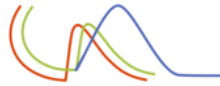
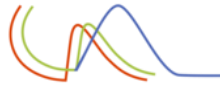


Contenido

LENGUAJE DE MODELADO UNIFICADO (UML).....	3
CARACTERÍSTICAS DE UML:	4
DIAGRAMAS DE ESTRUCTURA (VISTA ESTÁTICA)	4
DIAGRAMA DE CLASES	4
DIAGRAMA DE COMPONENTES	10
DIAGRAMA DE OBJETOS	11
DIAGRAMA DE ESTRUCTURA COMPUESTA.....	12
DIAGRAMA DE DESPLIEGUE	13
DIAGRAMA DE PAQUETES	15
JSE.....	16
Lenguaje java	18
1. Palabras reservadas del lenguaje.....	18
2. Tipos de datos.....	18
3. Identificadores	19
4. Variables	20
Operadores.....	21
1. Operadores.....	21
3. Precedencia operadores.....	22
Estructuras de programación	24
1. Comentarios	24
2. Sentencia if	24
3. Sentencia switch	24
4. Bucle for.....	26
5. Bucle for-each.....	26
6. Bucle while	26
7. Bucle do / while.....	27
8. Sentencia break y continue.....	27
9. Return.....	28
Arrays.....	29
Paquetes y módulos	31
1. Paquetes	31
Biblioteca de clases de Java.....	32
Paquetes más usados	32
2. Módulos	34
Clases	36
1. Definición de una clase.....	36
2. Variables	39



3. Referencia this	41
4. Constructores de una clase	41
5. Instanciación de objetos	42
6. Métodos	43
7. Método main	46
8. Clases internas	46
Herencia	48
1. Constructores y herencia	48
2. Redefinición de métodos	50
3. Clases abstractas	50
4. Métodos abstract	50
5. Clases y métodos final	51
INTERFACES	52
1. Definición de una interface	52
2. Herencia en interfaces	53
3. Upcasting y downcasting	54
Excepciones	56
1. Definición de una excepción	56
2. La pila de llamadas	56
3. Tipos de excepciones: verificadas y no verificadas	56
4. Lanzar y definir excepciones	57
6. Capturar y tratar una excepción	59
6. Crear excepciones personalizadas	60
10. ENTRADA / SALIDA	62
INTERFACES DE USUARIO	63
Novedades	66
1. Java 8	66
2. Java 9	66
3. Java 10	66
4. Java 11	67
5. Java 12	67
6. Java 13	68
7. Java 14	68



LENGUAJE DE MODELADO UNIFICADO (UML)

Un **modelo** es una simplificación de la realidad. El modelo nos proporciona los planos de un sistema, desde los más generales, que proporcionan una visión general del sistema, hasta los más detallados.

En un modelo se han de incluir los elementos que tengan más relevancia y omitir los que no son interesantes para el nivel de abstracción que se ha elegido.

UML (Unified Modeling Language) es un lenguaje de propósito general que ayuda a especificar, visualizar y documentar modelos de sistemas software, incluido su estructura y diseño, de tal forma que se unifiquen todos sus requerimientos.

Por un lado, el lenguaje de modelado puede servir de modelo para un proyecto y garantizar así una arquitectura de información estructurada; por el otro, ayuda a los desarrolladores a presentar la descripción del sistema de una manera que sea comprensible para quienes están fuera del campo. UML se utiliza principalmente en el desarrollo de software orientado a objetos. Al ampliar el estándar en la versión 2.0, también es adecuado para visualizar procesos empresariales.

Desde finales de la década de 1980 hasta la de 1990 se desarrollaron muchas técnicas de modelado OO. El resultado fue una confusa abundancia de métodos que apenas eran comparables entre sí. Destacamos los siguientes métodos:

- **Booch**, de Grady Booch.
- **OMT** (Object Modeling Technique), de James Rumbaugh.
- **OOSE** (Object-Oriented Software Engineering), de Ivar Jacobson.

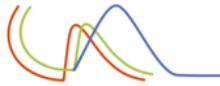
UML comenzó a gestarse a mediados de los noventa, cuando James Rumbaugh, Grady Booch e Ivar Jacobson se unieron en Rational. Esto derivó en el primer borrador de UML apareció en octubre de 1995.

Este lenguaje se abrió a la colaboración de otras empresas para que aportaran sus ideas. Todas estas colaboraciones condujeron a la definición de la primera versión de UML. En 1997 la versión UML 1.1 fue aprobada por la OMG convirtiéndose en la notación estándar de facto para el análisis y el diseño orientado a objetos.

OMG (Object Management Group - Grupo de Gestión de Objetos) es un consorcio dedicado al cuidado y el establecimiento de diversos estándares de tecnologías orientadas a objetos, tales como UML, XMI, CORBA. Es una organización sin ánimo de lucro que promueve el uso de tecnología orientada a objetos mediante guías y especificaciones para las mismas. El grupo está formado por compañías y organizaciones de software como HP, Oracle o Microsoft.

Tras esto, los desarrolladores crearon un grupo de trabajo para mejorar el lenguaje a través de múltiples versiones. Las críticas existentes incluían una semántica imprecisa e innecesariamente compleja, la falta de adaptabilidad y una estandarización inadecuada. Por lo tanto, se llevó a cabo una revisión a fondo. El resultado fue finalmente **UML 2.0**, que definió el **nuevo estándar** en 2005. La versión 2.4.1 constituye la base de la normalización:

- ISO/IEC **19505-1:2012 Infraestructura**.
- ISO/IEC **19505-2:2012 Superestructura**.



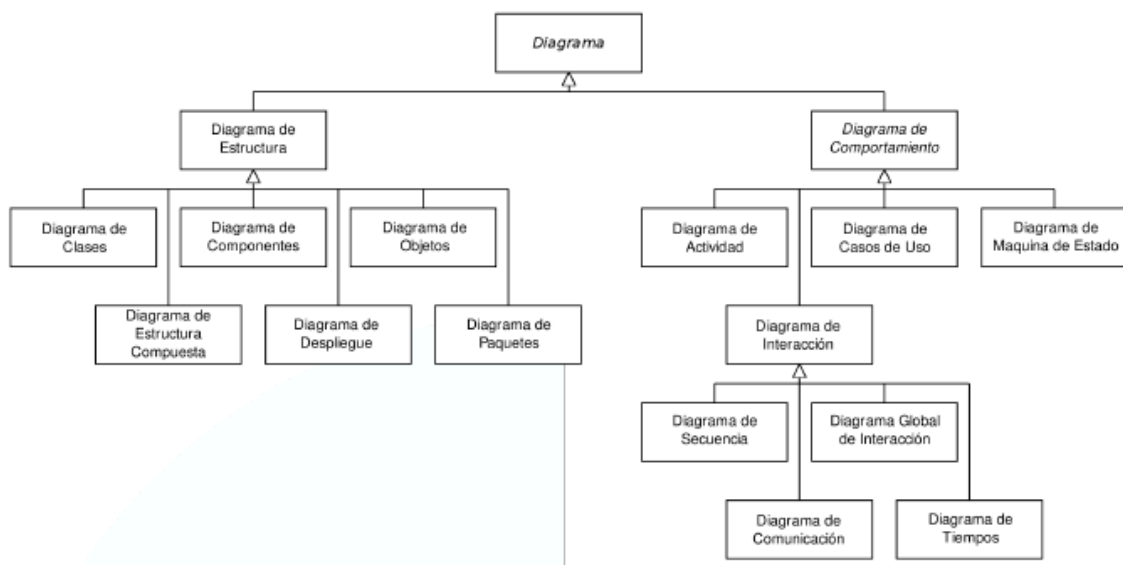
La versión actual de UML es la **2.5.1**, que apareció en diciembre de 2017.

CARACTERÍSTICAS DE UML:

- Es un **lenguaje estándar** que sirve para escribir los planos del software.
- Es un **lenguaje gráfico** para visualizar, especificar, construir y documentar un sistema.
- Es **extensible**. Es posible personalizar el lenguaje para un dominio concreto.
- Modela sistemas, desde los requisitos hasta los artefactos ejecutables, utilizando **técnicas de Orientación a Objetos**.
- Es **escalable**: puede manejar sistemas grandes y pequeños.
- Es **exhaustivo**: describe todos los aspectos importantes del sistema.
- **Construido sobre la experiencia**: es la culminación de las mejores prácticas en la comunidad de OO de los últimos años.
- Permite ver el sistema desde distintas perspectivas (**vistas**). UML abarca todas las fases del ciclo de vida de un proyecto, soporta diferentes maneras de visualización dependiendo de quién tenga que interpretar los planos y en qué fase del proyecto se encuentre.
- **No es una metodología**. Se puede usar en distintos tipos de metodología (cascada, iterativo...).

UML categoriza los diagramas en dos tipos:

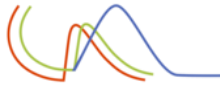
1. **ESTRUCTURALES**: muestran la estructura estática del sistema a modelar.
2. **De COMPORTAMIENTO**: muestran el comportamiento dinámico del sistema.



DIAGRAMAS DE ESTRUCTURA (VISTA ESTÁTICA)

DIAGRAMA DE CLASES

El objetivo principal de este modelo es la representación de los aspectos estáticos del sistema, utilizando diversos mecanismos de abstracción (clasificación, generalización, agregación).



El diagrama de clases recoge las clases de objetos y sus asociaciones. En este diagrama se representa la estructura y el comportamiento de cada uno de los objetos del sistema y sus relaciones con los demás objetos, pero no muestra información temporal.

Con el fin de facilitar la comprensión del diagrama, se pueden incluir paquetes como elementos del mismo, donde cada uno de ellos agrupa un conjunto de clases.

Este diagrama no refleja los comportamientos temporales de las clases, aunque para mostrarlos se puede utilizar un diagrama de máquina de estado.

Los **elementos** básicos del diagrama son:

- **Clases:** Una clase describe un conjunto de objetos con propiedades (atributos) similares y un comportamiento común. Los objetos son instancias de las clases.

No existe un procedimiento inmediato que permita localizar las clases del diagrama de clases. Éstas suelen corresponderse con sustantivos que hacen referencia al ámbito del sistema de información y que se encuentran en los documentos de las especificaciones de requisitos y los casos de uso.

Dentro de la estructura de una clase se definen los atributos y las operaciones o métodos:

1. Los atributos de una clase representan los datos asociados a los objetos instanciados por esa clase.
2. Las operaciones o métodos representan las funciones o procesos propios de los objetos de una clase, caracterizando a dichos objetos.

El diagrama de clases permite representar clases **abstractas**. Una clase abstracta es una clase que no puede existir en la realidad, pero que es útil conceptualmente para el diseño del modelo orientado a objetos. Las clases abstractas no son instanciables directamente sino en sus descendientes. Una clase abstracta suele ser situada en la jerarquía de clases en una posición que le permita ser un depósito de métodos y atributos para ser compartidos o heredados por las subclases de nivel inferior.

Las clases y en general todos los elementos de los diagramas, pueden estar clasificados de acuerdo a varios criterios, como por ejemplo su objetivo dentro de un programa. Esta clasificación adicional se expresa mediante un estereotipo. Algunos de los autores de métodos OO, establecen una clasificación de todos los objetos que pueden aparecer en un modelo. Los tipos son:

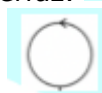
1. Objetos Entidad.



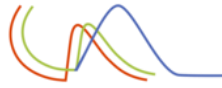
2. Objetos límite o interfaz.



3. Objetos de control.



Éstos son estereotipos de clases. Un estereotipo representa una la meta-clasificación de un elemento. Dependiendo de la herramienta utilizada, también se puede añadir información adicional a las clases para mostrar otras propiedades de



las mismas, como son las reglas de negocio, responsabilidades, manejo de eventos, excepciones, etc.

- **Relaciones:** Los tipos más importantes de relaciones estáticas entre clases son los siguientes:

o Asociación. Las relaciones de asociación representan un conjunto de enlaces entre objetos o instancias de clases. Es el tipo de relación más general, y denota básicamente una dependencia semántica. Por ejemplo, una Persona trabaja para una Empresa.

Cada asociación puede presentar elementos adicionales que doten de mayor detalle al tipo de relación:

§ Rol, o nombre de la asociación, que describe la semántica de la relación en el sentido indicado. Por ejemplo, la asociación entre Persona y Empresa recibe el nombre de trabaja para, como rol en ese sentido.

§ Multiplicidad, que describe la cardinalidad de la relación, es decir, especifica cuántas instancias de una clase están asociadas a una instancia de la otra clase.

Los tipos de multiplicidad son: Uno a uno, uno a muchos y muchos a muchos.

o Herencia. Las jerarquías de generalización/especialización se conocen como herencia. Herencia es el mecanismo que permite a una clase de objetos incorporar atributos y métodos de otra clase, añadiéndolos a los que ya posee. Con la herencia se refleja una relación “**es_un**” entre clases. La clase de la cual se hereda se denomina superclase, y la que hereda subclase.

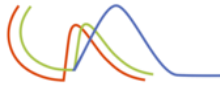
o La **generalización** define una superclase a partir de otras. Por ejemplo, de las clases profesor y estudiante se obtiene la superclase persona. La **especialización** especificación es la operación inversa, y en ella una clase se descompone en una o varias subclases. Por ejemplo, de la clase empleado se pueden obtener las subclases secretaria, técnico e ingeniero.

o Agregación. La agregación es un tipo de relación jerárquica entre un objeto que representa la totalidad de ese objeto y las partes que lo componen. Permite el agrupamiento físico de estructuras relacionadas lógicamente. Los objetos “**son-partede**” otro objeto completo. Por ejemplo, motor, ruedas, carrocería son parte de automóvil.

o Composición. La composición es una forma de agregación donde la relación de propiedad es más fuerte, e incluso coinciden los tiempos de vida del objeto completo y las partes que lo componen. Por ejemplo, en un sistema de Máquina de café, las relaciones entre la clase máquina y producto, o entre máquina y depósito de monedas, son de composición.

o Dependencia. Una relación de dependencia se utiliza entre dos clases o entre una clase y una interfaz, e indica que una clase requiere de otra para proporcionar alguno de sus servicios.

- **Interfaces:** Una interfaz es una especificación de la semántica de un conjunto de



operaciones de una clase o paquete que son visibles desde otras clases o paquetes.

Normalmente, se corresponde con una parte del comportamiento del elemento que la proporciona.

- **Paquetes:** Los paquetes se usan para dividir el modelo de clases del sistema de información, agrupando clases u otros paquetes según los criterios que sean oportunos.

