene ningún efecto y en principio carece de utilidad, no obstante, a veces es usado para remarcar que no se ejecuta ninguna acción cuando se alcanza esa situación.

Cuando se quieren evaluar distintas condiciones una detrás de otra, se usa la expresión else if { }. En este caso no se admite elseif todo junto como en otros lenguajes. De este modo, la evaluación que se produce es: si se cumple la primera condición, se ejecutan ciertas instrucciones; si no se cumple, comprobamos la segunda, tercera, cuarta… n condición. Si no se cumple ninguna de las condiciones, se ejecuta el else final en caso de existir.

```
// Varios if anidados
public class Condicional {
      public static void main(String args[]) {
            int edad=34;
            if (edad<18) {
                  System.out.println("Manor de edad");
                  System.out.println("Todavía no puedes entrar a la disco de noche");
            }
            else if (edad<27) {
                  System.out.println("Mayor de edad");
                  System.out.println("Ya puedes ir sólo por la vida, si te atreves");
            }
            else {
                  System.out.println("Mayor de edad");
                  System.out.println
                        ("Lastima, ya no te hacen descuento en los albergues");
            }
      }
}
```

Condicional de selección switch

La instrucción switch es una forma de expresión de un anidamiento múltiple de instrucciones if ... else. Su uso no puede considerarse, por tanto, estrictamente necesario, puesto que siempre podrá ser sustituida por el uso de if. No obstante, a veces nos resultará útil al introducir mayor claridad en el código.

La sintaxis será:

```
switch (expresión) {

        case valor1:
        instrucciones;
        break;

        case valor2:
        instrucciones;
        break;
        .
        .
        .
        default:
        sentencias;
        break;

 }
```

La cláusula default es opcional y representa las instrucciones que se ejecutarán en caso de que no se verifique ninguno de los casos evaluados. El último break dentro de un switch (en default si existe esta cláusula, o en el último caso evaluado si no existe default) también es opcional, pero lo incluiremos siempre para ser metódicos.

Switch solo se puede utilizar para evaluar ordinales (por ordinal entenderemos en general valores numéricos enteros o datos que se puedan asimilar a valores numéricos enteros) y cadenas (String).

Switch solo permite evaluar valores concretos de la expresión: no permite evaluar intervalos (pertenencia de la expresión a un intervalo o rango) ni expresiones compuestas.

Código de ejemplo:

```
String valor="A";
switch (valor){
    case "V": System.out.println("Valor V"); break;
    case "A": System.out.println("Valor A");
    case "B": System.out.println("Valor B"); break;

    case "Z": System.out.println("VAlor Z"); break;
    default: System.out.println("Valor desconocido");break;

}
```

Bucles en java

Nos referimos a estructuras de repetición o bucles en alusión a instrucciones que permiten la repetición de procesos un número n de veces. Los bucles se pueden materializar con distintas instrucciones como for, while, etc. Un bucle se puede anidar dentro de otro dando lugar a que por cada repetición del proceso exterior se ejecute n veces el proceso interior.

En java existen cuatro tipos de bucles:

Bucle for

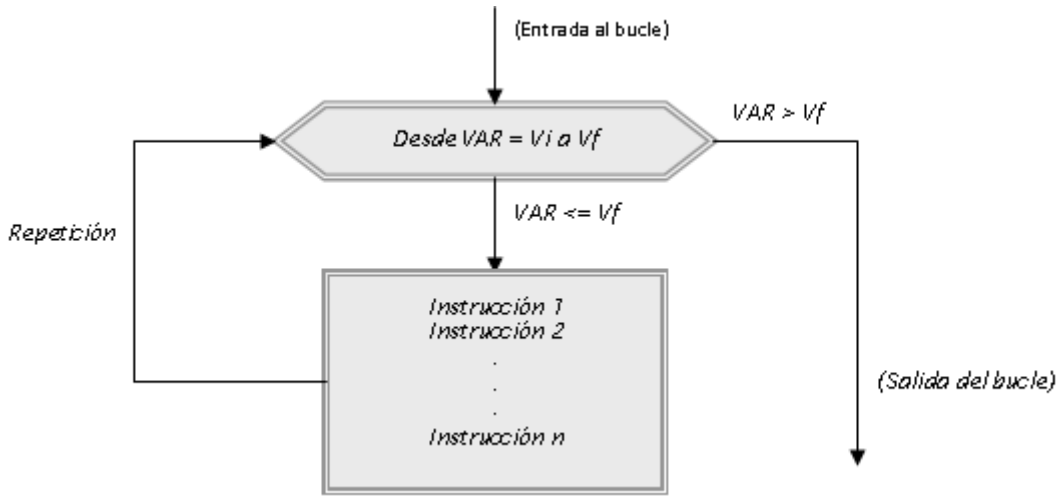
Bucle while

Bucle do while

Bucle foreach

Bucle for

En Java existen distintas modalidades de for. El caso más habitual, que es el que expondremos a continuación, lo denominaremos for normal o simplemente for. Conceptualmente el esquema más habitual es el siguiente:



La sintaxis habitual es:

```
for (int i = unNumero; i < otroNumero; i++) {
instrucciones a ejecutarse
}
```

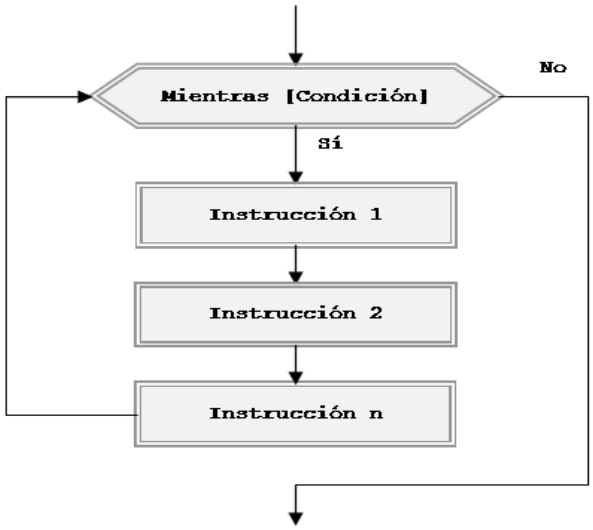
donde int i supone la declaración de una variable específica y temporal para el bucle. El nombre de la variable puede ser cualquiera, pero suelen usarse letras como i, j, k, etc. unNumero refleja el número en el que se empieza a contar, con bastante frecuencia es 0 ó 1. i < otroNumero ó i <= otroNumero refleja la condición que cuando se verifique supondrá la salida del bucle y el fin de las repeticiones. i++ utiliza el operador ++ cuyo significado es "incrementar la variable i en una unidad". Este operador se puede usar en cualquier parte del código, no es exclusivo para los bucles for. Igualmente se dispone del operador "gemelo" – –, que realiza la operación en sentido contrario: reduce el valor de la variable en una unidad.

```
// Ejemplo for
public class Para {
        public static void main(String args[]) {
                int c;
                for (c=1;c<=10;c++)
                        System.out.println(c);
        }
}
```

Un bucle for (o de cualquier otro tipo) puede ser interrumpido y finalizado en un momento intermedio de su ejecución mediante una instrucción **break**; El uso de esta instrucción dentro de bucles solo tiene sentido cuando va controlada por un condicional que determina que si se cumple una condición, se interrumpe la ejecución del bucle.

While y do while

El bucle while presenta ciertas similitudes y ciertas diferencias con el bucle for. La repetición en este caso se produce no un número predeterminado de veces, sino mientras se cumpla una condición. Conceptualmente el esquema más habitual es el siguiente:



La sintaxis en general es:

```
while (condición) {
instrucciones a ejecutarse
}
```

donde condición es una expresión que da un resultado true o false en base al cual el bucle se ejecuta o no. Escribe y prueba el siguiente código, donde además vemos un ejemplo de uso de la instrucción break;

```
// Ejemplos while
public class Mientras {
        public static void main(String args[]) {
                int i=0;
                while (i<10) {  //while
                        i++; System.out.println(i);
                }
                i=0;
                do {  //do....while
                        i++; System.out.println(i);
                } while (i<10);
        }
}
```

Bucle do ... while

El bucle do … while es muy similar al bucle while. La diferencia radica en cuándo se evalúa la condición de salida del ciclo. En el bucle

while esta evaluación se realiza antes de entrar al ciclo, lo que significa que el bucle puede no llegar ejecutarse. En cambio, en un bucle do … while, la evaluación se hace después de la primera ejecución del ciclo, lo que significa que el bucle obligatoriamente se ejecuta al menos en una ocasión.

```
// Ejemplo break y continue.
public class Interrupcion {
        public static void main(String args[]) {
                int n=0;
                do {
                        n++;
                        System.out.println(n);
                        if (n==10) break; // sale de un bucle for, while o do while.
                } while (true);
                n=0;
                do {
                        n++;
                        System.out.println(n);
                        if (n>10) continue;
                        // se salta el resto de las sentencias yendo al principio del bucle.
                        n++;
                } while (n < 20);
        }
}
```

Tema 7: Utilización de las librerías básicas de Java

La clase String

Los Strings son una secuencia de caracteres, incluyendo letras (mayúsculas y minúsculas), signos, espacios, caracteres especiales, etc. En palabras simples, los Strings sirven para guardar palabras, oraciones, etc.

En java los Strings son objetos, tipo String. Al momento de crearlos, la manera más sencilla es:

```
String hola = "Hello World";
```

Los Strings, al ser objetos, también pueden ser inicializados como tales, a través del constructor:

```
String s = new String(); // String vacio
String s = new String("Hola"); // String "Hola"
```

La principal característica de los Strings es que **son inmutables**, es decir un String no se puede modificar una vez se ha creado, si se modifica el valor será almacenado en una varible para poder ser manejado.

Cómo se obtiene información acerca del string

Una vez creado un objeto de la clase String podemos obtener información relevante acerca del objeto a través de las funciones miembro.

Para obtener la longitud, número de caracteres que guarda un string se llama a la función miembro length.

```
String str="El primer programa";
int longitud=str.length();
```

Podemos conocer si un string comienza con un determinado prefijo, llamando al método startsWith, que devuelve true o false, según que el string comience o no por dicho prefijo

```
String str="El primer programa";
boolean resultado=str.startsWith("El");
```

En este ejemplo la variable resultado tomará el valor true.

De modo similar, podemos saber si un string finaliza con un conjunto dado de caracteres, mediante la función miembro endsWith.

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

```
boolean resultado=str.endsWith("programa");
```

Si se quiere obtener la posición de la primera ocurrencia de la letra p, se usa la función indexOf.

```
String str="El primer programa";
int pos=str.indexOf('p');
```

Para obtener las sucesivas posiciones de la letra p, se llama a otra versión de la misma función

```
pos=str.indexOf('p', pos+1);
```

El segundo argumento le dice a la función indexOf que empiece a buscar la primera ocurrencia de la letra p a partir de la posición pos+1.

Comparación de strings

La comparación de strings nos da la oportunidad de distinguir entre el operador lógico == y la función miembro equals de la clase String. En el siguiente código

```
String str1="El lenguaje Java";
String str2=new String("El lenguaje Java");
if(str1==str2){
        System.out.println("Los mismos objetos");
}else{
        System.out.println("Distintos objetos");
}
if(str1.equals(str2)){
        System.out.println("El mismo contenido");
}else{
        System.out.println("Distinto contenido");
}
```

Esta porción de código devolverá que str1 y str2 son distintos objetos pero con el mismo contenido. str1 y str2 ocupan posiciones distintas en memoria pero guardan los mismos datos.