Las dependencias entre ellos se definen a partir de las relaciones establecidas entre los distintos elementos que se agrupan en estos paquetes.

Notación

Clases: Una clase se representa como una caja, separada en tres zonas por líneas horizontales.

En la zona superior se muestra el nombre de la clase y propiedades generales como el estereotipo. El nombre de la clase aparece centrado y si la clase es abstracta se representa en cursiva. El estereotipo, si se muestra, se sitúa sobre el nombre y entre el símbolo: << >>.

La zona central contiene una lista de atributos, uno en cada línea. La notación utilizada para representarlos incluye, dependiendo del detalle, el nombre del atributo, su tipo y su valor por defecto, con el formato:

visibilidad nombre : tipo = valor-inicial { propiedades }

La visibilidad será en general publica (+), privada (-) o protegida (#), aunque puede haber otros tipos de visibilidad dependiendo del lenguaje de programación empleado.

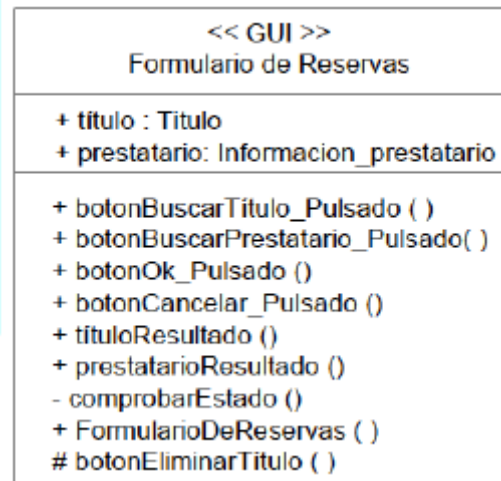
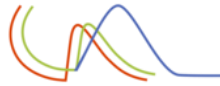
En la zona inferior se incluye una lista con las operaciones que proporciona la clase. Cada operación aparece en una línea con formato:

visibilidad nombre (lista-de-parámetros): tipo-devuelto { propiedad }

La visibilidad será en general publica (+), privada (-) o protegida (#), aunque como con los atributos, puede haber otros tipos de visibilidad dependiendo del lenguaje de programación.

La lista de parámetros es una lista con los parámetros recibidos en la operación separados por comas. El formato de un parámetro es:

nombre : tipo = valor-por-defecto



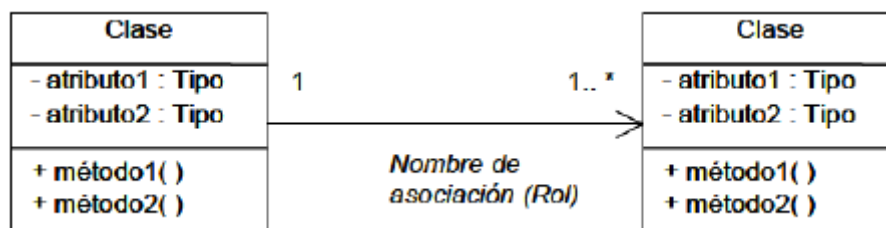
La notación especificada se puede simplificar según el nivel de detalle con el que se quiera trabajar en un momento dado.

Relaciones: Una relación de asociación se representa como una línea continua entre las clases asociadas. En una relación de asociación, ambos extremos de la línea pueden conectar con la misma clase, indicando que una instancia de una clase está asociada a otras instancias de la misma clase, lo que se conoce como asociación reflexiva.

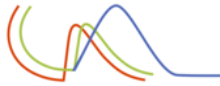
La relación puede tener un nombre y un estereotipo, que se colocan junto a la línea. El nombre suele corresponderse con expresiones verbales presentes en las especificaciones, y define la semántica de la asociación. Los estereotipos permiten clasificar las relaciones en familias y se escribirán entre el símbolo: << ... >>.

Las diferentes **propiedades de la relación** se pueden representar con la siguiente notación:

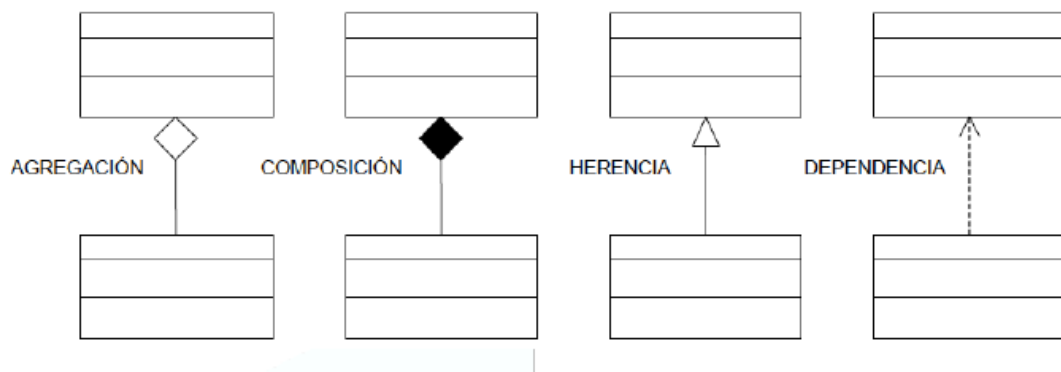
- Multiplicidad: La multiplicidad puede ser un número concreto, un rango o una colección de números. La letra 'n' y el símbolo '*' representan cualquier número.
- Orden: Se puede especificar si las instancias guardan un orden con la palabra clave '{ordered}'. Si el modelo es suficientemente detallado, se puede incluir una restricción que indique el criterio de ordenación.
- Navegabilidad: La navegación desde una clase a la otra se representa poniendo una flecha sin relleno en el extremo de la línea, indicando el sentido de la navegación.
- Rol o nombre de la asociación: Este nombre se coloca junto al extremo de la línea que esta unida a una clase, para expresar cómo esa clase hace uso de la otra clase con la que mantiene la asociación.



Además, existen notaciones específicas para los otros tipos de relación, como son:



- **Agregación:** Se representa con un rombo hueco en la clase cuya instancia es una agregación de las instancias de la otra.
- **Composición:** Se representa con un rombo lleno en la clase cuya instancia contiene las instancias de la otra clase.
- **Dependencia:** Una línea discontinua con una flecha apuntando a la clase cliente. La relación puede tener un estereotipo que se coloca junto a la línea, y entre el símbolo: << ... >>.
- **Herencia:** Esta relación se representa como una línea continua con una flecha hueca en el extremo que apunta a la superclase.



Interfaces: Una interfaz se representa como una caja con compartimentos, igual que las clases. En la zona superior se incluye el nombre y el estereotipo <<Interface>>. La lista de operaciones se coloca en la zona inferior, igual que en las representaciones de clases. La zona en la que se listan los atributos estará vacía o puede omitirse.

Existe una representación más simple para la interfaz: un círculo pequeño asociado a una clase con el nombre de la interfaz debajo. Las operaciones de la interfaz no aparecen en esta representación; si se quiere que aparezcan, debe usarse la primera notación.

Entre una clase que implementa las operaciones que una interfaz ofrece y esa interfaz se establece una relación de realización que, dependiendo de la notación elegida, se representará con una línea continua entre ellas cuando la interfaz se representa como un círculo y con una flecha hueca discontinua apuntando a la interfaz cuando se represente como una clase.

Paquetes: Los paquetes se representan mediante un icono con forma de carpeta y las dependencias con flechas discontinuas entre los paquetes dependientes.

Ejemplo: gestión de préstamos y reservas de libros y revistas de una biblioteca.

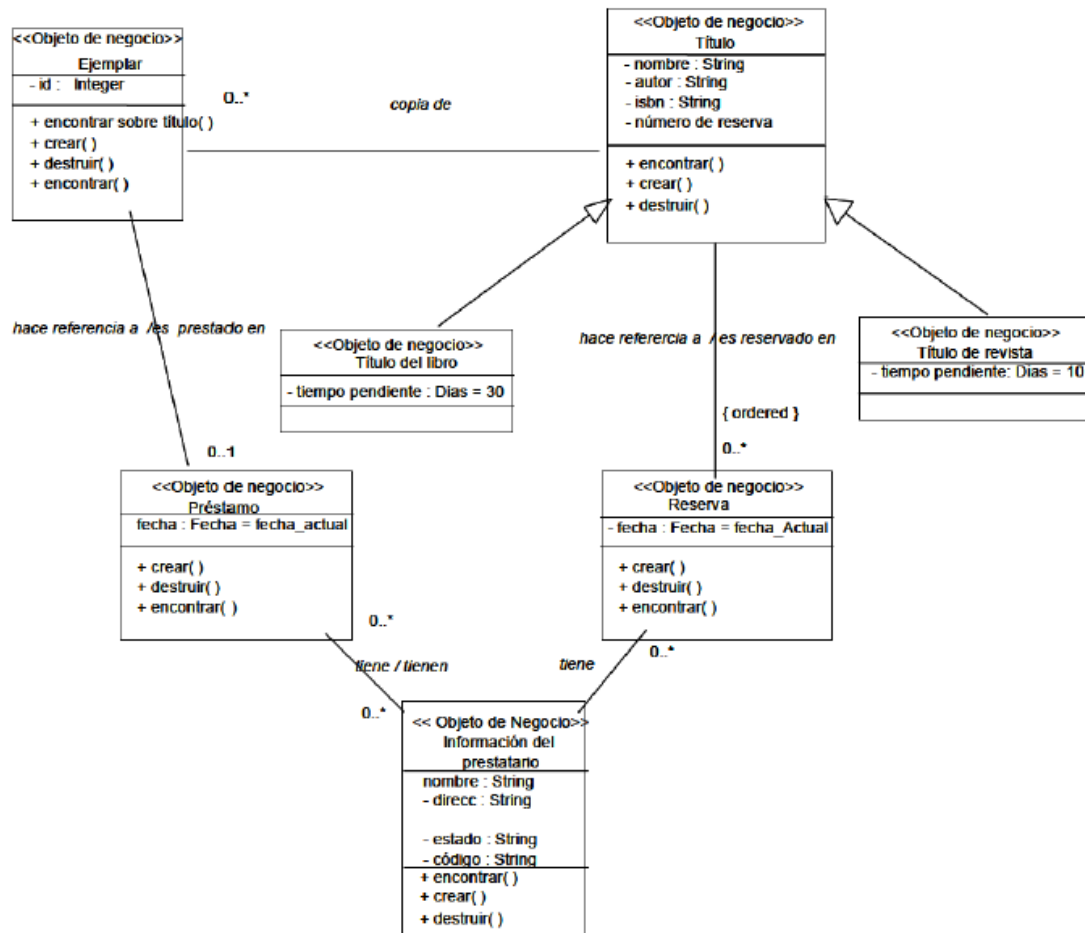
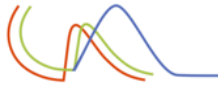


DIAGRAMA DE COMPONENTES

El diagrama de componentes proporciona una visión física de la construcción del sistema de información. Muestra la organización de los componentes software, sus interfaces y las dependencias entre ellos.

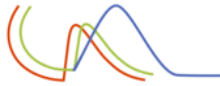
Como ya se ha indicado, los **elementos** de estos diagramas son los componentes software y las dependencias entre ellos.

Un componente es un módulo de software que puede ser código fuente, código binario, un ejecutable, o una librería con una interfaz definida. Una interfaz establece las operaciones externas de un componente, las cuales determinan una parte del comportamiento del mismo.

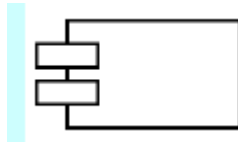
Además se representan las dependencias entre componentes o entre un componente y la interfaz de otro, es decir uno de ellos usa los servicios o facilidades del otro.

Estos diagramas pueden incluir paquetes que permiten organizar la construcción del sistema de información en subsistemas y que recogen aspectos prácticos relacionados con la secuencia de compilación entre componentes, la agrupación de elementos en librerías, etc.

Notación



Componente: Un componente se representa como un rectángulo, con dos pequeños rectángulos superpuestos perpendicularmente en el lado izquierdo.

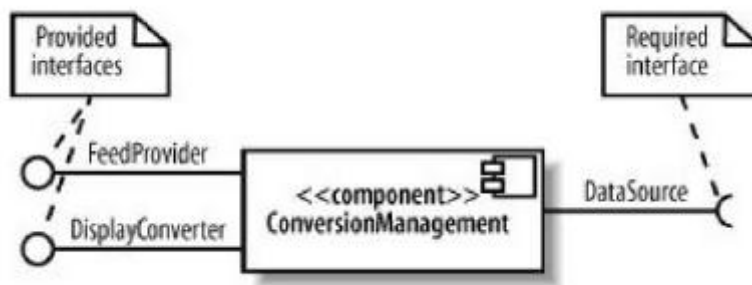


Para distinguir distintos tipos de componentes se les puede asignar un estereotipo, cuyo nombre estará dentro del símbolo: << ... >>

Interfaz: Se representa como un pequeño círculo situado junto al componente que lo implementa y unido a él por una línea continua. La interfaz puede tener un nombre que se escribe junto al círculo. Un componente puede proporcionar más de una interfaz. La relación entre un componente y su interfaz se llama **realización**.

Podemos distinguir dos tipos de interfaces:

- Interfaz de importación o requerida: un componente usa los servicios de otro.
- Interfaz de exportación o proporcionada: un componente realiza una interfaz (se ha definido junto al concepto de realización).



Paquete: Un paquete se representa con un icono de carpeta.

Relación de dependencia: si un componente requiere una interfaz, necesita que otro componente implemente o realice.

Ejemplo: gestión web tienda online.