Cambiemos la segunda sentencia y escribamos

```
String str1="El lenguaje Java";
String str2=str1;
System.out.prinln("Son el mimso objeto "+(str1==str2);
```

Los objetos str1 y str2 guardan la misma referencia al objeto de la clase String creado. La expresión (str1==str2) devolverá true.

Así pues, el método equals compara un string con un objeto cualquiera que puede ser otro string, y devuelve true cuando dos strings son iguales o false si son distintos.

```
String str="El lenguaje Java";
boolean resultado=str.equals("El lenguaje Java");
```

La variable resultado tomará el valor true.

La función miembro comapareTo devuelve un entero menor que cero si el objeto string es menor (en orden alfabético) que el string dado, cero si son iguales, y mayor que cero si el objeto string es mayor que el string dado.

```
String str="Tomás";
int resultado=str.compareTo("Alberto");
```

La variable entera resultado tomará un valor mayor que cero, ya que Tomás está después de Alberto en orden alfabético.

```
String str="Alberto";
int resultado=str.compareTo("Tomás");
```

La variable entera resultado tomará un valor menor que cero, ya que Alberto está antes que Tomás en orden alfabético.

Extraer un substring de un string

En muchas ocasiones es necesario extraer una porción o substring de un string dado. Para este propósito hay una función miembro de la clase String denominada substring.

Para estraer un substring desde una posición determinada hasta el final del string escribimos

```
String str="El lenguaje Java";
String subStr=str.substring(12);
```

Se obtendrá el substring "Java".

Una segunda versión de la función miembro substring, nos permite extraer un substring especificando la posición de comienzo y la el final.

```
String str="El lenguaje Java";
String subStr=str.substring(3, 11);
```

Se obtendrá el substring "lenguaje". Recuérdese, que las posiciones se empiezan a contar desde cero.

Convertir un número a string

Para convertir un número en string se emplea la función miembro estática valueOf (más adelante explicaremos este tipo de funciones).

```
int valor=10;
```

```
String str=String.valueOf(valor);
```

La clase String proporciona versiones de valueOf para convertir los datos primitivos: int, long, float, double.

Convertir un string en número

Cuando introducimos caracteres en un control de edición a veces es inevitable que aparezcan espacios ya sea al comienzo o al final. Para eliminar estos espacios tenemos la función miembro trim

```
String str="  12 ";
String str1=str.trim();
```

Para convertir un string en número entero, primero quitamos los espacios en blanco al principio y al final y luego, llamamos a la función miembro estática parseInt de la clase Integer (clase envolvente que describe los números enteros)

```
String str="  12 ";
int numero=Integer.parseInt(str.trim());
```

Para convertir un string en número decimal (double) se requieren dos pasos: convertir el string en un objeto de la clase envolvente Double, mediante la función miembro estática valueOf, y a continuación convertir el objeto de la clase Double en un tipo primitivo double mediante la función doubleValue

```
String str="12.35 ";
double numero=Double.valueOf(str).doubleValue();
```

Se puede hacer el mismo procedimiento para convertir un string a número entero

```
String str="12";
int numero=Integer.valueOf(str).intValue();
```

RESUMEN METODOS DE LA CLASE String	
char charAt(int index)	*Retorna un carácter en el índice especificado.*
int compareTo(String str)	*Compara dos objetos String lexicográficamente. Devuelve 0 si los dos Strings son iguales, un número menor que cero sí la cadena es lexicográficamente menor que el argumento*
int compareToIgnoreCase(String str)	*Compara dos objetos String lexicográficamente ignorando diferencias entre mayúsculas y minúsculas.*
String concat(String str)	*Concatena el String especificado al final de este String.*
static String copyValueOf(char[] dato,	*Retorna un String que es equivalente al array de caracteres especificado.*
static String copyValueOf(char[] dato)	*Retorna un String que es equivalente al array de caracteres especificado.*
boolean endsWith(String, sufijo)	*Comprueba si el String finaliza con el prefijo indicado.*
boolean equals(String str)	*Compara las dos cadenas y devuelve true o false.*

boolean equalsIgnoreCase(String str)	*Compara las dos cadenas ignorando diferencia entre mayúsculas y minúsculas.*
byte[] getBytes()	*Convierte el String en un array de bytes según la codificación*
void getChars(int ini, int fin, char array[], int iniArray)	*Copia caracteres de este String en un array de caracteres. Los argumentos son: ini: posición del primer carácter a copiar, comenzando por cero. fin: posición detrás del último carácter a copiar. array: array destino. iniArray: posición del array donde comenzará la copia.*
int hashCode()	*Retorna un código único para este String.*
int indexOf(int ch)	*Retorna el índice donde se encuentra la primera ocurrencia del carácter especificado.*
int indexOf(int ch, int indice)	*Retorna el índice donde se encuentra la primera ocurrencia del carácter especificado a partir de la posición del índice especificado.*
int indexOf(String str)	*Retorna el índice donde se encuentra la primera ocurrencia de la subcadena especificada.*
int indexOf(String str, int indice)	*Retorna el índice donde se encuentra la primera ocurrencia de la subcadena especificada a partir de la posición del índice especificado.*
int lastIndexOf(int ch)	*Retorna el índice donde se encuentra la última ocurrencia del*
int lastIndexOf(int ch, int indice)	*Retorna el índice donde se encuentra la última ocurrencia del carácter especificado a partir de la posición del*
int LastIndexOf(String str)	*Retorna el índice donde se encuentra la última ocurrencia de la subcadena especificada.*
int LastIndexOf(String str, int indice)	*Retorna el índice donde se encuentra la última ocurrencia de la subcadena especificada a partir de la posición del índice especificado.*
int length()	*Retorna el número de caracteres de este String.*
boolean regionMatches(int iniCad, String cad, int iniCadArg, int len)	*Comprueba si una porción de la cadena coincide con otra porción de la cadena especificada como argumento. Los parámetros son: iniCad: posición de inicio de la porción de la cadena. cad: cadena especificada como argumento. iniCadArg: posición de inicio de la porción de la cadena especificada como argumento.*
String replace(char angituo, char nuevo)	*Retorna una nueva cadena que resulta de reemplazar todas las ocurrencias de antiguo por nuevo.*
String substring(int inicio)	*Retorna una cadena que se genera extrayendo los caracteres*
String substring(int inicio, int final)	*Retorna una cadena que se genera extrayendo los caracteres*
char[] toCharArray()	*Devuelve un nuevo array de caracteres a partir de la*
String toLowerCase()	*Convierte el String a minúsculas.*
static String toUpperCase()	*Convierte el String a mayúsculas.*
static String valueOf(boolean b)	*Devuelve la representación en String del argumento*
static String valueOf(char c)	*Devuelve la representación en String del argumento*
static String valueOf(char[] array)	*Devuelve la representación en String del argumento*
static String valueOf(double d)	*Devuelve la representación en String del argumento*
static String valueOf(float f)	*Devuelve la representación en String del argumento*
static String valueOf(long l)	*Devuelve la representación en String del argumento*
static String valueOf(int i)	*Devuelve la representación en String del argumento*
static String valueOf(object o)	*Devuelve la representación en String del argumento*
static String trim()	*Elimina los espacios en blanco sobrantes de los extremos del*

Uso de la clase Math

La clase Math representa la librería matemática de Java. Las funciones que contiene son las de todos los lenguajes, parece que se han metido en una clase solamente a propósito de agrupación, por eso se encapsulan en Math, y lo mismo sucede con las demás clases que corresponden a objetos que tienen un tipo equivalente (Character, Float, etc.). El constructor de la clase es privado, por los que no se pueden crear instancias de la clase. Sin embargo, Math es public para que se pueda llamar desde cualquier sitio y static para que no haya que inicializarla.

Dicha clase se encuentra en el paquete básico java.lang.

```
public class Matematicas {
        public static void main(String args[]) {
                System.out.println("Raiz Cuadrada de 81");
                System.out.println(Math.sqrt(81));
                System.out.println("Valor de PI");
                System.out.println(Math.PI);
                System.out.println("Seno del ángulo de radio 30");
                System.out.println(Math.sin(30*Math.PI/180));
                System.out.println("Coseno del ángulo de radio 30");
                System.out.println(Math.cos(30*Math.PI/180));
                System.out.println("Valor de 5 elevado al cubo");
                System.out.println(Math.pow(5,3));
        }
}
```

Las clases Wrapper o envoltorio de tipos

Las clases de Java de envoltorio de tipos se usan con los tipos de datos básicos, tales como boolean, char, int, long, double y float, para proveer métodos adicionales para la ejecución de operaciones sobre estos datos.

A menudo, las clases de envoltorio tienen el mismo nombre que el tipo básico, solo que comienzan con mayúsculas.

Estos son algunos nombres de clases de envoltorio de tipos Java:

- Boolean
- Character
- Double
- Float
- Integer
- Long

Métodos comunes:

Formato	*Ejemplo*	*Descripción*
public classtype(basicty pe)	*Integer objNumero = new Integer(65); Character objCaracter = new Character('A');*	*Se usa como un constructor para crear el objeto de la clase classtype.*
public basictype classtypeValue()	*numero = objNumero.intValue(); caracter = objCaracter.charValue();*	*Extrae desde una clase envoltorio, el valor de tipo básico.*
public String toString();	*System.out.println(objNumero.to Stri ng()); System.out.println(objCaracter.to Stri ng());*	*Convierte el valor almacenado en la clase envoltorio al tipo equivalente en la*
public boolean equals(Object obj);	*if (objNumero.equals(objNumero2)) System.out.println("S on Iguales"); else*	*Verifica si el valor en un objeto envoltorio de tipo es igual a otro en otro objeto envoltorio de tipo.*

```
// Clases de envoltorio de tipo
public class Envoltorio {
        public static void main (String[] args) {
                Integer objNumero = new Integer(65);
                Integer objNumero2 = new Integer(65);
                Character objCaracter = new Character('A');
                int numero;  char caracter;
                numero = objNumero.intValue(); caracter
                = objCaracter.charValue();
                System.out.println(objNumero.toString());
                System.out.println(objCaracter.toString());
                if (objNumero.equals(objNumero2))
                        System.out.println("Son Iguales");
                else
                        System.out.println("No son iguales");

                char ch; ch = 'a';
                if (Character.isUpperCase (ch)) {
                        System.out.println(ch+" ES MAYUSCULA");
               }else{   System.out.println(ch+" NO ES MAYUSCULA");
                 ch=Character.toUpperCase(ch);

                System.out.println("AHORA SI: "+ch);
                }
        }
}
```

La clase envoltorio de tipo Boolean

La clase Boolean tiene definidas internamente dos variables estáticas y públicas:

```
public static final Boolean TRUE = new Boolean(true);
public static final Boolean FALSE = new Boolean(false);
```

Estas sentencias definen TRUE y FALSE como objetos que contienen los valores booleanos de verdadero y falso.

La palabra reservada *public* hace que los objetos TRUE y FALSE estén disponibles para ser usados fuera de la clase Boolean.

La palabra *static* hace que las variables estén disponible para todas las instancias de esa clase. En otras palabras, todas las instancias del objeto compartirán la misma copia del objeto estático. Como la variable no está asociada a una instancia en particular, se puede acceder a ella sin definir una instancia de la clase. Ejemplo:

El uso de la palabra reservada *final* hace que la variable sea constante.