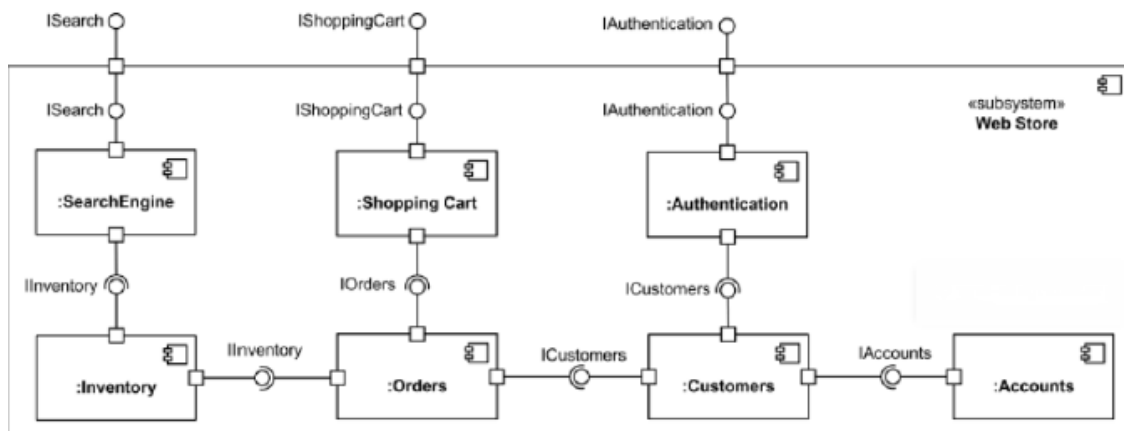
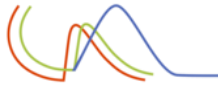


DIAGRAMA DE OBJETOS

Contiene los objetos con sus valores. Muestra una instantánea del sistema en un



determinado momento en el tiempo.

Notación

Se utiliza la misma notación que para el diagrama de clases. Se amplía con el concepto de objeto.

Instancename :
ClassName

Ejemplo:

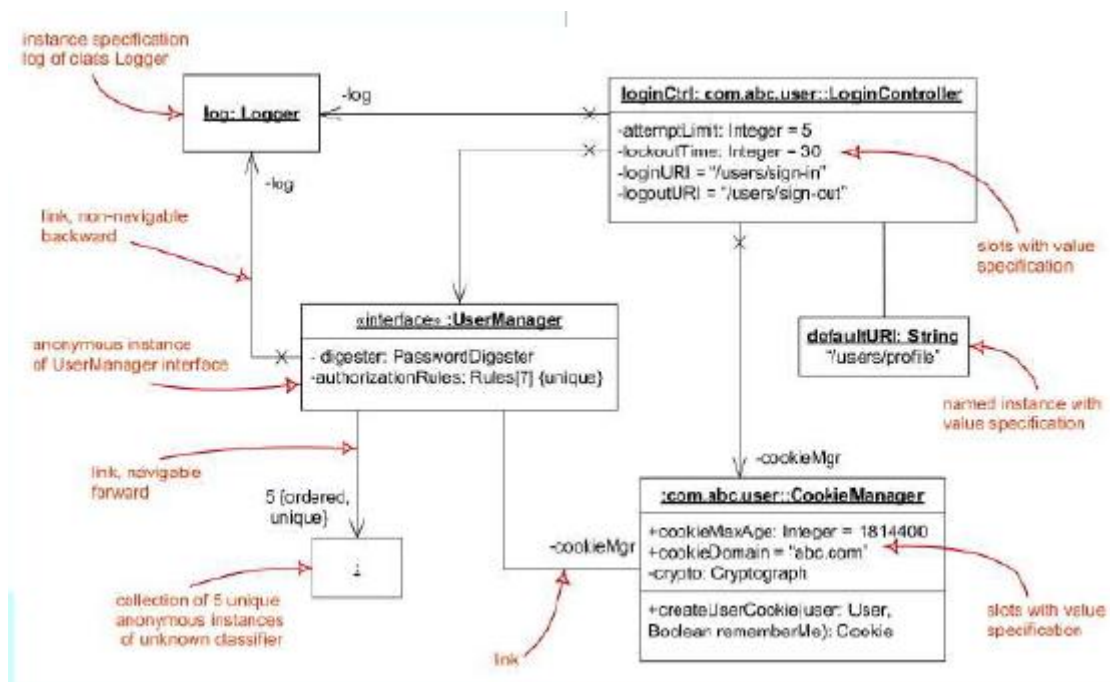


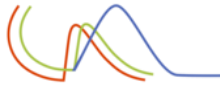
DIAGRAMA DE ESTRUCTURA COMPUESTA

Un diagrama de estructura compuesta muestra la estructura interna de una clase (visión de la estructura como una caja blanca), incluyendo sus puntos de interacción con otras partes del sistema (comunicación entre estructuras).

Este tipo de diagrama es un complemento al diagrama de clases. Es usado cuando es necesario representar la relación inusual entre elementos.

Los elementos de un diagrama de estructura compuesta son:

- **Parte o propiedad:** es un elemento de la estructura interna de una clase, representando una instancia o colección de instancias en tiempo de ejecución. La multiplicidad indica el número de instancias.
- **Conector:** comunica las partes de una estructura, conectando instancias concretas. La multiplicidad del conector indica el número de instancias que se conectan a cada lado.
- **Puerto:** indica un punto de interacción entre la clase y su entorno, especificando los servicios que provee y que requiere. Por eso, un puerto agrupa un conjunto de interfaces (requeridas y/o proporcionadas) relacionadas.

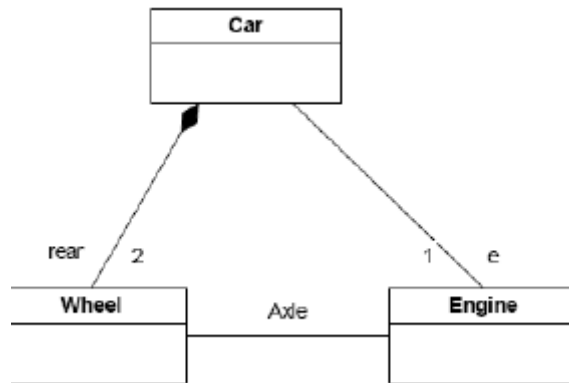


- **Colaboración:** define un conjunto de roles que cooperan entre sí para mostrar una funcionalidad concreta. Así, un rol es el papel que una instancia desempeña.

Notación

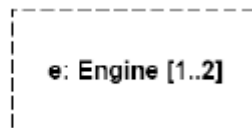
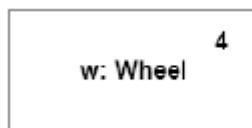
Clase: como ya se ha definido anteriormente.

Parte: es un objeto que se ubica dentro de una clase y que puede estar relacionado con otros objetos.



La multiplicidad de las partes se expresa mediante el número de instancias concretas o un rango.

Ejemplo: 4 ruedas y uno o dos motores.



Puerto: se muestra gráficamente como un rectángulo con nombre. Las interfaces proporcionadas se representan con un círculo y las requeridas con un semicírculo.

Ejemplo:

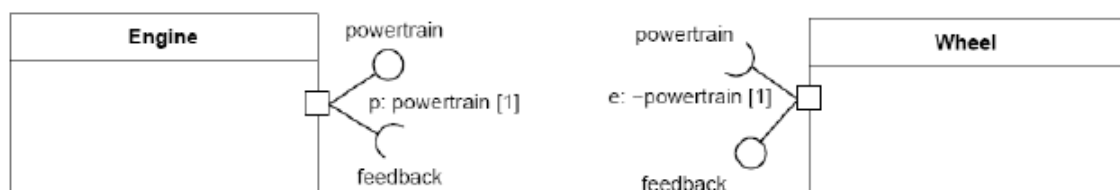
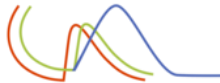


DIAGRAMA DE DESPLIEGUE

El objetivo de estos diagramas es mostrar la disposición de las particiones físicas del sistema de información y la asignación de los componentes software a estas particiones. Es decir, las relaciones físicas entre los componentes software y hardware en el sistema a entregar.

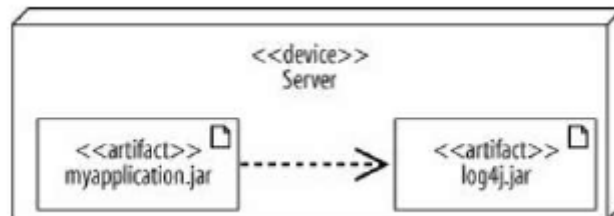
En estos diagramas se representan dos tipos de elementos, **nodos** y **conexiones**, así como la distribución de componentes del sistema de información con respecto a la partición física del sistema.

Se propone una definición concreta de nodo, prescindiendo de determinados detalles, pero permitiendo una continuidad tanto en el diseño como en la



construcción del sistema de información. Con este fin, se utiliza el nodo como partición física o funcional real, pero sin descender a detalles de infraestructura o dimensionamiento; por ejemplo, interesa si el nodo procesador es arquitectura Intel, pero no tanto si tiene dos o cuatro procesadores.

Dentro de un nodo se ubican los artefactos. Un **artefacto** es una pieza física de información que representa un software, como ejecutables, librerías, ficheros fuente, ficheros de configuración o bases de datos.



Las **conexiones** representan las formas de comunicación entre nodos.

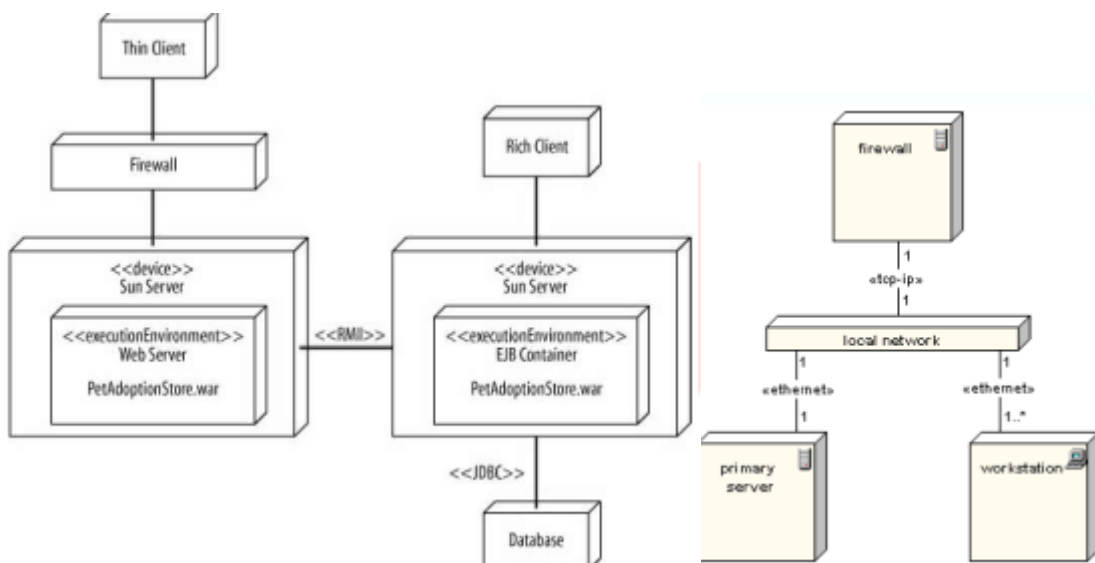
Además, a cada nodo se le asocia un subsistema de construcción que agrupa componentes software, permitiendo de este modo, determinar la distribución de estos componentes. Por lo tanto, un diagrama de despliegue puede incluir, dependiendo del nivel de detalle, todos los elementos descritos en la técnica de diagrama de componentes, además los nodos y las conexiones propios de esta técnica.

Notación

Nodo: Se representa con la figura de un cubo. El nodo se etiqueta con un nombre representativo de la partición física que simboliza. Se pueden asociar a los nodos subsistemas de construcción.

Conexión: Las conexiones se representan con una línea continua que une ambos nodos y pueden tener una etiqueta que indique el tipo de conexión. Ejemplo: canal, red, protocolo, etc. Se le puede asignar un estereotipo a la comunicación indicando el protocolo concreto.

Ejemplos:



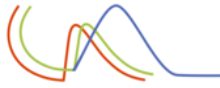


DIAGRAMA DE PAQUETES

El objetivo de estos diagramas es obtener una visión más clara del sistema de información orientado a objetos, organizándolo en subsistemas, agrupando los elementos del análisis, diseño o construcción y detallando las relaciones de dependencia entre ellos. El mecanismo de agrupación se denomina Paquete.

Estrictamente hablando, los paquetes y sus dependencias son elementos de los diagramas de casos de uso, de clases y de componentes, por lo que se podría decir que el diagrama de paquetes es una extensión de éstos.

Estos diagramas contienen dos tipos de elementos:

- **Paquetes:** Un paquete es una agrupación de elementos, bien sea casos de uso, clases o componentes. Los paquetes pueden contener a su vez otros paquetes anidados que en última instancia contendrán alguno de los elementos anteriores.
- **Dependencias entre paquetes:** Existe una dependencia cuando un elemento de un paquete requiere de otro que pertenece a un paquete distinto. Es importante resaltar que las dependencias no son transitivas.

Se pueden optimizar estos diagramas teniendo en cuenta cuestiones como: la generalización de paquetes, el evitar ciclos en la estructura del diagrama, la minimización de las dependencias entre paquetes, etc.

Resumiendo, el diagrama de paquetes es un mecanismo para gestionar la complejidad del modelado, permitiendo organizar los elementos del modelo y controlar el acceso a sus contenidos (visibilidad).

Notación

Paquete: Un paquete se representa mediante un símbolo con forma de 'carpeta' en el que se coloca el nombre en la pestaña y el contenido del paquete dentro de la 'carpeta'. En los casos en que no sea visible el contenido del paquete se podrá colocar en su lugar el nombre.

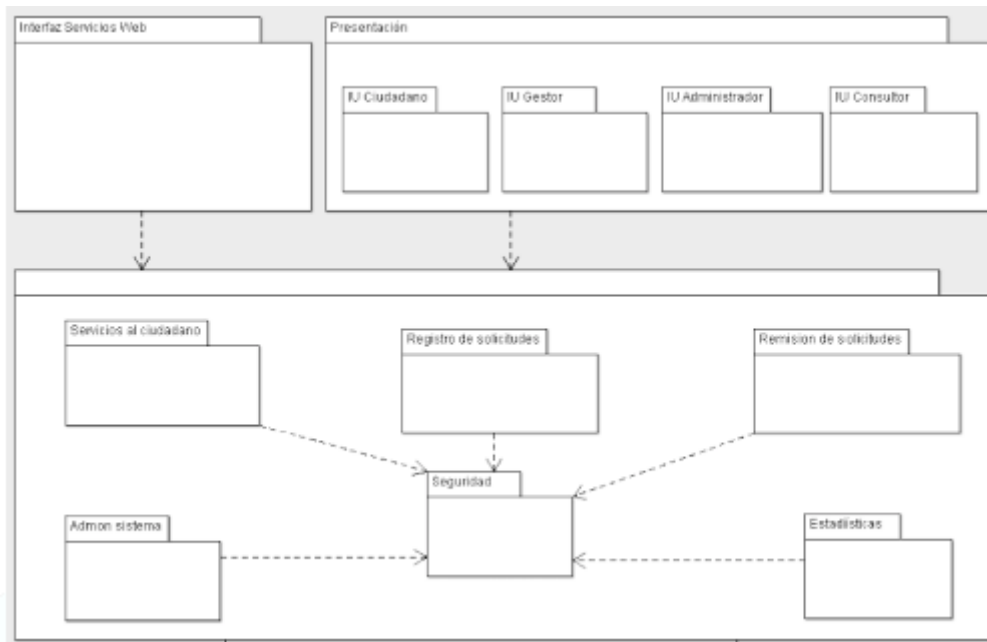
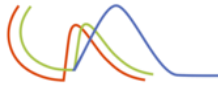
Si el paquete tiene definido un estereotipo, éste se representa encima del nombre entre el símbolo << ... >>, y si se definen propiedades, se representan debajo del nombre y entre llaves.