La clase envoltorio Character

La clase envoltorio Character, además de los métodos comunes, provee métodos adicionales que, en todos los casos, son estáticos. Por ello, pueden ser llamados sin crear una instancia del objeto de la clase Character mediante el uso de la sintaxis siguiente:

```
public static boolean isLowerCase(char ch); public static
   boolean isUpperCase(char ch); public static boolean
   isDigit(char ch);
public static char toLowerCase(char ch);
public static char toUpperCase(char ch);
public static char isSpace(char ch); // verdadero si se trata
   de un carácter de espacio como tabulador, nueva línea,
   etc.
```

Las clases envoltorio de tipo Integer, Long, Float y Double

```
public class Numeros
{
	public static void main (String[] args)
	{
		System.out.println("MINIMO INTEGER: " + Integer.MIN_VALUE);
		System.out.println("MAXIMO INTEGER: " +
		Integer.MAX_VALUE);
		System.out.println("MINIMO LONG: " + Long.MIN_VALUE);
		System.out.println("MAXIMO LONG: " + Long.MAX_VALUE);
		System.out.println("MINIMO FLOAT: " + Float.MIN_VALUE);
		System.out.println("MAXIMO FLOAT: " + Float.MAX_VALUE);
		System.out.println("MINIMO DOUBLE: " + Double.MIN_VALUE);
		System.out.println("MAXIMO DOUBLE: " + Double.MAX_VALUE);
	}
}
```

Con las constantes MIN_VALUE y MAX_VALUE se puede averiguar el rango de los tipos de datos integer, long, float y double.

valueOf y las clases de envoltorio

```
public class Conversion {
	public static void main (String[] args) {
		int numero;
		numero = Integer.valueOf("235").intValue();
	}
}
```

El método valueOf se emplea para devolver un objeto de clase envoltorio dado un valor numérico de cadena. Se utiliza para convertir una cadena en un número. Hay que recordar que el método valueOf del objeto String hacia justamente lo contrario (convertir un valor a cadena).

Tema 8: Los arrays en Java

Introducción

Un array es básicamente un conjunto de elementos del mismo tipo que comparten el mismo nombre y pueden ser tratados uniformemente desde el programa.

En Java, a diferencias de C++, los arrays no son simples áreas de memoria, sino objetos y pueden ser tratados como tal. Cuando se crean arrays en memoria se instalan a partir de subclases de la clase Object.

Declaración

<TipoDato> <NombreArray>[];

o

<TipoDato>[] <NombreArray>;

Los siguientes ejemplos son declaraciones de arrays:

```
long saldoMedio[]; String nombres[]; int valores[];
```

Hay que tener en cuanta que la declaración del array en ningún momento supone la creación en memora para sus elementos. Los lo cual una vez declarado, habrá que reservar espacio para sus elementos de la siguiente forma:

```
saldoMedio = new long[4];
nombres = new String[3];
valores = new int[4];
```

Asignar valores a los elementos del array:

```
nombres[0]=”Pepe”;
nombres[1]=”Juan”;
nombres[2]=”Miguel”;
```

Recorrer un array:

```
for (int i=0; i<nombres.length;i++)
   System.out.println(nombres[i]);


for (String nombre : nombres)
   System.out.println(nombre);
```

En java, al contrario de la mayoría de los lenguajes no se puede indicar el número de elementos en la propia declaración del array. Las siguientes declaraciones serían erróneas:

~~*int ventas[4];*~~
~~*int[12] ventas;*~~

Si serían correctas las siguientes sentencias :

int ventas[] = new int[4];
int[] ventas = new int[4];

En este caso estamos declarando el array, pero también lo estamos creando y en esta creación si podemos indicar el número de elementos.

Otra forma de crear un array:

```
String
   meses[]={"Enero","Febrero","Marzo","Abril","Mayo","Junio",
   "Julio","Agosto","Septiembre","Octubre","Noviembre","Dicie
   mbre"};
for (int i=0;i<meses.length;i++)
   System.out.println(meses[i]);
```

Arrays de dos dimensiones:

Cada fila puede tener distinto número de elementos.

```
// Ejemplo de una matriz
public class Matrices {
        public static void main(String args[]) {
                int tabla[][];
```

```
                tabla = new int[4][];
                tabla[0]=new   int[3];
                tabla[1]=new   int[2];
                tabla[2]=new   int[5];
                tabla[3]=new int[13];

                for (int i=0;i<tabla.length;i++)
                {
                        System.out.println();
                        for (int j=0;j<tabla[i].length;j++)
                                System.out.print(tabla[i][j]);
                }
        }
}
```

Excepción de salida de los límites del array:

Si se sobrepasan los límites del array, se produce la siguiente excepción (error en tiempo de ejecución), aunque más adelante se estudiarán las excepciones.

java.lang.ArrayIndexOutOfBoundsException

Tema 9: Programación Orientada a Objetos con Java

Definición de clases en java

El siguiente ejemplo muestra la implementación de la clase Room, en ellas nos basaremos para todas las explicaciones.

```
public class Room {
        // Variables de instancia de clase o
        // propiedades del objeto.
        int camas;
        boolean baño;
        String color;

        // Variables de clase.
        static int cuentaRooms=0;

        // Constantes de clase.
        static final int UNA=1;
        static final int DOS=2;
        static final int TRES=3;

        // Constructores
        Room() { camas=2;
                baño=true;
                color="Blanco";
                cuentaRooms++;
        }

        Room(int camas) {
                this.camas=camas;
                baño=true;
                color="Blanco";
                cuentaRooms++;
        }

        Room(int camas, boolean baño) {
                this.camas=camas;
                this.baño=baño;
                color="Blanco";
                cuentaRooms++;
        }

        Room(int camas, boolean baño, String color) {
                this.camas=camas;
                this.baño=baño;
                this.color=color;
                cuentaRooms++;
        }

        // Funciones de instancia de clase o métodos del objeto.
        public void caracteristicas() {
                System.out.println("Camas: "+camas);
                if (baño)
                        System.out.println("Con baño");
```

```
            else

                  System.out.println("Sin baño");
            System.out.println("Color: "+color);
}

public float precio(int edad) {
            if (edad<27)
                  return 1500f*camas;
            else
                  return 2000f*camas;
}
```

```
        public float precio() { // Sobrecarga de métodos.
                return 2000f*camas;
        }

        // Funciones estáticas o de clase.
        // no es necesario crearse un objeto (instancia)
        // para utilizarse.
        public static void informacion() {
                System.out.println("Nuestro hotel dispone de habitaciones");
                System.out.println("de 1, 2, 3, 4 y 5 dormitorios");
                System.out.println("casi todas con cocina y baño");
        }
}
```

Utilización de la clase Room.

```
public class UsoRoom {
        public static void main(String args[]) {
                // Declaración de un array de tres objetos Room.
                Room[] cuartos=new Room[3];
                float importe;
                // Construcción de los tres objetos llamando a constructores distintos.
                cuartos[0] = new Room(3,true,"Rojo");
                cuartos[1] = new Room();
                cuartos[2] = new Room(5,false);
                // Utilización de una constante de clase.
                cuartos[1].camas=Room.DOS;
                for (int i=0;i<cuartos.length;i++) {
                        // Acceso a los métodos de la clase.
                        cuartos[i].caracteristicas();
                        importe=cuartos[i].precio();
                        System.out.println("Precio habitación: "+importe);
                        System.out.println("--------------------");
                }
                // Uso a un método estático (acceso por el nombre de la clase).
                System.out.println("Número total de cuartos: "+Room.cuentaRooms);
                Room.informacion();
        }
}
```

Estructura de la clase:

```
class nombre-clase {
        // definiciones de variables de clase
        // métodos de clases
}
```

Variables de instancia de clase y variables de clase

Las variables de instancia de clase para objetos separados tienen existencia separada. Una variable de clase, sin embargo, es común para todas las instancias de la clase. Estas variables están precedidas por la

palabra clave *static*. Esto es útil para cuando se necesita algún contador global que cuente el nº de veces que se ha utilizado la clase.

Las variables de clase, por no estar asociadas a ninguna instancia, van precedidas del nombre de la clase y no del nombre del objeto de dicha clase.

```
// Variables de instancia de clase o
// propiedades del objeto.
int camas;
boolean baño;
String color;

// Variables de clase.
static int cuentaRooms=0
```

Constantes de clases

Una constante de clase de declara con la palabra final y a menudo viene también acompañada de la palabra clave static para que pueda ser utilizada sin necesidad de crear una instancia a la clase.

```
// Constantes de clase.
static final int UNA=1;
static final int DOS=2;
static final int TRES=3;
```

Implementación de los métodos de la clase

```
calificadores tipo-retorno nombre_método(parámetros) {
        .............
        [return expresión;]
}
```

Calificadores: son palabras tales como static, public, etc.. Son opcionales y controlan cómo se usa el método y cómo se puede ser accedido por el resto del código del programa Java.

Tipo-retorno: es el tipo de dato del valor que devuelve el método. El tipo de dato puede ser cualquiera de los tipos básicos de Java o el nombre de una clase o la palabra reservada void. Si se usa void, indica que no retorna ningún valor.

return expresión: El tipo de dato de la expresión debe coincidir con el tipo- retorno. Si el tipo-devuelto es void, no tiene que usar ninguna sentencia return. El método saldrá cuando se ejecute su última sentencia. Si se necesita terminar un método de tipo void antes de la última sentencia, se puede usar la palabra reservada return sin ninguna expresión de retorno.

parámetros: los parámetros son opcionales y se utilizan para pasar datos al método. SI el método no necesita parámetros, se codifica el método con una lista vacía de parámetros.

Sobrecarga de métodos

Pueden existir varios métodos con el mismo nombre pero con diferentes argumentos y tipo de retorno. Esto permite que un mismo método pueda adoptar diferentes formas (polimorfismo).

En la siguiente clase, el método precio está sobrecargado, puede ser llamado pasando como argumento la edad o sin argumento.

```
public float precio(int edad) {
        if (edad<27)
                return 1500f*camas;
        else
                return 2000f*camas;
}
```

```
public float precio() {
        return 2000f*camas;
}
```

Mecanismo del pase de parámetros a los métodos

En Java, el pase de parámetros se efectúa por medio de la técnica denominada "llamada por valor". Se llama así porque se pasa una copia del valor del parámetro, almacenado en una memoria separada del valor original del parámetro.

Uso de this

Java provee una palabra reservada *this* para hacer referencia a la instancia del propio objeto de la clase. Cuando se escribe un método, no se sabe de anticipado el nombre del objeto. El uso de la palabra this puede, en muchos casos, simplificar la escritura de los métodos de una clase.

```
Room(int camas, boolean baño, String color) {
        this.camas=cama
        s; this.baño=baño;
        this.color=color;
        cuentaRooms++;
}
```

Métodos constructores

Son métodos especiales cuya misión principal es construir una instancia de la clase. Puede ser que se tenga que inicializar las variables de la clase y las variables de instancia.

Esta es la sintaxis general de un método constructor de una clase:

```
nombre-clase (parámetros) {

}
```

Puede haber sobrecarga de constructores.

```
Room(int camas) {
        this.camas=camas;
        baño=true;
        color="Blanco";
```

```
        cuentaRooms++;
}

Room(int camas, boolean baño) {
        this.camas=camas;
        this.baño=baño;
        color="Blanco";
        cuentaRooms++;
}
```

Cuando se crea la instancia a la clase (objeto) se llama al constructor que corresponda según el número de argumentos que se pasen.

```
Room[] cuartos=new Room[3];
float importe;
cuartos[0] = new Room(3,true,"Rojo");
cuartos[1] = new Room();
cuartos[2] = new Room(5,false);
```

En este ejemplo se ha creado un array de 3 objetos Room llamando a tres constructores distintos.

Métodos estáticos

Si se declara un método con la palabra static, este método puede ser llamado sin necesidad de crear una instancia a la clase, al igual que las variables. Se trata de métodos de clase y no de instancia de clase.

```
public static void informacion() {
        System.out.println("Nuestro hotel dispone de habitaciones");
        System.out.println("de 1, 2, 3, 4 y 5 dormitorios");
        System.out.println("casi todas con cocina y baño");
}
```

Este método puede ser llamado de la siguiente forma:

```
Room.informacion();
```

Tema 10: La herencia en java

Cuando una clase E hereda otra clase S, la clase S se denomina Superclase y la clase E se denomina subclase o clase extendida.

Sintaxis de clase extendida:

```
class E extends S {
        // definición de la clase E
}
```

En el ejemplo de la pagina siguiente vamos a crear una clase llamada RentRoom (alquiler de habitación) que va a heredar todas las características de la clase Room.

```
import java.util.Date;

public class RentRoom extends Room
        { String nombreCliente;
        int diasAlquiler;
        private Date fecha=new Date();

        RentRoom()     {
                super(); // Llamada al constructor de la clase
                base. nombreCliente="Desconocido";
                diasAlquiler=1;
        }

        RentRoom(String nombre, int dias, int camas, boolean baño, String color) {
                super(camas,baño,color); // Llamada al constructor de la clase
                base. nombreCliente=nombre;
                diasAlquiler=dias;
        }

        // Sustitución de los métodos de la clase
        base. public void caracteristicas() {
                super.caracteristicas();
                System.out.println("Dias Aquiler: "+diasAlquiler);
        }

        public float precio(int edadCliente) {
                return diasAlquiler*super.precio(edadCliente);
        }

        public float precio() {
                return diasAlquiler*super.precio();
        }

        public String toString() {
                return "Objeto de la clase RentRoom, creado a fecha "+fecha.toString();
        }
}
```

Herencia y constructores

Al crear la clase derivada, hay que tener en cuenta también los constructores de la clase base.

Para llamar al constructor de la clase madre o superclase, se puede utilizar la siguiente declaración:

```
super(); o
super(argumentos);
```

```
RentRoom() {
        super(); // Llamada al constructor de la clase base.
        nombreCliente="Desconocido";
        diasAlquiler=1;
}

RentRoom(String nombre, int dias, int camas, boolean baño, String color) {
        super(camas,baño,color); // Llamada al constructor de la clase base.
        nombreCliente=nombre;
        diasAlquiler=dias;
}
```

La sentencia super() solo puede aparecer dentro de una clase extendida o clase hija.

También puede ser utilizada para llamar a un método de la clase base que tiene el mismo nombre que otro método en la clase derivada.

```
public float precio(int edadCliente) {
        return diasAlquiler*super.precio(edadCliente);
}

public float precio() {
        return diasAlquiler*super.precio();
}
```

Dentro de la clase derivada, los métodos de la clase base pueden ser sobreescritos o no según convenga.

Uso de métodos en el objeto raíz

En Java, la clase incorporada Object es la superclase de todos los objetos. Cuando se recorre la cadena de la jerarquía de clases para encontrar las superclases de un objeto, se llega finalmente al punto de la clase denominada Object.

La clase Object también se llama clase raíz y tiene métodos predefinidos, cuyas declaraciones se listan a continuación:

```
Object();
boolean equals(object obj);
Class getClass();
int hashCode();
void notify();
void notifyAll();
void wait(long timeout);
void wait(long timeout, int nanos);
void wait();
String toString();
Object clone();
Object finalize();
```

Reemplazo *del método toString();*

El método toString produce una visualización del objeto en formato de texto, esto tiene sentido en clases de envoltorio, pero para otras clases definidas por nosotros puede ser util escribir de nuevo el código para este método de la forma que más interese como en el siguiente ejemplo:

```
public String toString()
{
        return ("Habitación de "+camas+" camas ");
}
```

El calificador public asegura que el método toString() pueda ser llamado desde clases que no están en el mismo archivo.

Miembros privados

Si se quiere que una variable o un método sea inaccesible a cualquier otra clase, se debe declarar esa variable o método como private.

```
class Coordenada {
        private int x; private int y;

        Coordenada() {
        x=y=12;
        }

        int getX() {
                return x
            }
        void setX(int valorX) {
                x=valorX;
         }
        int getY() {
                return y
            }
        void setY(int valorY) {
                x=valorY;
         }
}
```

Los miembros dato x e y están encapsulados dentro de la clase coordenada y sólo pueden modificarse por medio de los métodos setX y setY o consultarse por medio de los métodos getX y getY.

Este sería un ejemplo de utilización de la clase Coordenada:

```
Coordenada2 miCoordenada = new Coordenada2();
miCoordenada.setX(3);

miCoordenada.setY(8);
System.out.println(miCoordenada.getX ());
System.out.println(miCoordenada.getY ());
```

Tema11: Las clases abstractas

Una clase abstracta es otra forma de lograr el polimorfismo. Están pensadas para crear otras clases derivadas que implementen sus métodos abstractos. Pensemos por ejemplo en el concepto abstracto de figura: puede ser un cuadrado, un circulo, un rectángulo, un polígono, etc.

Las clases cuadrado, rectángulo, circulo, polígono, etc. pueden tener métodos compartidos y otros con implementación diferente. Se podría calcular el área de todas estas figuras, pero la fórmula para cada una sería diferente.

En el ejemplo expuesto anteriormente, podríamos crear la clase abstracta Figura y sus clases derivadas cuadrado, rectángulo, circulo, polígono, etc.

Veamos un ejemplo con una clase abstracta llamada Animal:

```
/*
   Esta es una clase abstracta.
   Sólo puede ser usada para crear clases derivadas
*/

public abstract class Animal {
        // Variable de instancia de
        clase. String nombre;

        Animal(String n) { //
                Constructor. nombre =
                n;
        }

        // Métodos abstractos, deben ser implementados
        // en una clase derivada.
        abstract String morder(Animal ani);
        abstract String mover();

        // Método no abstracto, podrá ser o no implementado
        // en la clase derivada.
        public String toString()
        {
                return "Saludos desde Animal";
        }
}
```

Los métodos morder() y mover() son abstractos, necesitan obligatoriamente implementarse en las clases derivadas.

Para crear un método abstracto, es necesario que la clase sea abstracta.

El método toString() no es abstracto, por lo tanto podrá ser implementado en las clases derivadas o no.

Veamos cómo crear clases derivadas de la clase abstracta Animal:

```
/*
        Esta es una clase derivada que implementa una
        clase abstracta. Será obligatorio implementar los
        métodos abstractos de la clase base.
*/
public class Pulga extends Animal {
        Pulga() {
                super ("Pulga");
        }

        String morder(Animal ani) {
                return "Pulga muerde "+ani.nombre;
        }

        String mover() {
                return "Pulga se mueve";
        }

        public String toString() {
                return "Saludos de la pulga";
        }
}

public class Tiranosaurio extends Animal {
        Tiranosaurio() {
                super ("Tiranosaurio");
        }

        String morder(Animal ani) {
                return "Tiranosaurio muerde "+ani.nombre;
        }

        String mover()  {
                return "Tiranosaurio se mueve";
        }

        public String toString() {
                return "Saludos del tiranosaurio";
        }
}
```

Como usar las clases abstractas

No se puede crear una instancia de una clase abstracta. La sentencia siguiente ocasiona un error:

```
Animal MiAnimal = new Animal("Pulga");
```

Sin embargo si podría realizar la siguiente sentencia:

Animal MiAnimal = new Pulga();

En el siguiente ejemplo, vamos a crear un array con dos objetos de la clase animal a los cuales les vamos a asignar una pulga y un tiranosaurio respectivamente.

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

                Animal[] zoo =  {new Pulga(), new Tiranosaurio()};
                System.out.println(zoo[0].toString());
                System.out.println(zoo[0].morder(zoo[1]));
                System.out.println(zoo[0].mover());
                System.out.println(zoo[1].toString());
                System.out.println(zoo[1].morder(zoo[0]));
                System.out.println(zoo[1].mover());
        }
}
```

Tema 12: Las interfaces

Java no permite la herencia múltiple, la siguiente sentencia ocasionaría un error:

```
public class ClaseDerivada extends Clase1, Clase2
{
        // declaraciones de la clase
}
```

Para resolver este problema, se han creado las interfaces.

- Una interface especifica la forma de sus métodos, pero no da ningún detalle de su implementación; por lo tanto no se puede pensar en ella como la declaración de una clase.
- Una interface no puede tener constructor.
- Todos los métodos de una interface son abstracto, aunque no se incluya la palabra clave abstract. Por lo tanto, no se podrá incluir implementación en ninguno de los métodos.
- Las variables miembro definidas en la interface son por defecto finales aunque se pueden redefinir en las clases derivadas.
- Una interface se define con la palabra clave interface.

```
public interface Vehiculo {
        int VELOCIDAD_MAXIMA=120;

        String frenar(int cuanto);
        String acelerar(int cuanto);
}
```

Sus clases derivadas se declaran con la palabra clave implements. Vamos a crear las clases Moto y Coche implementando la clase Vehiculo.

```
public class Coche implements Vehiculo {
        int velocidad=0;

        public String acelerar(int cuanto) {
                String cadena="";
                velocidad=velocidad+cuanto;
                if (velocidad>VELOCIDAD_MAXIMA) cadena="Exceso de velocidad ";
                cadena=cadena+"El coche a acelerado y va a "+velocidad+" km/hora";
                return cadena;
```

```
	}
	public String frenar(int cuanto) {
		velocidad=velocidad-cuanto;
		return "El coche ha frenado y va a "+velocidad+" km/hora";
	}
}

public class Moto implements Vehiculo {
	int velocidad=0;

	public String acelerar(int cuanto) {
		String cadena="";
		velocidad=velocidad+cuanto;
		if (velocidad>VELOCIDAD_MAXIMA)
			cadena="Exceso de velocidad ";
		cadena=cadena+"La moto a acelerado y va a "+velocidad+" km/hora";
		return cadena;
	}

	public String frenar(int cuanto) {
		velocidad=velocidad-cuanto;
		return "La moto ha frenado y va a "+velocidad+" km/hora";
	}
}
```

Ahora podemos declarar un objeto de la clase Vehiculo y asignarle indistintamente una Moto o un Coche, de la misma forma que ocurría con las clases abstractas. La diferencia principal es que ahora se ha resuelto algo el problema de la herencia múltiple, ya que una clase derivada puede implementar dos interfaces o bien implementar de una y heredar otra clase.

```
public class UsoVehiculo {
	public static void main (String[] args) {
		Vehiculo v1 = new Moto();
		Coche v2 = new Coche();
		System.out.println(v1.acelerar(100));
		System.out.println(v1.frenar(25));
		System.out.println(v2.acelerar(130));
		System.out.println(v2.frenar(25));
	}
}
```

Colección garbage y gestión de memoria

En java, se cuenta con un proceso ya construido llamado colección garbage. Este proceso es automático y dispondrá de la memoria alocada que no tenga más referencias. Para que funcione la colección garbage, se puede poner un objeto a null.

Cuando un objeto está dentro del proceso de garbage, este recolector llama a un método del objeto llamado finalize, si existe.