La visibilidad de los elementos que forman el paquete se debe indicar anteponiendo a su nombre los símbolos:

- '+' para los públicos.
- '-' para los privados.
- '#' para los protegidos.

Dependencia: Las dependencias se representan con una flecha discontinua con inicio en el paquete que depende del otro.

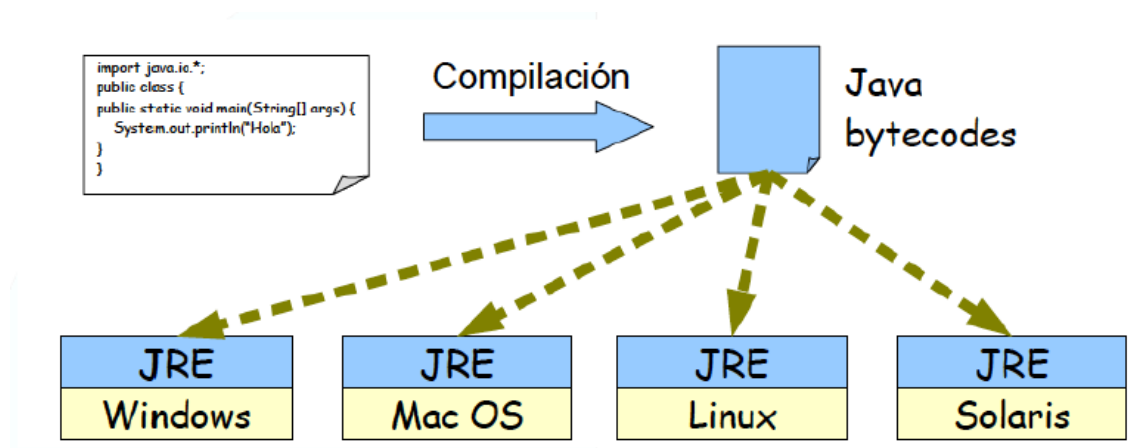
Ejemplo: Diagrama de paquetes de un sistema de Registro.



JSE

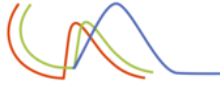
Java es un lenguaje de programación orientado a objetos desarrollado por Sun Microsystem. Una de sus principales características es la generación de código independiente de la plataforma, gracias al entorno de ejecución JRE (Java Runtime Environment).

Java es un lenguaje compilado, generando código binario Java (bytecodes) a partir del código fuente. Este código binario Java es independiente de la plataforma pero no directamente ejecutable. Para ello es necesario el uso de la JVM (Java Virtual Mach), la cual forma parte del JRE, que será la encargada de interpretar el código binario Java a la plataforma concreta.

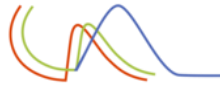


La JDK (Java Development Kit) incluye la JRE, el compilador de Java y las API de Java. Implementaciones de la JDK:

- **Java Platform, Standard Edition Development Kit:** es la implementación oficial de Oracle. Actualmente, en su versión 21 en 2024.
- **OpenJDK:** es la JDK de código abierto.



- **HotSpot VM:** desarrollada por Oracle (antes por Sun) ofrece implementaciones para clientes, servidores y sistemas embebidos (minimal VM).
- **GraalVM:** es una JDK diseñada para acelerar el rendimiento de las aplicaciones Java y consumir menos recursos. Además del lenguaje Java, proporciona runtime para JavaScript, Ruby, Python, R... Las capacidades políglotas de GraalVM permiten mezclar lenguajes de programación en una sola aplicación.



1. Palabras reservadas del lenguaje

PALABRAS RESERVADAS							
abstract	assert	boolean	break	byte	case	catch	char
class	const	continue	default	do	double	else	enum
extends	final	finally	float	for	goto	if	implements
import	instanceof	int	interface	long	native	new	null
package	permits	private	protected	public	return	sealed	short
static	super	strictfp	switch	synchronized	this	throw	throws
transient	try	void	volatile	while	yield		
NUEVAS PALABRAS RESERVADAS (soporte módulos)							
open	module	requires	transitive	exports	opens	to	uses
provides	with						

Las nuevas palabras reservadas únicamente se consideran como tales donde aparecen módulos (sensibles al contexto). En el resto, pueden utilizarse como identificadores.

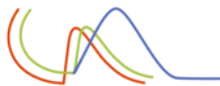
2. Tipos de datos

Tenemos dos categorías de tipos de datos:

1. **Primitivos:** representan un único valor del tamaño y formato adecuado.
2. **Referencias:** sus valores representan la dirección de un objeto, no el objeto en sí. Arrays, clases e interfaces son tipos referencia. Java no utiliza punteros y no permite modificar la dirección de memoria, sí el objeto.

Los tipos de datos primitivos son:

TIPO	DESCRIPCIÓN	TAMAÑO
byte	Entero de un byte	8 bits con signo en complemento a dos
short	Entero corto	16 bits con signo en complemento a dos
int	Entero	32 bits con signo en complemento a dos



long	Entero largo	64 bits con signo en complemento a dos
float	Nº en coma flotante de simple precisión	Nº de 32 bits IEEE 754
double	Nº en coma flotante de doble precisión	Nº de 64 bits IEEE 754
char	Un carácter Unicode	Carácter Unicode de 16 bits
boolean	Valor booleano (true, false)	8 bits

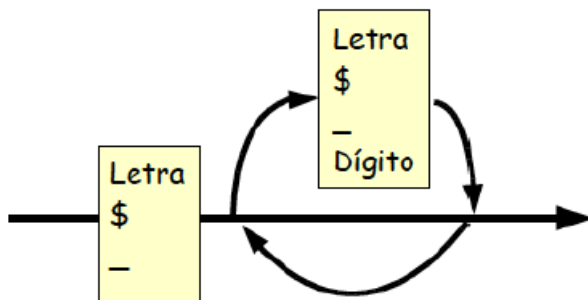
Caracteres especiales o de control para valores de tipo char:

Codigo	Significado
<code>\b</code>	Retroceso(backspace)
<code>\f</code>	Avance de página
<code>\n</code>	Nueva línea
<code>\r</code>	Retorno de carro
<code>\t</code>	Tabulador horizontal
<code>\"</code>	Comilla doble
<code>'</code>	Comilla simple
<code>\\</code>	Barra invertida
<code>\h</code>	Constante hexadecimal

3. Identificadores

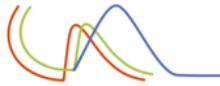
Utilizados para dar nombre a clases, métodos, campos, variables, constantes y etiquetas. Un identificador no puede coincidir con ninguna palabra reservada del lenguaje. Además, hay que tener en cuenta que Java es sensible a mayúsculas y minúsculas.

Un identificador sólo puede comenzar por una letra, \$ o _ y continuar con letra, \$ o dígitos numéricos:



Para el nombrado de identificadores se sigue el siguiente convenio:

- **Variables:** las palabras en minúscula. Si el nombre es compuesto, la inicial de cada palabra a partir de la segunda en mayúsculas (Ej: unaVariable). A esta forma de expresarlo se le llama nomenclatura camello.
- **Clases:** todo en minúscula salvo la primera letra de cada palabra (Ej: MiClase).



- **Constantes:** las palabras en mayúsculas separadas cada una por el carácter subrayado (UNA_CONSTANTE).

Un identificador no puede coincidir con una palabra clave.

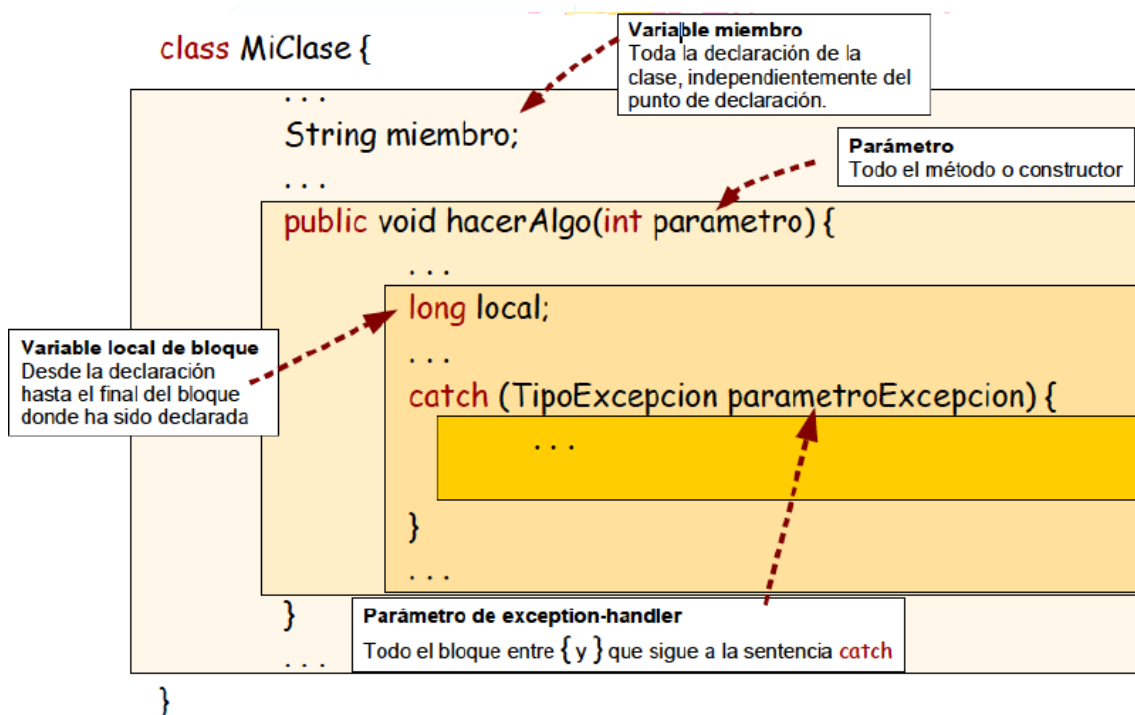
4. Variables

Toda variable tiene un nombre, un tipo y un alcance. También se le pueden asignar otras propiedades mediante modificadores y es posible asignarle un valor inicial. La sintaxis es:

[<modificadores>] <tipo_dato> <nombre_variable> [= <valor_inicial>];

Alcance, ambito o scope

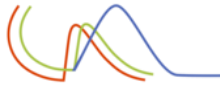
Es la región del programa dentro de la cual podemos referirnos a una variable por su nombre. También determina cuándo la variable es creada y destruida. Tenemos 4 categorías: variable miembro, parámetro de método, variable local y parámetro de excepción-handler



Las variables miembros y locales pueden ser inicializadas en la propia declaración.

Valores por defecto: boolean a false, char a '\0' y el resto a cero. Las variables de referencia a null.

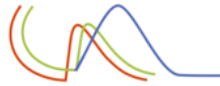
En el apartado de clases se verán las variables miembro estáticas y finales. (constantes).



Operadores

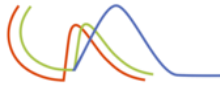
1. Operadores

TIPO	OPERADOR	SIGNIFICADO
Aritméticos	+	Adición (con String concatena las cadenas)
	-	sustracción
	*	multiplicación
	/	división
	%	resto de división entera
	Con los operadores anteriores se pueden combinar enteros, char y reales. char, byte y short se convierten a int antes de operar. Si los operandos de una expresión son de distinto tipo el inferior es convertido al tipo superior y el resultado es del tipo superior: int → long → float → double	
Lógicos y relacionales	>	mayor que
	>=	mayor o igual que
	<	menor que
	<=	
	==	igual
	!=	distinto
	&&	AND lógico
		OR lógico
	!	NOT lógico
Asignación	++	incremento unario
	--	decremento unario
	=	asignación simple
	*=	asignación de la multiplicación
	/=	asignación de la división
	%=	asignación del resto
	+=	asignación de la suma

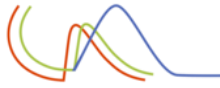


	-=	asignación de la resta
	<<=	asignación del desplazam. a la izquierda
	>>=	asignación del desplazam. a la derecha
	>>>=	asignación del desplazam. a la derecha con relleno de ceros
	&=	asignación de la operación AND
	=	asignación de la operación OR
	^=	asignación de la operación XOR
Tratamiento de bits	~	operación NOT
	&	operación AND
		operación OR
	^	operación XOR
	>>	desplazam. a la derecha con signo (relleno con ceros a la izquierda, salvo el primer bit que es el de signo)
	<<	desplazam. a la izquierda (relleno con ceros a la derecha)
	>>>	
Matrices	[]	acceso a los elementos de una matriz
	.	acceso a variables miembro
Expresiones	()	Modificación orden evaluación expresiones
Instancia de	instanceof	Comprobar si un objeto pertenece a una clase determinada
Condicional	?	Condición ? true : false
Casting	(tipo)	Convierte un tipo de mayor precisión en uno de menor
Creación	new	Crea una instancia de un objeto

3. Precedencia operadores



PRECEDENCIA OPERADORES	
Postfijos	[] . () expr++ expr--
Prefijos	++expr --expr ~ !
Creación o cast	new (tipo) expr
Aritméticos	* / %
	+ -
Desplazamiento	<< >> >>>
Relacionales	< <= >= > instanceof
Igualdad	== !=
AND bit a bit	&
XOR bit a bit	^
OR bit a bit	
AND lógico	&&
OR lógico	
Condicional	?:
Asignación	= += -= *= /= %= >>= <<= >>>= &= ^= =



Estructuras de programación

1. Comentarios

Java permite tres estilos de comentarios:

```
// comentario de línea
```

```
/* comentario de una o varias líneas */
```

```
/** igual que el anterior. Utilizado por javadoc para documentar app */
```

2. Sentencia if

FORMA 1

```
if(condición)
    sentencia;
```

FORMA 2

```
if(condición)
    sentencia1;
else
    sentencia2;
```

FORMA 3

```
if(condición)
    sentencia1;
else if(condición)
    sentencia2;
else
    sentencia3;
```

3. Sentencia switch

Expresión puede ser de tipo byte, short, char, int o enum. Si es una variable no puede

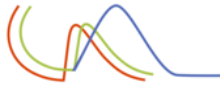
declararse en expresión. Se impone como restricción que no puede haber etiquetas case duplicadas. Es opcional el uso de default y no es obligatorio que se sitúe al final.