```
protected void finalize() {
        System.out.println("Esto es algo así como el destructor");
}
```

Paquetes

Un paquete es un directorio donde se agrupan un conjunto de clases o librerías de clases.

Los nombres de los paquetes o directorio deben ir en minúsculas.

Para crear un paquete: package <nombre directorio>;

```
public class Ejemplo {
}
```

Para leer el paquete:

```
import <nombre paquete>
```

```
package mates; // este es el nombre de un directorio.

public class Circunferencia {// clase dentro del paquete mates.
        double radio;
        // variable por defecto protected, sólo accesible desde clases del mismo paquete.

        public Circunferencia(double r) {
                // constructor público para que sea accesible desde clases fuera del paquete.
                radio=r;
        }

        public double longitud() {
                return 2*Math.PI*radio;
        }

        public double area(){
                return Math.PI*radio*radio;
        }
}

package mates;

public class Triangulo {
        public int lado1; public int lado2; public int lado3;

        public Triangulo(int l1, int l2, int l3) {
                lado1=l1;
                lado2=l2;
                lado3=l3;
        }
```

```
        public String tipo() {
                if (lado1==lado2 && lado2==lado3) return "Equilátero";
                else if (lado1==lado2 || lado2==lado3 || lado1==lado3) return "Isósceles";
                else return "Escaleno";
        }

}


import mates.Triangulo;
import mates.Circunferencia;

public class UsoMates {
        public static void main(String args[]) {
                Triangulo t = new
                Triangulo(3,2,1);
                System.out.println(t.tipo());
                Circunferencia c = new Circunferencia(12);
                double l=c.longitud();
                double a=c.area();
                System.out.println("Longitu
                d="+l);
                System.out.println("Area="+
                a);
        }
}
```

Modificadores de acceso para variables y métodos

Establecen el tipo de protección (visibilidad o acceso) de una variable, método o clase.

public	*accesible desde cualquier clase.*
protected	*accesible sólo desde clases del mismo paquete.*
private	*accesible sólo dentro de la clase.*
Final	*no modificable. Para las variables, hace que sean constantes. Para los métodos, hace que no se puedan sobreescribir.*
Static	*define si el acceso al método o variable se hace por medio de un objeto o no. Se puede acceder con NombreClase.miembro.* *Se denominan variables o métodos de clase a diferencia de las demás que se denominan variables de instancia de clase.* *Un método estático sólo puede utilizar variables de clase estáticas.* *Dentro de un método estático se puede acceder a otro método estático, pero no a un método de instancia de clase.* *Las variables de clase static se definen antes de crear el objeto.*

Los objetos como referencia de memoria

El nombre de un objeto representa una dirección de memoria o referencia en memoria. Si igualamos un objeto con otro, en realidad los dos objetos son el mismo ya que están apuntando a la misma dirección de memoria.

Añadiendo a la clase UsoMates las siguientes sentencias, vemos que al crear un objeto llamado t2 e igualarlo con el objeto t, los cambios realizados en el objeto t2 se reflejan también en el objeto t.

```
Triangulo t2; t2=t; t2.lado1=1;
System.out.println(t);
```

Tema 13: Tratamiento de excepciones

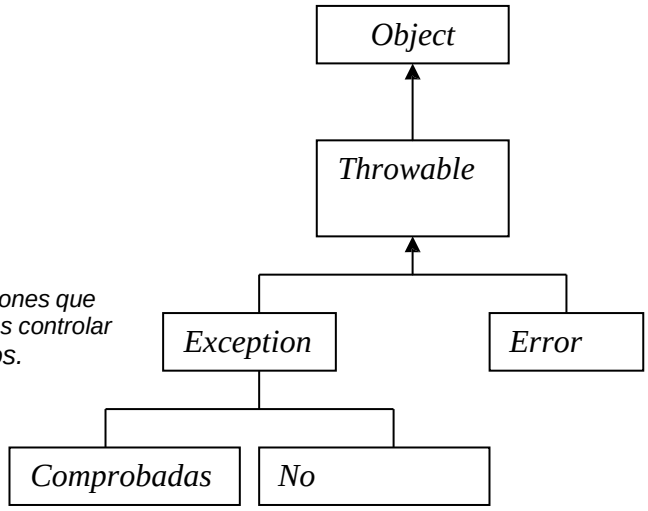
Una **excepción** es una **situación anómala** que puede provocar que el programa no funcione de forma correcta o termine de forma inesperada.

Ejemplos de situaciones que provocan una excepción:

- No hay memoria disponible para asignar
- Acceso a un elemento de un array fuera de rango
- Leer por teclado un dato de un tipo distinto al esperado
- Error al abrir un fichero
- División por cero
- Problemas de Hardware

Cuando se produce una condición de error, se crea un objeto que representa esa excepción y se envía al método donde se ha producido el error, para que este método trate el error.

Tipos de excepción



Excepciones que podemos controlar nosotros.

Las excepciones de tipo Error las utiliza el interprete de java cuando se utiliza un error interno como por ejemplo, un desbordamiento de pila.

Necesita que estén indicadas en el método throws como por ejemplo IOException.

Las controla java automáticamente, no tienen que estar escritas en el método.

Control de las excepciones. Bloque try-catch

Vamos a analizar el siguiente ejemplo:

```
public class DivCero {
        public static void main(String args[]) {
                int cero=0;
                int resul=6/cero;
        }
}
```

En el momento en que el código intenta dividir por cero, se detiene la ejecución del programa, creando un objeto excepción que es lanzado inmediatamente para ser tratado. Como no hemos programado ningún gestor de excepción propio, la excepción pasa al gestor de excepciones del interprete de java, que te proporcinará un tratamiento por defecto.

Para proporcionar un gestor de excepciones propio dentro de un bloque de código, se debe encerrar dicho código dentro de un bloque try. Inmediatamente después del bloque try, se debe incluir un bloque catch, indicando el tipo de error que se desea capturar y el código que se quiere ejecutar en caso de producirse dicho error.

```
public class DivCero {
	public static void main(String args[]) {
		int cero=0;
		try {
			int resul=6/cero;
		} catch (ArithmeticException e) {
			System.out.println("Se ha intentado dividir por cero");
		}//fin catch

		System.out.println("Final del programa");
	}
}
```

Múltiples cláusulas catch

```
public class DivCero {
	public static void main(String args[]) {
		int cero=0;
		int array[]=new int[3];
		int resul;
		for (int i=0; i<=3; i++) {
			try {
				if (i>1)
					resul=6/cero;
				else
					array[5]=23;
			}
```

```
                    catch (ArithmeticException e) {
                        System.out.println("Se ha intentado dividir por cero");
                    }//fin catch división por cero.
                    catch (ArrayIndexOutOfBoundsException e)
                        { System.out.println
                      ("Se ha sobrepasado el final del array");
                    }//fin catch fuera de índice array.
                }
                System.out.println("Final del programa");
        }
}
```

Clases de excepción personalizadas

Deben extender de la clase Throwable.

```
public class RaizNegativaException extends Throwable
        { RaizNegativaException(double raiz) {
                super("Raiz Negativa "+raiz); // Constructor de la clase Throwable.
        }
}
```

cláusula throw

Provoca un error en tiempo de ejecución creando un nuevo objeto de excepción.

```
public static double ecuacionX1(double a, double b, double c)    throws RaizNegativaException {

        double f=b*b-4*a*c;
        if (f<0)
                throw new RaizNegativaException(f);
        return (-b+Math.sqrt(f))/(2*a);
}
```

cláusula throws

Advierte de cuáles son las posibles excepciones generadas en un método, está cláusula va acompañando a la cabecera de la función.

```
public class UsoUtilidades {
        public static void main(String args[]) throws RaizNegativaExceptio        {
                ..............................
        }
}
```

Tema 14: Aplicaciones multitarea

Las tramas o hilos (concurrencia)

Una trama es un conjunto de instrucciones ejecutadas en una secuencia lógica dentro de un programa y representa un grupo de control dentro de un programa. Mediante el uso de tramas, se puede obtener el proceso paralelo y simultáneo dentro de un mismo programa.

Si su ordenador tiene un único procesador, la ejecución simultánea de las tramas no es estrictamente posible. Las tramas comparten el mismo procesador. En un determinado momento, sólo una trama se está ejecutando por vez, pero cuando lleva cierto tiempo de ejecución, su proceso se suspende y se pasa la ejecución a otra trama. El algoritmo de planificación de las tramas en un procesador es diferente según el sistema operativo y el entorno de ejecución.

En la animación de applets, a las tramas se le asignan tareas específicas, tales como pintar un área determinada del applet.

Existen dos maneras de utilizar las tramas en Java:

- Mediante la creación de una subclase de una clase predefinida Thread.
- Usando la interfaz Runnable.

Obtener el hilo principal

Todo programa se ejecuta al menos en un hilo, este hilo se puede capturar con el método currentThread() de la clase Thread.

```
public class HiloPrincipal {
        public static void main(String args[]) throws InterruptedException {
                Thread hiloPrincipal = Thread.currentThread();
                // El método estático currentThread() devuelve el hilo principal
                // de ejecución.

                // El hilo principal se llama main.
                System.out.println("Hilo Principal: "+hiloPrincipal.toString());

                  // Cambio el nombre y la prioridad del hilo principal.
                hiloPrincipal.setName("Hilo1");
                hiloPrincipal.setPriority(3);
                System.out.println("Hilo Principal: "+hiloPrincipal.toString());

                  //Duerme el hilo durante dos segundos.
                hiloPrincipal.sleep(2000);
                System.out.println("Hola");
                System.out.println(hiloPrincipal.getName());
                System.out.println(hiloPrincipal.getPriority())
                ;
        }
}
```

Crear un hilo extendiendo la clase Thread

```
public class Hilos extends Thread {

        Hilos(String nombre, int prioridad)         {
                super(nombre);
                setPriority(prioridad);
                   // El método start() comienza la ejecución del hilo llamando el método run().
                start();
        }

         // El método run() contiene la lógica del hilo.
        public void run()  {
                for (int i=1; i<=10;i++)
                {
                        System.out.println(i+ " "+this.toString());
                        try        {
                                this.sleep(500);
                        } catch (InterruptedException e) {
                                System.out.println(e.toString());
                        }//fin catch.
                }
        }
}
```

Crear un hilo implementando la interface Runnable

```
public class OtroHilo implements Runnable {
        // Necesitamos declarar un objeto de la clase Thread como variable de instancia de
clase.
        Thread h;

        OtroHilo(String nombre, int prioridad) {
                // Preparamos el objeto hilo y comenzamos la ejecución.
            h=new Thread(this, nombre);
            h.setPriority(prioridad);
            h.start();
        }


         // Aquí va la lógica del hilo
        public void run()
        {
                for (int i=1; i<=10;i++)
                {
                        System.out.println(i+ " "+h.toString());
                        try     {
                                h.sleep(500);
                        } catch (InterruptedException e) {
                                System.out.println(e.toString());
                        }//fin catch.
                }
        }
}
```

Utilización de los hilos

```
public class UsoHilos
{
	public static void main(String args[]) throws InterruptedException
	{
		Hilos hilo2 = new Hilos("Hilo secundario",7);
		Hilos hilo3 = new Hilos("Hilo tercero",10);
		for (int i=1; i<=10;i++)
		{
			System.out.println(i+ " "+Thread.currentThread().toString());
			Thread.currentThread().sleep(500);
		}