Ejemplo:

```
switch (numHijos) {
    case 0:      System.out.println("No tiene hijos"); break;
    case 1:
    case 2:      System.out.println("1 ó 2 hijos");
                break;
    default:     System.out.println("Familia numerosa"); }
```

Ejemplo:

```
switch (numHijos) {
    case 0:      System.out.println("No tiene hijos");
                break;
```

case 1:

case 2: System.out.println("1 ó 2 hijos");

break;

default: System.out.println("Familia numerosa");

Novedad en Java 12

No hace falta usar la sentencia break. Se pueden utilizar varios casos para cada rama.

Ejemplos:

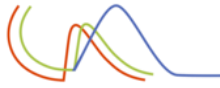
```
switch (day) {  
    case MONDAY, FRIDAY, SUNDAY -> System.out.println(6);  
    case TUESDAY -> System.out.println(7);  
    case THURSDAY, SATURDAY -> System.out.println(8);  
    case WEDNESDAY -> System.out.println(9);  
}  
  
final int integer = 2;  
String numericString;  
switch (integer)  
{  
    case 1 -> numericString = "one";  
    case 2 -> numericString = "two";  
    case 3 -> numericString = "three";  
    default -> numericString = "N/A";  
}
```

Novedades Java 13

En Java 13 se permite crear bloques de sentencias para cada rama case y retornar el valor con la palabra reservada yield. En los bloques de sentencias puede haber algún cálculo más complejo que directamente retornar el valor deseado.

Ejemplo:

```
String numericString = switch(integer) {  
    case 0 -> {  
        String value = calculateZero();    yield value;  
    } ;
```



```
case 1, 3, 5, 7, 9 -> {  
    String value = calculateOdd();    yield value;  
};  
case 2, 4, 6, 8, 10 -> {  
    String value = calculateEven();    yield value;  
}; default -> {  
    String value = calculateDefault();    yield value;  
}; };
```

4. Bucle for

Las tres partes son opcionales. La inicialización y el incremento pueden tener más de una sentencia separadas por comas. Las variables declaradas solo son visibles en el cuerpo del bucle.

```
for(inicialización; condición; incremento) {  
    sentencias;  
}
```

Ejemplo:

```
for (int n = 1, total = 0; n <= 10; total += n++);  
int i = 0, j = 0, k = 0;  
for (i++, j = i + 1, k = j + 2; j < 20; j++, k*=2)  
    System.out.println(i+j+k);  
for (;;;);
```

5. Bucle for-each

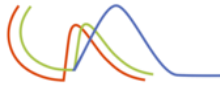
Permite iterar sobre una colección de elementos.

```
for(variable : coleccion) {  
    sentencia;  
}
```

Ejemplo:

```
int[] primos = {1,2,3,5,7,9,11,13};  
for (int p : primos)  
    System.out.println(p);
```

6. Bucle while



```
while(condición) {  
sentencia;  
}
```

7. Bucle do / while

```
do  
sentencia;  
while (condición);
```

8. Sentencia break y continue

Sentencias:

- **break**: permite salir de cualquier bloque. Se utiliza para interrumpir un bucle while, do, for o una sentencia switch.
- **continue**: salta al final del cuerpo de un bucle while, do o for, evaluándose la expresión boolean de control.

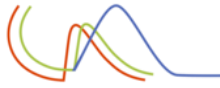
Ejemplo:

```
int n, total = 0;  
while ((n = siguiente()) > 0) {  
if (n % 2 == 1)  
continue;  
total += n++;  
}
```

```
int n = siguiente(), total = 0;  
for (;;) n = siguiente() {  
if (n == -1)  
break;  
if (n % 2 == 0)  
total += n++;  
}
```

Por defecto break y continue saltan el bucle más cercano, aunque podemos utilizar etiquetas para referirlo a cualquier bucle anidado. Para cambiar este comportamiento, podemos utilizar etiquetas.

Ejemplo:



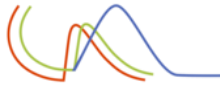
```
bucle1: while (...) {  
  for (...) {  
    // ...  
    continue bucle1;  
  }  
}
```

9. Return

Termina la ejecución de un método y puede devolver un valor de retorno. Es obligatorio si el método no es void.

Ejemplo:

```
void f() {  
  // ...  
  return;  
}  
  
double g(double x, double y) {  
  // ...  
  return x * y / (x + y);  
}
```



Arrays

Estructura de datos que almacena múltiples valores del mismo tipo. El tamaño del array se determina en la creación del array no pudiendo modificarse después.

Sintaxis de la declaración de un array:

```
<tipo>[] <nombre_array>;
```

Ejemplo: `String[] titulos; Double[] precios;`

Para la creación de un array utilizamos el operador `new`. Los elementos son inicializados automáticamente a sus valores por defecto.

Ejemplo: `titulos= new String[10];`

`Double[] precios= new double[20];`

Es posible la creación e inicialización en un único paso. El tamaño del array viene determinado por el número de elementos especificados entre las llaves.

Ejemplo:

```
String [] titulos= {"Nombre", "Apellidos","NIF"};
```

```
Double[] precios= {2.4, 3.25, 4.99};
```

```
Rectangle [] rects = {new Rectangle(), new Rectangle(4,2)};
```

Todo array tiene una variable implícita denominada `length` que permite conocer su tamaño.

Ejemplo:

```
int[] primos = new int[10];
```

```
System.out.println("primos tiene " + primos.length + " elementos");
```

Para acceder a los elementos de un array, tanto para consultarlos como para modificarlos, utilizamos el operador `[]` y un índice que varía de 0 a `length-1`.

Ejemplo:

```
for (int i = 0; i < primos.length; i++) {
```

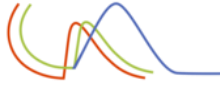
```
System.out.println(primos[i]);
```

```
}
```

El acceso a un elemento es comprobado en tiempo de ejecución. El acceso fuera de rango provoca el lanzamiento de la excepción `java.lang.ArrayIndexOutOfBoundsException`.

Ejemplo:

```
Point[] puntos = { new Point(0,0), new Point(1,3) };
```



```
puntos[2].translate(-1, 3);
```

Por último, el lenguaje nos permite definir arrays de arrays.

Ejemplo:

```
double[][] matriz1 = new double[3][2];
```

```
double[][] matriz2 = {{2, 1}, {0, -1}, {3, -2}};
```

```
String[][] autores = {
```

```
{ "Elton John", "Sting", "Bruce Springsteen"},
```

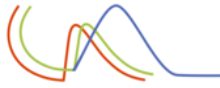
```
{ "Shakira", "Alejandro Sanz"},
```

```
{ "Estopa", "El último de la fila", "Hombre G" } };
```

```
System.out.println("Mi preferido es " + autores[0][2]);
```

```
Point[][] puntos = new Point[3][];
```

```
puntos[0] = new Point[5];
```



1. Paquetes

Las clases en Java se agrupan en paquetes. Un paquete es una estructura lógica, análoga a un directorio. Un paquete podrá contener subpaquetes y clases. Un paquete tiene un nombre del tipo:

identificador[.identificador ...]

La estructura del nombre de un paquete se corresponde con la estructura de directorios donde se encuentran sus clases a partir de un cierto directorio del classpath.

El nombre totalmente calificado de una clase es:

nombre_paquete.nombre_clase

Para referirnos a una clase sin ambigüedades y que Java pueda encontrar la clase, usamos el nombre totalmente calificado de ésta.

Los paquetes tienen dos funciones:

- Hacer uso en nuestras clases de clases contenidas en otros paquetes (import).
- Definir paquetes para nuestras clases (package).

Para evitar usar nombres tan largos, utilizamos la sentencia import, que anuncia que vamos a utilizar una clase concreta o varias de un paquete.

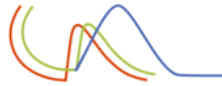
```
import paquete.clase; // Esa clase en concreto
```

```
import paquete.*; // Varias clases del paquete
```

Es posible importar una clase concreta dentro de un paquete o todas las clases del paquete (usamos *).

Ejemplo:

```
import java.util.*; // importa todas las clases de java.util
import java.net.Socket; // solo importa la clase Socket de java.net
```



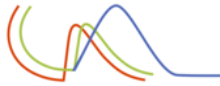
Las clases del paquete `java.lang` se importan automáticamente y no es necesaria sentencia `import` cuando desarrollamos cualquier clase.

Biblioteca de clases de Java

Java define una gran cantidad de clases estándar que están disponibles para todos los programas, solo es necesario usar la sentencia `import`. Esta biblioteca de clase a menudo se denomina API de Java, la cual se almacena en paquetes. En la parte superior de la jerarquía del paquete está `java`. Descendiendo de Java hay varios subpaquetes.

Paquetes más usados

PAQUETES MÁS USADOS	
java.io	Entrada/Salida
java.lang	Clases fundamentales: <code>Object</code> , tipos de datos (<code>Integer</code> , <code>Double</code> , <code>String...</code>), <code>threads</code> . Destacan: <ul style="list-style-type: none">- String: cadena de caracteres- StringBuffer: secuencia de caracteres mutable y segura para subprocesos- StringBuilder: secuencia de caracteres mutable. Se recomienda su uso ya que es más eficiente- Object: clase raíz de la jerarquía. Métodos: <ul style="list-style-type: none">◦ <code>clone</code>: crea una copia de un objeto◦ <code>equals</code>: indica si un objeto es igual a otro◦ <code>hashCode</code>: devuelve un código hash para el objeto◦ <code>toString</code>: cadena de caracteres que representa al objeto
java.math	Clases para tratamientos aritméticos
java.naming	API de JNDI
java.net	Aplicaciones de red
java.rmi	
java.sql javax.sql	Manejo de fuentes de datos



java.util	Clases de utilidad: <ul style="list-style-type: none">- Collection (interfaz): grupo de objetos- Map (interfaz): conjunto de pares clave-valor, no admite claves duplicadas- HashMap: implementa Map en tabla hash- Set (interfaz): colección que no contiene elementos duplicados- HashSet: implementación Set en tabla hash- List (interfaz): colección ordenada- ArrayList: array que implementa List- Vector: array- Stack: pila, hereda de vector- Queue (interfaz): cola- Deque (interfaz): cola que permite la inserción/borrado de elementos por los dos extremos
	- Iterator (interfaz) : itera sobre una colección.
java.crypto	Operaciones criptográficas
javax.swing	Clases para interfaz gráfica de usuario
javax.xml	Tratamiento de documentos XML

API de Java SE 21: <https://docs.oracle.com/en/java/javase/21/docs/api/index.html>

Por otro lado, cuando definimos una clase, ésta forma parte de un paquete que definimos nosotros.

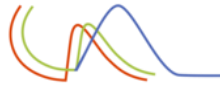
Ejemplo:

```
package dominio;
```

```
public class Solicitante() {
```

```
...
```

```
}
```



2. Módulos

Hasta ahora, desplegamos una aplicación empaquetando en un JAR todos los paquetes que vamos a utilizar, sin dependencias entre ellos y siendo todas las clases visibles. Java 9, para solucionar este problema, incorpora el concepto de módulo.

Un **módulo** es un conjunto de clases que pueden contener uno o varios packages y que define las dependencias con el resto de módulos, así como la visibilidad de las clases que contiene.

Los módulos van a mejorar una de las deficiencias existentes en la visibilidad de las clases entre paquetes. Los módulos de Java proporcionan una mayor encapsulación de las clases contenidas en un paquete y las librerías. Esta encapsulación evita que una aplicación u otra librería haga uso y dependa de clases y paquetes de los que no debería, lo que mejora la compatibilidad con versiones futuras.

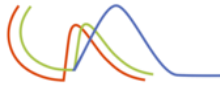
Los desarrolladores de una librería con los módulos ahora tienen un mayor control de los paquetes que expone una librería y que forma parte de su API pública.

Los módulos proporcionan:

- **Encapsulación fuerte:** se diferencia entre que es la API pública y usable y la parte privada a la que impide su uso accidental y acoplamiento indeseado entre módulos. La parte privada está encapsulada y de esta forma puede modificarse libremente con la seguridad de no afectar a los usuarios del módulo.
- **Interfaces bien definidas:** el código no encapsulado forma parte de la API del módulo, dado que otros módulos pueden usar esta API pública hay que tener especial cuidado al modificarlo por si se introducen cambios que sean incompatibles. Los módulos deben exportar una API bien definida y estable.
- **Dependencias explícitas:** los módulos necesitan a menudo otros módulos, estas dependencias son parte de la definición del módulo. Las dependencias explícitas forman un grafo que es importante conocer para entender las necesidades de una aplicación y para ejecutarla con todas sus dependencias.

Las ventajas del uso de módulos son:

- **Configuración confiable:** el sistema de módulos comprueba si una combinación de módulos satisface todas las dependencias antes de compilar o ejecutar una aplicación.
- **Encapsulación fuerte:** se evitan dependencias sobre detalles internos de implementación.
- **Desarrollo escalable:** se crean límites entre el equipo que desarrolla un módulo y el que lo usa.



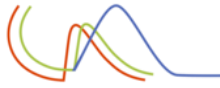
- Optimización: dado que el sistema de módulos sabe que módulos necesita cada uno solo se consideran los necesarios mejorándose tiempos de inicio y memoria consumida.
- **Seguridad:** la encapsulación y optimización limita la superficie de ataque.

La definición de un módulo se realiza con un nuevo archivo de código fuente de nombre `module-info.java`. Con la palabra reservada `requires` y una línea por paquete se definen qué paquetes requiere el módulo. Con la palabra reservada `exports` se define que paquetes del módulo se exportan y son visibles por algún otro módulo que lo requiera.

También se han añadido las palabras reservadas `provides` y `uses` para proporcionar y usar definiciones de servicios.

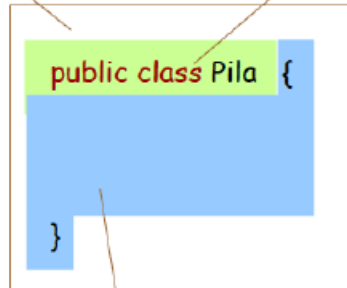
Ejemplo:

```
module modulo1 {  
    requires java.base;  
    exports app.utils;  
}
```



1. Definición de una clase

Definición de la clase



Declaración de la clase

Aquí decimos las propiedades de la clase

Cuerpo.

Aquí van los miembros:

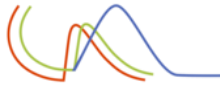
- Variables de clase y de instancia
- Constructores
- Métodos de clase y de instancia
- Otras cosas

La sintaxis para definir una clase es:

```
[acceso] class MiClase [extends Clase] [implements interfaces] {  
[acceso] [static] [final] tipo atributo1;  
[acceso] [static] [final] tipo atributo2;  
[acceso] [static] [final] tipo atributo3;  
...  
[access] [static] tipo metodo1(listaDeArgumentos) {  
...código del método...  
}  
...  
}
```

Ejemplo: Definición de la clase User.

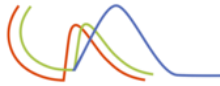
```
package dominio;  
import java.util.Date;  
import java.util.HashSet;  
import java.util.Iterator;  
import java.util.Set;
```



```
public class User {  
    private String username;  
    private String password;  
    private String nombre;  
    private String apellidos;  
    private String DNI;  
    private boolean admin;  
    private Date fechaAlta;  
    public User()  
    public User(String username) {this.username = username;}  
    public User(String username, String password) {  
        this.username = username;  
        this.password = password;  
    }  
  
    public String getUsername() {return username;}  
    public void setUsername(String username) {this.username = username;}  
    public String getPassword() {return password;}  
    public void setPassword(String password) {this.password = password;}  
    public String getNombre() {return nombre;}  
    public void setNombre(String nombre) {this.nombre = nombre;}  
    public String getApellidos() {return apellidos;}  
    public void setApellidos(String apellidos) {this.apellidos = apellidos;}  
    public String getDNI() {return DNI;}  
    public void setDNI(String DNI) {this.DNI = DNI;}  
    public boolean isAdmin() {return admin;}  
    public void setAdmin(boolean admin) {this.admin = admin;}  
    public Date getFechaAlta() {return fechaAlta;}  
    public void setFechaAlta(Date fechaAlta) {this.fechaAlta = fechaAlta;}  
    @Override  
    public String toString() {return nombre + " " + apellidos;}  
}
```

La forma completa (sin herencia ni interfaces) de declarar una clase es:

[**public**][**abstract**][**final**] class Nombre



[public][abstract][final] class Nombre

public	La clase es visible en otros paquetes. Por defecto no lo es.
abstract	La clase no puede tener instancias. Por defecto es concreta, es decir, sí puede tener instancias.
final	Otras clases no pueden heredar de ésta. Por defecto sí pueden.

Consideraciones:

- El orden de los modificadores es irrelevante.
- Una clase no puede ser abstract y final a la vez.

Ejercicio: Indique para cada declaración si es correcta o no.

public class A{...}

class B{...}

final class C{...}

final public D{...}

public abstract class E{...}

final abstract class F{...}

abstract class G{...}

abstract class public H{...}

Modificadores de acceso (visibilidad) de clases e interfaces

	Fuera del paquete	Paquete
default (sin modificador)	NO	SI
public	SI	SI

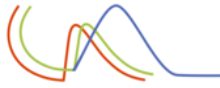
Ejemplos:

public class MiClase {...} // Clase pública

class MiClase2 {...} // Clase con modificador default. Solo visible dentro

// del paquete donde se define

Clases selladas (sealed)



La clase sellada (sealed class) es una característica introducida en Java 15 y mejorada en Java 16 y 17. Una clase sellada es una clase que tiene restricciones en la jerarquía de clases y controla qué clases pueden extender o implementar una clase sellada.

Para definir una clase sellada, se utiliza la palabra clave sealed antes de la definición de la clase, seguida de la palabra clave permits y la lista de clases permitidas para extender o implementar la clase sellada.

```
public sealed class Shape permits Circle, Square, Triangle {    // ... }
```

En este ejemplo, la clase Shape es sellada y permite que las clases Circle, Square y Triangle la extiendan o la implementen. Cualquier otra clase que intente extender o implementar la clase Shape dará como resultado un error de compilación.

Ejemplo:

```
public sealed class Rectangle extends Shape permits FilledRectangle {    public double length, width;
}
```

Con **non-sealed** se especifica una clase no sellada, esto es, que puede ser ampliada por subclases desconocidas.

```
public non-sealed class Square extends Shape {    public double side;
}
```

Como consecuencia, una clase sellada no puede impedir que sus subclases permitidas hagan esto.

2. Variables

Variables finales

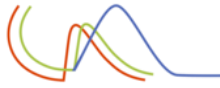
Podemos declarar una variable final, lo que significa que no puede modificarse una vez inicializada. Su uso es para definir constantes.

Ejemplo: final double PI = 3.141592654;

Variables miembro

Distinguimos dos tipos de variables miembro: de instancia y de clase.

Las **variables miembro de instancia o de objeto** son los atributos de nuestra clase en los que cada objeto mantiene una copia de cada una de ellas. Es decir,



todos los objetos de la misma clase tienen los mismos atributos y cada uno con sus valores concretos.

```
public class MiClase {  
    <tipo> <nombre_atributo>;  
}
```

Ejemplo:

```
public class User {  
    static int contador; // variable estática  
    static final int maxUsuarios; // variable estática final  
    private String username; // variable miembro de objeto  
    private String password; // variable miembro de objeto  
    ...  
}
```

La forma completa de declaración es:

acceso [static][final] [transient] [volatile] tipo nombre;

static **Se trata de un campo de clase en lugar de instancia. Por defecto es de instancia.**

final	Su valor no puede ser modificado una vez asignado, es decir, una constante. Por defecto sí puede ser modificado.
--------------	--

transient	Esta variable no debería ser serializada en el proceso de serialización de objetos. Por defecto sí lo es.
------------------	---

volatile	Poco utilizada. Indique que el valor de la variable puede ser modificado por varios hilos.
-----------------	--

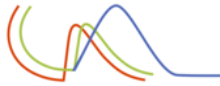
Consideraciones:

- El orden de los modificadores, incluyendo el de accesibilidad, es irrelevante.

Ejercicio: Indique para cada declaración si es correcta o no.

```
private final static double PI = 3.14159254;  
transient double x;  
protected final String[] cabeceras = {"x", "f(x)"};  
protected Rectangle rect;  
final double public AAA = 6.023E23;
```

Los campos pueden:



- **No inicializarse**, con lo que tendrán los valores por defecto según su tipo.
- **Inicializarse en la declaración**, asignándoles el valor de una expresión. En esa expresión no se pueden invocar métodos que lancen excepciones verificadas.

Modificadores de acceso (visibilidad) para variables

Modo de Acceso	Desde la misma clase	Desde el mismo paquete	Subclase	Resto de paquetes
public	SI	SI	SI	SI
protected	SI	SI	SI	NO
default	SI	SI	NO	NO
private	SI	NO	NO	NO

3. Referencia this

Cuando aparece una variable o un método sin una referencia explícita, se entiende que se refieren implícitamente al objeto donde aparece.

Ejemplo: `x = 3;` // `x` es una variable de instancia o clase

También podemos usar una referencia explícita con `this`.

Ejemplo: `this.x = 3;` // `x` es una variable de instancia o clase

Un objeto puede hacer referencia a sí mismo mediante la referencia especial `this`. La usaremos cuando queramos referirnos a una variable de instancia con el mismo nombre que un argumento del método. También en un método cuando pasemos como argumento una referencia a sí mismo.

Ejemplo:

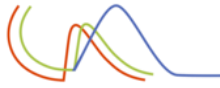
```
class Punto {  
    private int x, y;  
    public Punto(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
    // ...  
}
```

4. Constructores de una clase

Todas las clases tienen al menos un constructor pero pueden tener todos los que queramos.

Los constructores no tienen tipo de retorno y solo admiten modificadores de acceso.

[acceso] NombreClase([argumentos])



Si en una clase no declaramos ningún constructor, Java añade uno por defecto public y sin argumentos.

```
public class MiClase {  
}
```

```
public class MiClase {  
    public MiClase() {}  
}
```

Ejemplos:

```
public class Rectangle ... {  
    ...  
    public Rectangle(int x, int y, int width, int  
height) {...}  
    public Rectangle(int x, int y) {...}  
    public Rectangle(Point p) {...}  
    public Rectangle() {...}  
    ...  
}
```

```
public class Pila {  
    ...  
    public Pila() {...}  
    public Pila(int tamanyo)  
    {...}  
    ...  
}
```

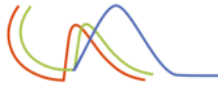
5. Instanciación de objetos

Cuando declaramos un objeto (por ejemplo, MiClase objeto) lo que tenemos es una referencia que no apunta a ningún objeto ya que su valor por defecto es null.

Con el operador new Java crea un objeto del tipo definido, paso necesario antes de poder usar dicho objeto. La sintaxis para instanciar un objeto a partir de su constructor por defecto es:

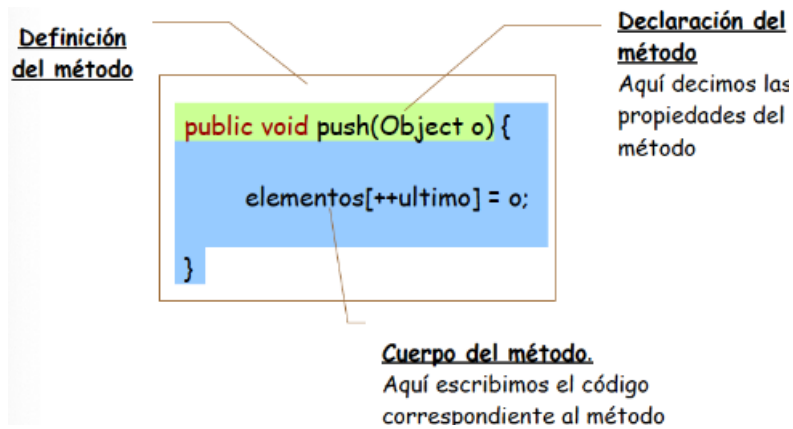
```
MiClase objeto = new MiClase();
```

Como diferencia con C++, en Java no se declara un destructor de clase. Java dispone de un recolector de basura (**garbage collector**) que se encarga de liberar de forma automática la memoria asignada a objetos cuando ya no es necesario su uso.



6. Métodos

Todos los métodos se definen dentro de una clase.



De entre los métodos definidos dentro de una clase, distinguimos dos tipos de métodos: de instancia y de clase.

Métodos de instancia

Son los métodos que se aplican a un objeto. Utilizamos para ello el operador punto. La sintaxis

para declarar un método de instancia es:

```
public class MiClase {  
<retorno> nombre_metodo([argumentos]) {...}  
}
```

La sintaxis para invocar a un método de instancia es:

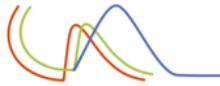
`objeto.metodo()`

Ejemplos:

```
public String getNombre() {  
    return nombre;  
}  
  
public void setNombre(String nombre) {  
    this.nombre = nombre;  
}
```

Métodos de clase o estáticos

Estos métodos se aplican a nivel de clase. No se necesita instanciar un objeto para hacer uso de ellos. Por este motivo en el cuerpo del método no se utiliza la referencia `this` ya que no hay objeto. La sintaxis para declarar un método estático es:



```
public class MiClase {  
    static <retorno> nombre_metodo([argumentos]) {...}  
}
```

La sintaxis para invocar a un método estático es:

Clase.metodo()

Ejemplos:

```
public static double suma(double a, double b) {  
    return a+b;  
}
```

Argumentos de un método

Los argumentos en Java se pasan por valor, es decir, se pasa una copia del valor del argumento.

El nombre de los argumentos debe ser único en su ámbito. No puede coincidir con el de:

- Otro argumento del método.
- Una variable local del método.
- Un parámetro de una sentencia catch.

Ejemplo:

```
void incorrecto(int a, boolean b, String a, double c) {  
    int b;  
    try {  
        ...  
    } catch (Exception c) {...}  
}
```

Sobrecarga de métodos

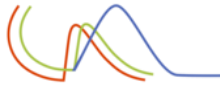
Podemos tener métodos de la misma clase con el mismo nombre, siempre que tipo y/o número de argumentos varíen. El método utilizado se determina en tiempo de ejecución por el tipo y/o número de los argumentos.

Ejemplos:

clase `java.io.PrintWriter`

```
public void print(boolean b);  
public void print(char[] s);  
public void print(float f);  
public void print(long l);  
public void print(String s);
```

```
public void print(char c);  
public void print(double d);  
public void print(int i);  
public void print(Object o);
```



La forma completa de declarar un método (sin excepciones) es:

[static][final][abstract][native][synchronized] tipo_retorno nombre(args) {...}

static	Método de clase en lugar de instancia. Por defecto de instancia.
final	No puede ser redefinido por las clases que extiendan ésta. Por defecto sí puede.
abstract	Se trata de un método abstracto (sin cuerpo). Por defecto es un método concreto y proporciona una implementación.
native	Es un método implementado en otro lenguaje (C, C++).
synchronized	Asegura que un único thread ejecuta el método a la vez.

Consideraciones:

- Orden de los modificadores irrelevante.
- Un método no puede ser `abstract` y `final` a la vez.

Ejercicio: Indique para cada declaración si es correcta o no.

`public synchronized void push(Object o) {...}`

`public synchronized Object pop() {...}`

`public synchronized boolean esVacio() {...}`

`protected final native void dibujar();`

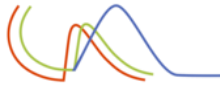
`public abstract double evaluar(double x);`

`public static double toRadians(double angulo)`

`void desplazar(double x, double y) {...}`

Modificadores de acceso (visibilidad) para métodos (igual que en variables)

	Fuera del paquete	Paquete	Clase	Subclase
<i>default (sin modificador)</i>	NO	SI	SI	SI (si mismo paquete)
<i>public</i>	SI	SI	SI	SI
<i>protected</i>	NO	SI	SI	SI (independientemente del paquete)
<i>private</i>	NO	NO	SI	NO



7. Método main

Cualquier aplicación desarrollada por Java está formada por un conjunto de clases. De todas ellas, solo una tendrá un método denominado `main` o principal. La sintaxis es:

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

Cuando ejecutamos la aplicación la ejecución comienza siempre por este método.