		OtroHilo hilo4 = new OtroHilo("Hilo cuarto",7);
		OtroHilo hilo5 = new OtroHilo("Hilo quinto",10);
	}
}
```

Estado de los hilos

Nuevo	Ya se ha creado el hilo pero no se ha llamado al método start(), es decir, no ha
Ejecutable	Un hilo está ejecutable una vez que se ha ejecutado sobre él el método start(). Un hilo ejecutable puede estar ejecutándose o no dependiendo del
Bloqueado	Un hilo ejecutable puede pasar a estado bloqueado si se ejecuta sobre él el método sleep() o el método wait(). También puede estar bloqueado por esperar
Muerto	Ha finalizado la ejecución del método run() y se ha llamado al método

Tema 15: Colecciones en Java

Una colección representa un grupo de objetos. Esto objetos son conocidos como elementos. Cuando queremos trabajar con un conjunto de elementos, necesitamos un almacén donde poder guardarlos. En Java, se emplea la interfaz genérica Collection para este propósito. Gracias a esta interfaz, podemos almacenar cualquier tipo de objeto y podemos usar una serie de métodos comunes, como pueden ser: añadir, eliminar, obtener el tamaño de la colección… Partiendo de la interfaz genérica Collection extienden otra serie de interfaces genéricas. Estas subinterfaces aportan distintas funcionalidades sobre la interfaz anterior.

La API de Colecciones (Collections) de Java provee a los desarrolladores con un conjunto de clases e interfaces que facilitan la tarea de manejar colecciones de objetos. En cierto sentido, las colecciones trabajan como los arreglos, la diferencia es que su tamaño puede cambiar dinámicamente y cuentan con un comportamiento (métodos) más avanzado que los arreglos.

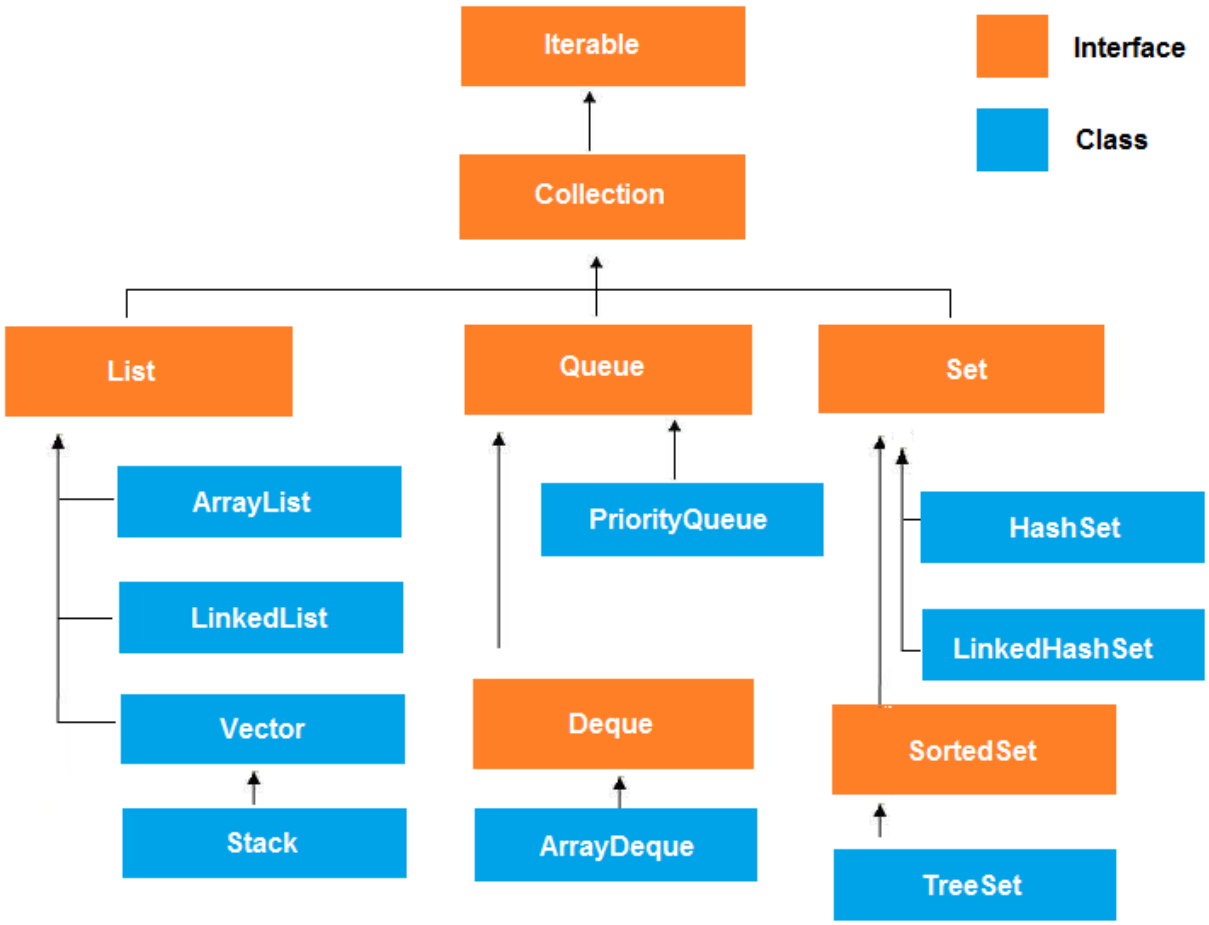
La mayoría de las colecciones de Java se encuentran en el paquete java.util

Tipos de colecciones en Java

Los tipos de colecciones vienen representados por interfaces. Una serie de interfaces clasifica las colecciones de Java en los siguientes tipos:

- **Listas**: Una lista ordenada, o secuencia. Normalmente permiten duplicados y tienen acceso aleatorio (es decir, puedes obtener elementos alojados en cualquier índice como si de un array se tratase).
- **Sets**: Colecciones que no admiten dos elementos iguales. Es decir, colecciones que no admiten que un nuevo elemento B pueda añadirse a una colección que tenga un elemento A cuando **A.equals(B)**.
- **Maps**: Colecciones que asocian un valor a una clave. Parecido a la estructura de "array asociativo" que se encuentra en otros lenguajes. Un Map **no puede tener dos claves iguales**.

- **Colas**: Colecciones que permiten crear colas LIFO o FIFO. No permiten acceso aleatorio, solo pueden tomarse objetos de su principio, final o ambos, dependiendo de la implementación.



Interfaz Iterable

La interfaz Iterable es una de las interfaces raíz de las clases de colecciones. La interfaz Collection hereda de Iterable, así que todos los subtipos de Collection también implementan la interfaz Iterable. Una clase que implementa la interfaz Iterable puede ser usada con el bucle foreach.

```
List list = new ArrayList();
for(Object o : list){
   //hacer algo con o;
 }
```

La interfaz Iterable únicamente tiene un método:

```
public interface Iterable {
   public Iterator iterator();
}
```

Interfaz Collection

La interfaz Collection es otra de las interfaces raíz de las clases de colecciones de Java. Aunque no es posible instanciar de manera directa una Colección, en lugar de eso se instancia un subtipo de Colección (una lista por ejemplo), de manera recurrente querremos tratar a estos subtipos de manera uniforme como una Colección. Las siguientes interfaces (tipos de colección) heredan de la interfaz Collection:

- List
- Set
- SortedSet
- NavigableSet
- Queue
- Deque

Sin importar el subtipo de Collection que se esté utilizando, existen algunos métodos comunes para agregar y quitar elementos de una colección:

```
String cadena = "Esto es una cadena de texto";
Collection coleccion = new HashSet();
boolean huboUnCambio = coleccion.add(cadena);
boolean seEliminoElemento = coleccion.remove(cadena);
```

- add() agrega el elemento dado a la colección y regresa un valor verdadero (true) si la colección cambió como resultado del llamado al método add(). Un conjunto (set) por ejemplo, podría no haber cambiado ya que, si el conjunto ya contenía el elemento, dicho elemento no se agrega de nuevo. Por otro lado, si el método add() fue llamado con una lista (List) y la lista ya contenía el elemento, dicho elemento se agrega de nuevo y ahora existirán 2 elementos iguales dentro de la lista.
- remove() quita el elemento dado y regresa un valor verdadero si el elemento estaba presente en la colección y fue removido. Si el

elemento no estaba presente, el método remove() regresa falso (false). También se pueden agregar y remover colecciones de objetos. Aquí hay algunos ejemplos:

Interfaz List. Listas

La colección más básica de Java. También, es la más usada por los programadores que no han investigado el framework de colecciones a fondo por hacernos pensar que se trata de una especie de *array hipervitaminado* ya que hace su trabajo y es fácil de entender.

Sabiendo los requisitos de tu aplicación, seguramente encontrarás otra colección que haga el trabajo de una lista mejor. Procura asegurarte de que cuando vas a usar una lista es porque realmente te hace falta, no porque no conoces el resto. La diferencia de rendimiento puede ser enorme. Y cuando digo enorme, me refiero a superior a un 600% en según qué operaciones.

¿Qué beneficios tienen las listas?

- Acceso aleatorio.
- Están ordenadas (Podemos usar **Colections.sort()** para ordenar los elementos siguiendo el criterio que queramos).
- Podemos añadir / eliminar elementos sin ninguna restricción.
- Tienen un iterador especial *ListIterator* que permite modificar la lista en cualquier dirección.
- Siguen la notación de los arrays, por lo que son fáciles de comprender.

¿Qué problemas tienen las listas?

- Bajo rendimiento en operaciones especializadas respecto a otras colecciones.

Tipos de Lista

- **ArrayList**: Muy rápida accediendo a elementos, relativamente rápida agregando elementos si su capacidad inicial y de crecimiento están bien configuradas. Es la lista que deberías usar casi siempre.

- **LinkedList**: Una lista que también es una cola (hablaré de esto más tarde). Más rápida que ArrayList añadiendo elementos en su principio y eliminando elementos en general. Utilízala en lugar de ArrayList si realizas más operaciones de inserción (en posición 0) o

de eliminación que de lectura. La diferencia de rendimiento es enorme.

- **Vector**: Terreno peligroso. Vector es una colección *deprecated* (obsoleta), así que usadla únicamente si necesitáis un ArrayList concurrente.
- **CopyOnWriteArrayList**: Colección concurrente que es muy poco eficiente en operaciones de escritura, pero muy rápida en operaciones de lectura.

Listas Genéricas

Por defecto se puede poner cualquier tipo de objeto en una lista, pero a partir de Java 5. "Java Generics" hace posible limitar los tipos de objeto que se pueden insertar en una lista, por ejemplo:

```
List<MyObject> lista = new ArrayList<MyObject>();
```

Esta lista ahora sólo puede tener instancias de la clase MyObject dentro de ella. Se puede entonces acceder e iterar estos elementos sin necesidad de hacer un casting:

```
MyObject unObjeto = lista.get(0);
for(MyObject unObjeto : lista){ //hacer algo con anObjeto...
   }
```

Interfaz Set.

Los sets, o conjuntos, son colecciones que por norma general no admiten elementos iguales en su interior. Como mencionaba antes, dos elementos A y B son iguales si A.equals(B).

Podemos añadir y eliminar elementos de un set, así como saber si determinado elemento está en un set, pero no tenemos acceso aleatorio, por lo que hay que tener muy claro cuando queremos usarlos.

Una colección normal, en apariencia, nos da todo lo anterior -excepto el hecho de eliminar duplicados-, así que, ¿por qué usar un Set y no una Lista?, el motivo es simple: **eficiencia**.

Supongamos este código:

```
List b = new ArrayList();
   if (!b.contains(1))
       b.add(1);
```

La funcionalidad es la misma que la de *set.add*, pero el rendimiento es muchísimo peor, hasta el punto de que podemos tener problemas muy serios cuando añadamos muchos elementos y no queremos eso, ¿verdad?.

¿Qué beneficios tienen los sets?

- No permiten elementos duplicados.
- Implementación muy eficiente de *.add* para asegurarnos de que no hay duplicados.

¿Qué desventajas tienen?

- No tienen acceso aleatorio.
- Solo algunos tipos de set pueden ordenarse y lo hacen de forma poco eficiente.

Tema 16: Enumerados

Introducción

Un enumerado (o Enum) es una clase "especial" (tanto en Java como en otros lenguajes) que limitan la creación de objetos a los especificados explícitamente en la implementación de la clase. La única limitación que tienen los enumerados respecto a una clase normal es que si tiene constructor, este debe de ser privado para que no se puedan crear nuevos objetos.

En las versiones anteriores a la versión 1.5 o 5 de Java no existían los tipos de datos enum con lo que debíamos usar constantes de la siguiente forma:

```
public static String COLOR_ROJO = "rojo";
public static String COLOR_VERDE = "verde";
public static String COLOR_AZUL = "azul";
```

A partir de la versión 5 de Java se incorporarón al lenguaje los tipos de datos enumerados con el objetivo de mejorar varios aspectos sobre el uso de las constantes. Básicamente, un enum en Java es un conjunto fijo y relacionado de constantes como pueden ser los días de la semana y deberían usarse siempre que se necesite representar un conjunto de constantes con esas características. La característica de relacionado es importante, las constantes solo lo están por la convención que sigamos al darles un nombre. Los enums se definen de la siguiente forma:

```
public enum Dia {
 LUNES, MARTES, MIERCOLES, JUEVES, VIERNES, SABADO, DOMINGO
}
```

Es muy importante entender que un "*Enum*" en java *es realmente una clase* (cuyos objetos solo pueden ser los definidos en esta clase: PORTERO, …, DELANTERO) *que hereda* de la clase "*Enum(java.lang.Enum)*" y por tanto los enumerados tienen una serie de métodos heredados de esa clase padre. A continuación, podemos ver algunos de los métodos más utilizados de los enumerados:

```
public enum Demarcacion{PORTERO, DEFENSA, CENTROCAMPISTA,
   DELANTERO}
Demarcacion delantero = Demarcacion.DELANTERO;    //
   Instancia de un enum de la clase Demarcación
delantero.name();    // Devuelve un String con el nombre de
   la constante (DELANTERO)
delantero.toString();    // Devuelve un String con el nombre
   de la constante (DELANTERO)
delantero.ordinal();    // Devuelve un entero con la
   posición del enum según está declarada (3).
delantero.compareTo(Enum otro);    // Compara el enum con el
   parámetro según el orden en el que están declarados lo
   enum
Demarcacion.values();    // Devuelve un array que contiene
   todos los enum
```

Un enum es una clase especial que limita la creación de objetos a los especificados en su clase (por eso su constructor es privado, como se ve en el siguiente fragmento de código); pero estos objetos pueden tener atributos como cualquier otra clase.

En la siguiente declaración de la clase, vemos un ejemplo en la que definimos un enumerado "Equipo" que va a tener dos atributos; el nombre y el puesto en el que quedaron en la liga.

```
public enum Equipo
{
        BARÇA("FC Barcelona",1), REAL_MADRID("Real Madrid",2),
        SEVILLA("Sevilla FC",4), VILLAREAL("Villareal",7);

        private String nombreClub;
        private int puestoLiga;

        private Equipo (String nombreClub, int puestoLiga){
                this.nombreClub = nombreClub;
                this.puestoLiga = puestoLiga;
        }

        public String getNombreClub() {
                return nombreClub;
        }

        public int getPuestoLiga() {
                return puestoLiga;
        }

}
```

Como se ve en la clase definimos un constructor que es privado (es decir que solo es visible dentro de la clase Equipo) y solo definimos los métodos "get". Para trabajar con los atributos de estos enumerados se hace de la misma manera que con cualquier otro objeto; se instancia un objeto y se accede a los atributos con los métodos get. En el

siguiente fragmento de código vamos a ver como trabajar con enumerados que tienen atributos:

```
// Instanciamos el enumerado
Equipo villareal = Equipo.VILLAREAL;

// Devuelve un String con el nombre de la constante
System.out.println("villareal.name()= "+villareal.name());

// Devuelve el contenido de los atributos
System.out.println("villareal.getNombreClub()=
   "+villareal.getNombreClub());
System.out.println("villareal.getPuestoLiga()=
   "+villareal.getPuestoLiga());
```

Como salida de este fragmento de código tenemos lo siguiente:

```
villareal.name()= VILLAREAL
villareal.getNombreClub()= Villareal
villareal.getPuestoLiga()= 7
```

Es muy importante tener claro que los enumerado no son Strings (aunque pueden serlo), sino que son objeto de una clase que solo son instanciables desde la clase que se implementa y que no se puede crear un objeto de esa clase desde cualquier otro lado que no sea dentro de esa clase. Es muy común (sobre todo cuando se esta aprendiendo que son los enumerados) que se interprete que un enumerado es una lista finita de Strings y en realidad es una lista finita de objetos de una determinada clase con sus atributos, constructor y métodos getter aunque estos sean privados.

Tema 17: JDBC. Acceso a datos desde aplicaciones Java

Introducción

Para establecer el acceso a una base de datos en java necesitamos:

- Un controlador o driver que comunique nuestra aplicación con un gestor de bases de datos como Access, SQL Server u Oracle. JDK trae un driver que permite conectar a través de ODBC. Este es "sun.jdbc.odbc.JdbcOdbcDriver".

- Un objeto Connection que nos permita establecer la conexión con una base de datos concreta.

- Un objeto Statement que representa una sentencia SQL que podrá ser de acción o selección.

```
import java.sql.*;

public class CrearTablasBaseDatos {
        public static void main (String args[]) {
                Connection conexion;
                Statement sql=null;
                try {
                        /* Este método carga el driver de acceso para la base de datos al
                        ejecutar el método forName() indicando el nombre del driver.
                          En concreto, este es el driver que trae por defecto JDK y
                         aprovecha un driver ODBC para realizar la conexión. */
                        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
                        try {
                                // Establece la conexión con una fuente de datos ODBC.
                                conexion=DriverManager.getConnection
                                ("jdbc:odbc:Amelia");
                                System.out.println("Conexión establecida");
                                // Devuelve un objeto Statement.
                                sql=conexion.createStatement();
                                // Ejecutar una sentencia SQL.
                                sql.executeUpdate ("CREATE
                                   TABLE Libros (idLibro Counter
                                   primary Key, Titulo char(30),
                                   Autor char(30),
                                     Genero char(30), Precio Double)");
                                System.out.println("TABLA CREADA");
                                conexion.close();
                                System.out.println("CERRADA LA CONEXION");
                        } catch (SQLException e) {
                                System.out.println
                                      ("NO SE PUEDE CREAR LA TABLA ");
                        } // Fin catch.
                } catch (ClassNotFoundException e) {
                        System.out.println("Error del controlador ");
                } // Fin catch.
        } // Fin main
} // Fin clase
```

DriverManager

Accede a un driver previamente cargado en memoria.

Por medio del método getConnection() podemos establecer la conexión con una base de datos a través del driver cargado.

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
conexion=DriverManager.getConnection("jdbc:odbc:Amelia");
```

Connection

Representa la conexión a una base de datos específica. public Statement createStatement()

```
                    throws
                SQLException

public Statement createStatement(int
                    resultSetType, int
                    resultSetConcurrency)
                throws
```

SQLException resultSetType:

ResultSet.TYPE_FORWARD_ONLY Cursor que permite un sólo recorrido y hacia delante.

ResultSet.TYPE_SCROLL_INSENSITIVE Cursor que puede recorrerse varias veces y en cualquier dirección, pero que no es sensible a los cambios de otros usuarios (estático).

ResultSet.TYPE_SCROLL_SENSITIVE Cursor que puede recorrerse varias veces y en cualquier dirección, sensible a los cambios que realizan otros usuarios.

resultSetConcurrency:

ResultSet.CONCUR_READ_ONLY Cursor de sólo lectura.

ResultSet.CONCUR_UPDATABLE Cursor que permite actualización.

public void close()

throws SQLException

Cierra la conexión con la base de datos.

public boolean isClosed()
throws SQLException

Averigua si la conexión está abierta o

cerrada.

public String getCatalog()
throws SQLException

Devuelve una cadena con el nombre del catálogo (SQL Server) o nombre y ubicación de la base de datos (Access).

public void commit()
throws SQLException

Realiza todos los cambios efectuados desde el último commit() o

rollback().

public void rollback()
throws SQLException

Anula todos los cambios efectuados desde el último commit() o

rollback().

public void setAutoCommit(boolean autoCommit)
throws SQLException

Si se establece a true crea una transacción automática por cada operación de actualización.

public boolean getAutoCommit()
throws SQLException

Obtiene el tipo de transacción.

ResultSet (Métodos relacionados con la posición del Cursor)

public boolean absolute(int row)

throws SQLException

Si pasamos como argumento un valor positivo, por ejemplo 3, se mueves tres posiciones desde delante del primer registro, accediendo así al tercer registro. Si pasamos como argumento un número negativo, por ejemplo –3, se mueve tres posiciones desde el final del último registro.

public void afterLast()

throws SQLException

Se mueve por detrás el último registro (End-Of-File).

public void beforeFirst()

throws SQLException

Se mueve por delante del primer registro (Botton-

Of-File).

public boolean first()

throws SQLException

Se posiciona en el primer registro.

public boolean isAfterLast()

throws SQLException

Devuelve true si estamos en End-Of-

File.

public boolean isBeforeFirst()

throws SQLException

Devuelve true si estamos en Botton-Of-

File. public boolean isFirst()

throws SQLException

Devuelve true si estamos en el primer registro.

public boolean isLast()

throws
SQLException

Devuelve true si estamos en el último registro.

public boolean last()

throws SQLException

Coloca el puntero en el último registro.

public boolean next()

throws SQLException

Coloca el puntero en el siguiente registro.

public boolean previous()

throws SQLException

Coloca el puntero en el registro

anterior.

public boolean relative(int rows)
throws SQLException

Si pasamos como argumento un valor positivo, por ejemplo 3, mueve el puntero tres posiciones hacia delante.

Si pasamos un valor negativo, por ejemplo –3, mueve el puntero tres posiciones hacia atrás.

Métodos para obtener el valor de un campo del registro activo

Estos métodos retornan el valor de uno de los campos en el registro activo por su nombre o posición.

public boolean getBoolean(int columnIndex) throws SQLException
public boolean getBoolean(String columnName) throws
SQLException public byte getByte(int columnIndex) throws

SQLException
public byte getByte(String columnName) throws SQLException
public byte[] getBytes(int columnIndex) throws SQLException
public byte[] getBytes(String columnName) throws
SQLException public Date getDate(int columnIndex)

throws SQLException

public Date getDate(String columnName)

throws SQLException

public double getDouble(int columnIndex)

throws SQLException

public double getDouble(String columnName)

throws SQLException

public float getFloat(int

columnIndex)

throws SQLException

public float getFloat(String columnName)

throws SQLException

public int getInt(int

columnIndex)

throws SQLException

public int getInt(String columnName)

throws SQLException

public long getLong(int columnIndex)

throws SQLException

public long getLong(String columnName)

throws SQLException

public short getShort(int columnIndex)

throws SQLException

public short getShort(String columnName)

throws SQLException

public String getString(int columnIndex)

throws SQLException

public String getString(String columnName)

throws SQLException

public Time getTime(int

columnIndex)

throws SQLException

public Time getTime(String columnName)

throws SQLException

Otros métodos

public void close() throws SQLException

Cierra el cursor y libera los recursos que este está empleando.