Ejemplo:

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

8. Clases internas

Una clase interna es una clase que se declara dentro de otra:

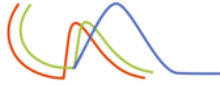
```
class Contenedora {  
    ...  
    class Interna {  
        ...  
    }  
}
```

Características:

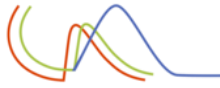
- Una clase interna puede acceder a los miembros `private` de la clase contenedora.
- Una clase interna es ocultada a las clases del mismo paquete.
- Las clases internas anónimas son útiles para implementar callbacks.

Tipos de clases internas:

- Clases internas estáticas (clases anidadas).
- Clases internas miembro (no estáticas) o simplemente clases internas.
- Clases internas locales (declaradas dentro de un bloque de código).



- Clases internas anónimas (como las anteriores pero sin nombre).



Herencia

Todas las clases en Java constituyen una jerarquía de herencia con la clase `java.lang.Object` en la raíz. Si una clase no hereda explícitamente de otra, lo hace implícitamente de `Object`.

Java admite herencia simple de clases y múltiples interfaces.

La forma completa de declaración de clases es:

`modif class nombre [extends clase] [implements interfaz, ...]`

Extends clase	La superclase directa de la clase que estamos declarando. Por defecto, es <code>java.lang.Object</code>
Implements interfaz	La lista de interfaces que implementa la clase. Por defecto, ninguno

Consideraciones:

- `modif` son los modificadores de clase que habíamos visto: `public`, `abstract` y `final`.

Ejemplos:

```
public class MiVentana extends JFrame {...}
```

```
public final class EchoHandler extends Thread {...}
```

```
public class String extends java.lang.Object implements java.io.Serializable,  
java.lang.Comparable, java.lang.CharSequence {...}
```

1. Constructores y herencia

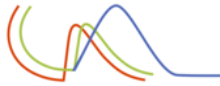
Para asegurar la correcta construcción de un objeto, todos los constructores de las superclases del mismo deben invocarse. Para llamar a un constructor de una superclase, utilizamos la palabra reservada `super`.

```
super(parámetros);
```

Reglas a seguir:

REGLA 1: la primera sentencia de un constructor debe ser una llamada con `super` a algún constructor de la superclase inmediata.

Ejemplo:



```
public class MiVentana extends JFrame { public MiVentana(String titulo) {  
super(titulo);
```

```
// ...
```

```
}
```

```
// ... }
```

REGLA 2: si no se invoca ningún constructor, Java coloca una llamada implícita `super()` a un constructor de la superclase sin argumentos.

Ejemplo:

```
public class MiVentana extends JFrame {
```

```
public MiVentana(String titulo) {
```

```
// llamada implícita a super()
```

```
}
```

Regla3: sin no se crea ningún constructor, Java crea uno por defecto sin argumentos que llama implícitamente a un constructor sin argumentos de la superclase, que debe existir.

Ejemplo:

```
public class MiVentana extends JFrame {
```

```
// Sin constructor
```

```
}
```

```
// ...
```

```
}
```

Java crea automáticamente el siguiente constructor:

```
public MiVentana() {
```

```
super();
```

```
}
```

REGLA 4: se puede retrasar la llamada a un constructor de la superclase si se invoca como

primera sentencia a un constructor propio con `this()`.

Ejemplo:

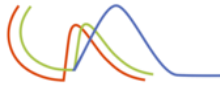
```
public class MiVentana extends JFrame {
```

```
public MiVentana(String titulo) {
```

```
super(titulo);
```

```
// ...
```

```
}
```



```
public MiVentana() {  
    this("Ventana principal"); // Llama al constructor anterior  
}  
// ..  
}
```

2. Redefinición de métodos

Una subclase puede anular (proporcionar una versión distinta) métodos de la superclase. Este es el concepto de redefinición o sobrescritura (override) de métodos. El método debe tener los mismos tipos de argumentos y tipo de retorno.

Ejemplo: De la clase `Object` heredan todas las clases. Entre otros, define el método `toString()`.

```
public class Object {  
    // ...  
    public String toString() {...}  
    // ...  
}  
  
Public class User { // no es necesario especificar extends Object  
    @Override  
    public String toString() {  
        return nombre + " " + apellidos;  
    }  
}
```

3. Clases abstractas

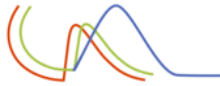
Las clases a veces representan conceptos abstractos, no instanciables. Para modelar este tipo de conceptos nacen las clases abstractas. Una clase abstracta es aquella clase que no puede instanciarse. Utilizamos la palabra reservada `abstract` en la definición de la clase.

Ejemplo:

```
abstract class Figura {}  
  
public abstract class Animal {}  
  
abstract class Vehiculo {}
```

4. Métodos abstract

Si cuando definimos una clase sabemos que un método no puede ser implementado porque el código depende del objeto concreto que se instancie a partir de una subclase dentro de la jerarquía estamos ante un método abstracto. Por ejemplo, si tenemos la clase `Figura` y las subclases `Círculo` y `Cuadrado`, el método `area()` de



la clase `Figura` no tiene implementación porque se desconoce a priori la figura concreta que tomará. Para este tipo de casos, declaramos en la superclase el método como `abstract`.

Si una clase tienen algún método `abstract`, la clase debe ser obligatoriamente `abstract`.

Las clases hijas tienen la obligación de implementar el método (o declararse `abstract`).

Ejemplo:

```
abstract class Mamifero extends Animal {  
    public abstract void hablar();  
    class Perro extends Mamifero {  
        public void hablar() {  
            System.out.println("Guau");  
        }  
    }  
    class Gato extends Mamifero {  
        public void hablar() {  
            System.out.println("Miau");  
        }  
    }  
}
```

5. Clases y métodos `final`

La palabra reservada `final` nos permite:

- En una clase, que ninguna otra clase puede heredar (o extender) a dicha clase.
- En un método, que ninguna subclase puede redefinir dicho método.

Ejemplo:

Ejemplo:

```
final class A {...}
```



NO PERMITIDO

```
class A1 extends A {...}
```

```
class B {
```

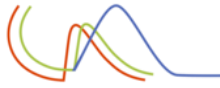
```
    final public void m() {...}
```

```
}
```



NO PERMITIDO

```
class B1 extends B {  
    public void m() {...}  
}
```



1. Definición de una interface

Las interfaces constituyen un mecanismo para separar interfaz (declaración del método) de implementación (código). Se declaran de manera similar a las clases:

```
modif interface nombre [extends interfaz, ...] {...}
```

Consideraciones:

- modif: un interfaz sólo puede tener los modificadores public y abstract (redundante).

Un interfaz únicamente puede tener firmas de métodos (sólo la declaración, sin cuerpo) y constantes.

Restricciones:

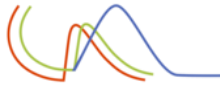
- Los métodos de una interface son implícitamente public y abstract, aunque no lo explicitemos. No pueden ser final ni static.

- Los atributos son implícitamente public, static y final y deben ser inicializados en la

declaración.



Ejemplo: definición de una interface e implementación de la misma por una clase.

```
interface Contador {  
    void incrementar();  
    void reiniciar();  
    String toString();  
}  
  
class ContadorEntero implements Contador {  
    private int valor = 0;  
    public void incrementar() {  
        valor++;  
    }  
    public void reiniciar() {  
        valor = 0;  
    }  
    public String toString() {  
        return String.valueOf(valor);  
    }  
}
```



}

Ejemplo: definición de una interface con solo constantes y aplicación de las restricciones vistas.

<pre>interface Figura2D { double area(); double perimetro(); } public interface SwingConstants { int NORTH = 1; int EAST = 3; int SOUTH = 5; int WEST = 7; // ... }</pre>	 	<pre>interface Figura2D { public abstract double area(); public abstract double perimetro(); } public interface SwingConstants { public static final int NORTH = 1; public static final int EAST = 3; public static final int SOUTH = 5; public static final int WEST = 7; // ... }</pre>
---	--	---

Es posible declarar métodos e implementarlos directamente en la interface con la palabra reservada default.

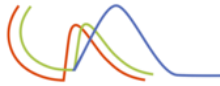
```
public interface MiInterfaz {  
    default void metodoPredeterminado() {  
        // implementación del método default  
    }  
}
```

Ejemplo:

```
public interface Vehiculo {  
    String obtenerMarca();  
    String acelerar();  
    String desacelerar();  
    default String activarAlarma() {  
        return "Activando la alarma del vehículo.";  
    }  
    default String desactivarAlarma() {  
        return "Desactivando la alarma del vehículo.";  
    }  
}
```

2. Herencia en interfaces

Entre las interfaces existe una jerarquía que permite **herencia simple y múltiple**.



Cuando una interface deriva de otra, incluye todas sus constantes y declaraciones de métodos.

Una interface puede derivar de varias interfaces. Para la herencia de interfaces se utiliza asimismo la palabra `extends`, seguida por el nombre de las interfaces de las que deriva, separadas por comas.

Ejemplo: definición de una interface que hereda de dos interfaces

```
interface Figura2DEscalable extends Figura2D, Escalable {...}
```

Los interfaces son independientes de la jerarquía de clases en Java. Sin embargo, podemos usar una referencia de tipo interfaz en cualquier lugar donde se esperaría un

`java.lang.Object`.

Ejemplo:

```
public Object hacerAlgo() {  
    ArrayList lista = new ArrayList();  
    Figura2D figura = new Circulo(2);  
    lista.add(figura);  
    ...  
    return figura;  
}
```

3. Upcasting y downcasting

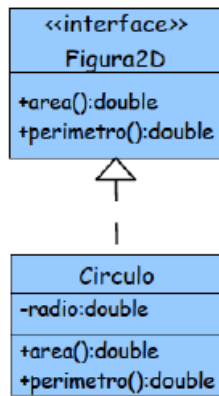
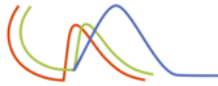
En algunas situaciones necesitamos asignar una referencia de un tipo a otra de un tipo relacionado:

- Paso de parámetros.
- Asignar a una referencia el resultado de una llamada a un método.
- Sentencia `return`.

En esos casos, hemos de cumplir unas normas.

Upcasting consiste en asignar una referencia de un tipo a otra de un tipo superior. Siempre es seguro y lo podemos hacer sin problemas.

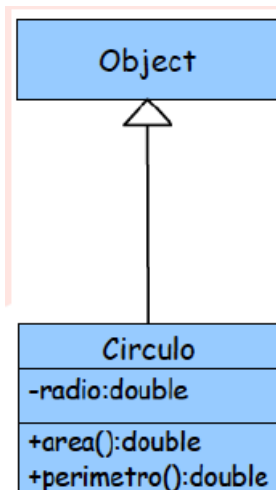
Ejemplo: La clase `Circulo` implementa la interface `Figura2D`. De forma implícita hereda de la clase `Object`.



```
Figura2D figura = new Circulo(2);
```

```
Object obj = figura;
```

Downcasting es justo lo contrario a upcasting, consiste en asignar referencias de un tipo superior a un tipo inferior.



Ejemplo:

```
ArrayList lista = new ArrayList();
```

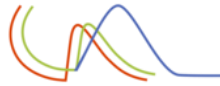
```
Circulo c = new Circulo(2);
```

```
lista.add(c); // upcasting
```

```
// Posteriormente, para recuperarlo,
```

```
// invocamos el método Object get(int i)
```

```
c = (Circulo) lista.get(0); // downcasting
```



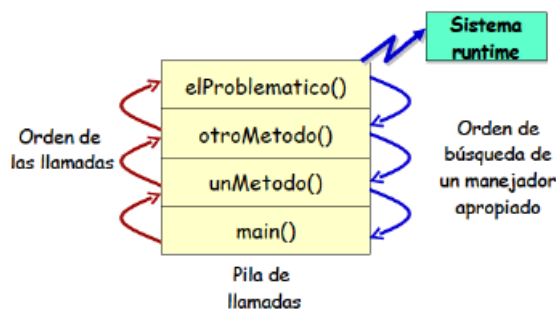
1. Definición de una excepción

Una excepción es un evento que interrumpe el flujo normal de ejecución de un programa.

Cuando durante la ejecución de un método ocurre un evento de ese tipo:

- El método lanza una excepción que contiene información sobre el error y el punto en el que se ha producido y es recogida por el sistema runtime.
- o El runtime busca en la pila de llamadas un manejador apropiado de la excepción.
- o Si lo encuentra, le pasa la excepción para su tratamiento.
- Si no lo encuentra, el programa termina.

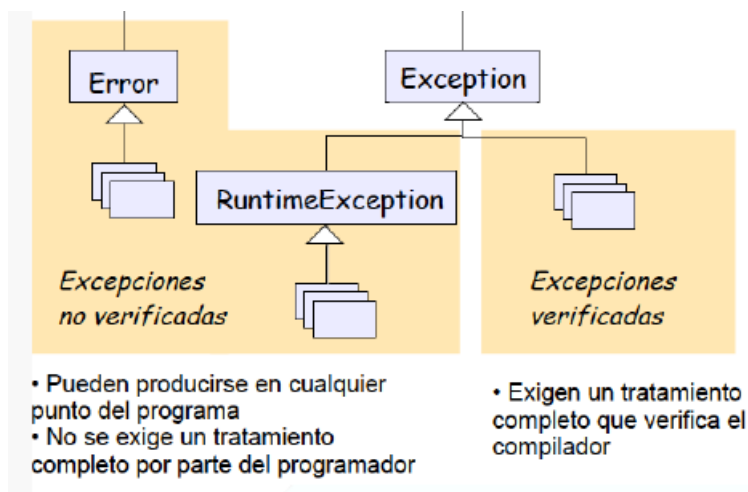
2. La pila de llamadas



El sistema runtime busca un manejador apropiado en la pila de llamadas en el orden inverso a las llamadas, empezando por el método que generó la excepción

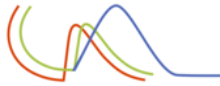
3. Tipos de excepciones: verificadas y no verificadas

Dentro de la API de Java, las clases relacionadas con el manejo de excepciones se encuentran en el paquete `java.lang`.



Errores

(`java.lang.Error` y subclases) Representan condiciones anormales que nunca deberían producirse y que una aplicación no debe tratar de capturar.



Excepciones runtime

(`java.lang.RuntimeException` y subclases)

Excepciones que pueden ocurrir en cualquier momento como una división por cero, acceder a un objeto a través de una referencia null...