Un objeto ResultSet es cerrado de forma automática al cerrar el objeto Connection que lo creo o cuando este ejecuta una nueva sentencia.

public int getRow() throws SQLException

Devuelve la posición del registro actual dentro del cursor o conjunto de registros.

public int getType()

throws SQLException

Retorna uno de estos valores:

ResultSet.TYPE_FORWARD_ONL

Y
ResultSet.TYPE_SCROLL_INSENSITIVE
ResultSet.TYPE_SCROLL_SENSITIVE

public void refreshRow()

throws SQLException

Actualiza los datos del registro actual según el contenido en la base de datos.

Tema 18: Expresiones Lambda

Introducción

Una expresión Lambda es una función anónima, básicamente es un método abstracto es decir un método que sólo está definido en una interfaz, pero no implementado, y esa es la clave de la funciones lambda, al no estar implementado, el programador lo puede implementar dónde el crea conveniente sin haber heredado de la interfaz.

Sintaxis

La sintaxis de una expresión lambda suele darse de las siguiente manera:

(argumentos)->{cuerpo}

Por ejemplo:

(arg1, arg2…) -> { cuerpo}

Esta sintaxis puede cambiar tanto como para los argumentos o el cuerpo de la expresión Lambda de acuerdo con lo siguiente.

PARA LOS ARGUMENTOS:

Los argumentos de una función Lambda pueden ser declarados explícitamente o a su vez pueden ser inferidos por el compilador de acuerdo al contexto, cuando digo de acuerdo al contexto, me refiero a que si el método que estamos implementado recibe una cadena el compilador asumirá que el argumento es una cadena y así con el tipo de dato que estuviéramos recibiendo en el método.

Entonces un argumento se puede declarar explícitamente, esto se refiere al tipo de dato, por ejemplo: (int x)-> {cuerpo} y si tiene más parámetros sería (int x, int y….)-> {cuerpo}.

También se lo puede hacer de forma implícita, por ejemplo:

(x)-> {cuerpo}, de echo si existe un sólo argumento y se lo declara de forma implícita puede ir incluso sin paréntesis, por ejemplo

x -> {cuerpo}, si se declara más de un parámetro obligatoriamente deben ir seguido de comas y entre paréntesis.

Por último, puede haber expresiones Lambda en las que no haya argumentos y se expresan de la siguiente forma:

()-> {cuerpo}

PARA EL CUERPO DE LA EXPRESIÓN:

El cuerpo de la expresión puede ir o no dentro de llaves esto puede variar como se ve en los siguientes ejemplos.

Es obligatorio que el cuerpo de una expresión vaya entre llaves en los siguientes casos:

Cuando devuelve más de un valor o la sentencia tiene más de una instrucción, por ejemplo:

(int a, int b) -> { return a + b; }.

() -> { return 3.1415 }.

Mientras que cuando devuelve un sólo valor no es obligatorio, de todas maneras, el compilador no muestra error si se pone entre llaves por ejemplo, (aunque hay algunas excepciones):

() -> 10

n -> System.out.print(n + ” “)

Pero se la puede encerrar dentro de llaves, para esto es necesario añadir al final de la sentencia un punto y coma:

n -> {System.out.print(n + ” “);}

() -> new ArrayList<Integer>().

Interfaz funcional

Bien, las expresiones Lambda van de la mano con las **interfaces funcionales**, el concepto de interfaz funcional es añadido con la versión de Java 8 dada la necesidad de las expresiones Lambda.

Una interfaz funcional guarda el mismo concepto que una interfaz en las anteriores versiones de Java, salvo que se añade 2 reglas y es que para que una interfaz sea funcional debe:

- Tener un sólo método abstracto.

- Una interfaz funcional debe implementar los métodos dentro la misma interfaz (esto no se podía hacer en versiones anteriores), para esto se debe anteponer la palabra reservada **default** al inicio de la declaración del método.

Nota: Claro está que sólo un método debe ser abstracto, si la interfaz tuviera más métodos estos deben implementarse dentro de la misma.

Así mismo, aunque es opcional se puede declarar la anotación @FunctionalInterface, esta anotación le indica al compilador que esta es una interfaz funcional.

Veamos un ejemplo sencillo, una interfaz funcional que declara un método para sumar dos números enteros y que retorna su valor, si te das cuenta el método sólo se declara.

```
@FunctionalInterface
public interface IFuncionLambda {
    //método abstracto para sumar 2 números, que lo implementará
    //el programador a partir de una expresión Lambda
    public void suma(int a, int b);
}
```

Ahora viene la implementación del método sumar, en este caso los parámetros estan declarados de forma implícita (a, b) de manera que el compilador empareje el tipo de dato con el que se encuentra en el método sumar.

Luego tenemos el cuerpo que es la implementación del método y que viene a ser { System.out.println(a + b); }.

```
public class TestLambdas {

    public static void main(String[] args) {

      int x = 10;
      int y = 5;

      //se implementa el método de la interfaz con una expresión
    //lambda
      IFuncionLambda iflambda = (a, b) -> {
            System.out.println(a + b);
      };
      //se utiliza el método con la implementación y se le envía los
    //valores de x e y
      iflambda.suma(x, y);
    }
}
```

En conclusión, una expresión Lambda puede utilizarse donde haya una interfaz funcional que tenga la declaración del método que utiliza la expresión.

Tipos de expresiones Lambda

Las expresiones Lambda se pueden dividir de la siguiente manera:

- Predicados
- Funciones
- Proveedores
- Consumidor
- Referencia a métodos

Predicados

Los predicados son expresiones que reciben un argumento y devuelven un valor lógico, por ejemplo, se usa la interface Predicate<T>:

```
Predicate<String> predicate = (s) -> s.length() > 0;
        //evalua si la cadena "predicado" es mayor a 0
System.out.println(predicate.test("predicado")); // true
        //niega la valor de la evaulación
    System.out.println(predicate.negate().test("predicado")); //
    false
```

En el siguiente ejemplo a partir de una lista de números enteros se imprime: los números pares, los números mayores a 5 y los impares.

```
import java.util.Arrays;import java.util.List;
import java.util.function.Predicate;

public class Predicados {

public static void main(String[] args) {
    List<Integer> listaNumeros = Arrays.asList(1, 2, 3, 4, 5,
   6, 7,8,9,10);

    System.out.println("Números pares:");
    evaluar(listaNumeros, (n)-> n%2 == 0 );

    System.out.println("Números impares:");
    evaluar(listaNumeros, (n)-> n%2 == 1 );

    System.out.println("Números mayores a 5:");
    evaluar(listaNumeros, (n)-> n > 5 );

  }
  public static void evaluar(List<Integer> listaNumeros,
  Predicate<Integer> predicado) {
    for(Integer n: listaNumeros)  {
        if(predicado.test(n)) {
            System.out.print(n + " ");
        }
    }
```

```
        System.out.println();
    }
}
```

Funciones

Las funciones reciben un argumento y devuelven un resultado, usan la interface Function<T,R>, revisemos un ejemplo sencillo.

```
Function<Integer, Integer> suma = x -> x + 8;
System.out.println("La suma de 5 + 8: " + suma.apply(5));
```

Podemos también encontrar el tamaño de una cadena por ejemplo:

```
Function<String, Integer> tamanioCadena = str -> str.length();
String cadena = "Lambdas tipo funciones";
System.out.println("Número de caracteres es : " + tamanioCadena.apply(cadena);
```

Proveedores

Las expresiones Lambda de este tipo no tiene parámetros de entrada, pero si devuelven un resultado, utilizan la interface Supplier<T>.

Veamos un ejemplo sencillo, que básicamente obtiene la cadena enviada a la interface funcional, a través de una expresión Lambda tipo proveedor.

```
Supplier<String> cadena = () -> "Ejemplo de Proveedor";
System.out.println(cadena.get());
```

Un ejemplo un poco más detallado, primero creamos una clase Persona.

```
public class Persona {
   private String nombre;
   private String apellido;
   private String direccion;


   public Persona(String nombre, String apellido, String
   direccion) {
      this.nombre = nombre;
      this.apellido = apellido;
      this.direccion = direccion;
   }
   public String getNombre() {
      return nombre;
   }
   public void setNombre(String nombre) {
      this.nombre = nombre;
   }
   public String getApellido() {
      return apellido;
   }
   public void setApellido(String apellido) {
```

```
            this.apellido = apellido;
        }
        public String getDireccion() {
            return direccion;
        }
        public void setDireccion(String direccion) {
            this.direccion = direccion;
        }
}
```

Se implementa la expresión Lambda que es de tipo proveedor utilizando la clase Supplier.

```
    mport java.util.function.Supplier;

    public class TestLambda {

        public static void main(String[] args) {
                    //se crea un proveedor de tipo Persona, el
        cual obtiene una persona
            Supplier<Persona> supplier =
        TestLambda::llenarPersona;
                    //obtiene desde el proveedor la persona y la
        asigna a per
            Persona per = supplier.get();
                    // imprime el nombre
            System.out.println(per.getNombre());
        }
        // asigna los nombres y dirección a la persona
        public static Persona llenarPersona(){
            return new Persona("Pablo", "Andrade", "Loja");
        }
    }
```

Consumidor

Utilizan la interfaz Consumer<T>, tienen un sólo argumento de entrada y no devuelven ningún valor, en este ejemplo se usa la misma clase Persona que se utilizó en el ejemplo de tipo proveedor.

```
Consumer<Persona> persona = (p) -> System.out.println("Hola, " +
   p.getNombre());
persona.accept(new Persona("Jorge", "Angel","Ramon"));
```

REFERENCIA A MÉTODOS

Con esta funcionalidad no sólo se puede utilizar expresiones lambda sino que se puede hacer referencia a los métodos del objeto utilizando el operador ::, existen 3 tipos:

Métodos estáticos

```
import java.util.ArrayList;
import java.util.List;

public class TestReferenciaMetodos {

   public static void main(String[] args) {
      List names = new ArrayList();
          names.add("Andrea");
          names.add("Luisa");
          names.add("Diego");
          names.add("Paúl");
          names.add("Dario");
          names.forEach(System.out::println);
   }

}
```

En este ejemplo se utiliza el método System.out::println, como referencia a métodos estáticos.

Métodos de instancia

//recibe un objeto Usuario y devuelve la impresión de sus propiedades.

```
Function<Usuario, String> ftoString= Usuario::toString;
System.out.println(ftoString.apply(new
   Usuario("Santiago","Pardo",18,new Direccion("Nueva
   Dirección"))));
```

En este ejemplo se utiliza el método toString() que fue redefinido en la clase Usuario.

Referencia a mensajes

```
// referencia a mensajes
LinkedList<Integer> lista = new
   LinkedList<Integer>(Arrays.asList(1, 2, 3));
Supplier<Integer> funcion3 = lista::removeLast;
System.out.println(funcion3.get()); // 3
lista.forEach(System.out::println);
```

En este ejemplo se utiliza El método removeLast para eliminar el último elemento de la lista por último, se imprime la lista.