Resto de excepciones (verificadas)

Se espera que la aplicación trate adecuadamente este tipo de excepciones. De clase `Throwable` destacamos los siguientes métodos:

`void printStackTrace()` Muestra por la salida estándar la traza de la pila de llamadas

`String getMessage()` Devuelve un mensaje explicando la excepción

Listado de excepciones predefinidas (clases) más importantes del paquete `java.lang`:

- `ArithmeticException`.
- `ArrayIndexOutOfBoundsException`: índice fuera de rango.
- `ClassNotFoundException`.
- **`CloneNotSupportedException`**: método `clone` llamado, pero no se implementa en la clase
- la interfaz `Cloneable`.
- `EnumConstantNotPresentException`: constante no definida.
- `IllegalArgumentException`.
- `IndexOutOfBoundsException`: índice fuera de rango.
- `NegativeArraySizeException`.
- **`NoSuchFieldException`**: campo no definido.
- **`NoSuchMethodException`**: método no definido.
- `NullPointerException`.
- `NumberFormatException`.
- `RuntimeException`.
- **`SecurityException`**: violación de seguridad lanzada por el gestor de seguridad.
- `StringIndexOutOfBoundsException`: índice fuera de rango.

4. Lanzar y definir excepciones

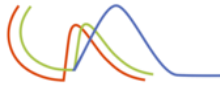
Para lanzar una excepción utilizamos una sentencia `throw`:

```
throw instancia_de_Throwable;
```

Ejemplos:

```
public Object pop()
```

```
throws PilaVacíaException {
```



```
if (ultimo >= 0)
return elementos[ultimo--];
else
throw new PilaVaciaException();
}
```

```
public Pila(int tamaño) {
if (tamaño < 0)
throw new IllegalArgumentException();
elementos = new Object[tamaño];
ultimo = -1;
}
```

Como se ha podido observar en los ejemplos, cuando definimos un método se pueden declarar las excepciones que puede lanzar dicho método. Para ello, usamos la cláusula throws en la declaración:

Sintaxis : nombreMetodo throws lista_de_tipos_Throwable

Las excepciones que un método puede lanzar forman parte del contrato del método.

Ejemplos:

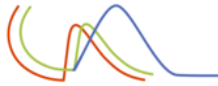
```
public Object pop() throws PilaVaciaException {...} // de Pila
public FileInputStream(String name) throws FileNotFoundException,
SecurityException {...} // de java.io.FileInputStream
void remove() throws UnsupportedOperationException, IllegalStateException {...}
// de java.util.Iterator
```

En relación con la herencia, cuando en una subclase se redefine un método de clase de la que herede, no se puede lanzar ninguna excepción que no sea del mismo tipo o subclase de las declaradas por el método de la superclase.

Ejemplo: error en la redefinición del método doPost en la subclase.

```
public class MiClase {
public void doPost(...) throws ServletException, IOException {}
}

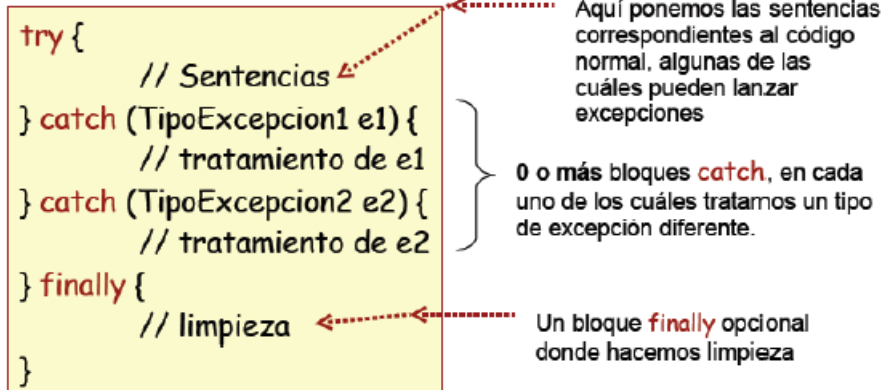
public class OtraClase extends MiClase {
public void doPost(...) throws SQLException {}
}
```



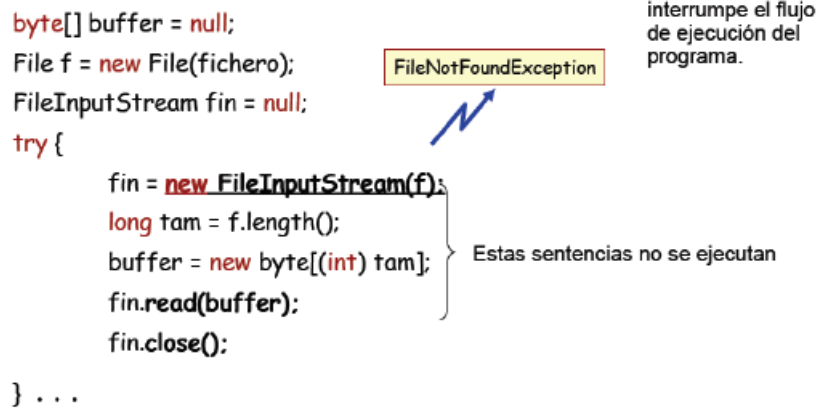
}

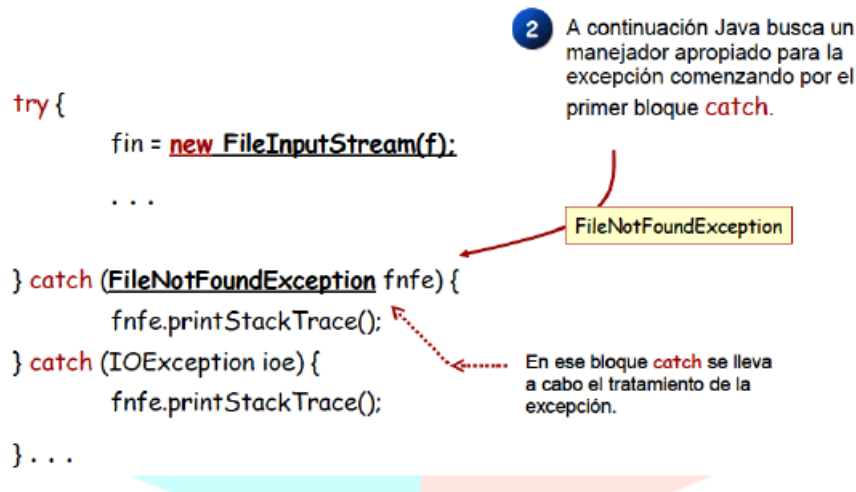
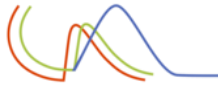
6. Capturar y tratar una excepción

Para capturar y tratar excepciones utilizamos bloques try-catch-finally:

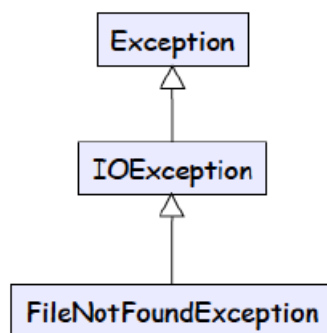


Ejemplo de funcionamiento:





En el punto 2 se dice que se busca un manejador apropiado. Esto es, se considera apropiado si la clase de la excepción lanzada es la misma o una subclase de la clase que figura en la sentencia **catch**.

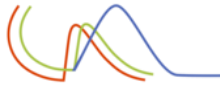


Teniendo en cuenta la jerarquía de clases de excepciones, se lanza la excepción **FileNotFoundException** y dos bloques **catch**, el primero trata excepciones del tipo **FileNotFoundException** y el segundo del tipo **IOException**. Ambos tipos de excepciones se consideran apropiados: el primero es igual a la excepción lanzada y el segundo es una superclase del tipo lanzado. Entonces, ¿qué bloque se ejecuta? En este caso, se ejecuta el primer bloque **catch** porque está situado en primer lugar.

Por tanto, a la hora de definir bloques **catch**, su orden irá de los tipos de excepciones más “concretos” a los más “abstractos”.

Por último, tenemos el bloque **finally**. Después de un bloque **try** y de los bloques **catch** que se definan, de forma opcional se puede definir un bloque **finally** que se ejecutará independientemente de si ha habido excepciones o no.

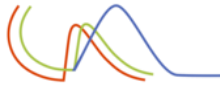
6. Crear excepciones personalizadas



Según el tipo de excepción, extenderemos `Exception` (lo normal), `RuntimeException` o `Error`. Simplemente proporcionaremos los constructores apropiados que invocarán mediante `super(...)` al constructor de la superclase.

Ejemplo:

```
class PilaVacíaException extends Exception {  
    public PilaVacíaException() {}  
    public PilaVacíaException(String msg) {  
        super(msg);  
    }  
    public PilaVacíaException(Throwable th) {  
        super(th);  
    }  
    public PilaVacíaException(String msg, Throwable th) {  
        super(msg, th);  
    }  
}
```



10. ENTRADA / SALIDA

En Java se accede a la E/S estándar a través de campos estáticos de la clase `java.lang.System`:

- **`System.in`** implementa la entrada estándar. Es una instancia de la clase `InputStream`.
- Método `read()`.
- **`System.out`** implementa la salida estándar. Es una instancia de la clase `PrintStream`.
- Métodos `print()` y `println()`.
- **`System.err`** implementa la salida de error. Igual que `System.out`.

Por tanto, para recibir mensajes utilizamos la sentencia:

```
System.in.read()
```

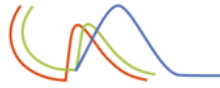
Para enviar mensajes por la salida estándar (pantalla) hacemos uso de los métodos:

```
System.out.print()
```

```
System.out.println()
```

El paquete `java.io` proporciona entrada y salida del sistema a través de flujos de datos,

serialización y el sistema de archivos.



AWT (Abstract Window Toolkit) es una biblioteca pesada de componentes IU definida a través del paquete `java.awt`. La gestión de elementos gráficos depende del sistema operativo (por eso es pesada) que es quien dibuja y gestiona la interacción sobre los elementos.

El aspecto de los componentes varía según el sistema operativo.

La API Swing (paquete `javax.swing`) es una biblioteca ligera de componentes IU, lo que significa que es Java quien visualiza y gestiona la interacción del usuario sobre los elementos de la IU. El aspecto de los componentes es similar, independientemente del sistema operativo sobre el que se ejecuta la aplicación.

Elementos principales de la API:

- **JFrame**: ventana principal con marco y barra de título.

`JFrame`

```
ventana = new JFrame("Primera Ventana");
```

```
ventana.setSize(400, 400);
```

```
ventana.setVisible(true);
```

- `JButton`.

```
JFrame ventana = new JFrame("Primera Ventana");
```

```
JButton boton = new JButton("Un botón");
```

```
ventana.getContentPane().add(boton); //Añadimos al panel de la ventana
```

```
ventana.setSize(400, 400);
```

```
ventana.setVisible(true);
```

- **JPanel**: contenedor de componentes IU.

```
JFrame ventana = new JFrame("FlowLayout Manager");
```

```
Container contenedor = ventana.getContentPane();
```

```
JPanel panel = new JPanel(); //Este panel contiene los componentes
```

```
panel.add(new JButton("Uno"));
```

```
panel.add(new JButton("Dos"));
```

```
panel.add(new JButton("Tres"));
```

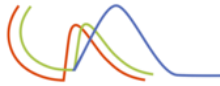
```
panel.add(new JButton("Cuatro"));
```

```
panel.add(new JButton("Cinco"));
```

```
contenedor.add(panel);
```

```
ventana.setSize(200, 200);
```

```
ventana.setVisible(true);
```



- **JTextArea**: campo de texto, en el que se puede definir filas y columnas.

```
JTextArea campo_observaciones = new JTextArea();
```

```
campo_observaciones.setColumns(50);
```

```
campo_observaciones.setRows(25);
```

- **TextField**: campo de texto.

```
TextField campo = new TextField();
```

- **Label**: etiqueta.

```
Label etiqueta = new Label();
```

```
etiqueta.setText("Nombre y apellidos");
```

Otros elementos:

- o JApplet: ejecución en navegador web (no recomendado).
- o JComponent.
- o JCheckBox.
- o JComboBox.
- o JDialog: ventana secundaria (por ejemplo, con explorador para localizar un fichero).
- o JMenu.
- o JMenuItem.
- o JProgressBar.
- o JRadioButton.
- o JSlider.
- o JSpinner.

Ejemplo:

```
import javax.swing.*;
```

```
public class HelloWorldSwing {
```

```
private static void createAndShowGUI() {
```

```
//Create and set up the window.
```

```
JFrame frame = new JFrame("HelloWorldSwing");
```

```
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
JLabel label = new JLabel("Hello World"); //Add "Hello World" label.
```

```
frame.getContentPane().add(label);
```

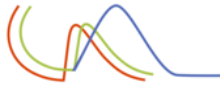
```
frame.pack();
```

```
frame.setVisible(true); //Display the window.
```

```
}
```

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

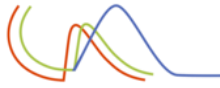
```
//Schedule a job for the event-dispatching thread:
```

//creating and showing this application's GUI.

```
javax.swing.SwingUtilities.invokeLater(new Runnable() {  
    public void run() {  
        createAndShowGUI();  
    }  
});  
}
```

JavaFX es una plataforma de aplicaciones cliente de código abierto para sistemas integrados, móviles y de escritorio construida en Java, con el objetivo de producir un conjunto de herramientas moderno, eficiente y con todas las funciones para desarrollar aplicaciones de cliente enriquecidas.



1. Java 8

Métodos por defecto en interfaces

Hasta ahora las interfaces en Java solo podían definir métodos pero no sus implementaciones.

El problema con las interfaces es que cuando se modifican se “rompen” todas las clases que las usan. Esto se ha resuelto de tal forma que se puedan añadir nuevos métodos con implementación por defecto a las interfaces.

Ejemplo:

```
public interface Math {  
    int add(int a, int b);  
    default int multiply(int a, int b) {  
        return a * b;  
    }  
}
```

Métodos estáticos en interfaces

Además de definir métodos por defecto en las interfaces a partir de ahora podemos definir métodos estáticos. Definiendo métodos estáticos en las interfaces evitaremos tener que crear clases de utilidad. Podremos incluir dentro la interface todos los métodos relacionados.

Ejemplo:

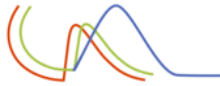
```
public interface Persona {  
    String getNombre();  
    int getAltura();  
    static String toStringDatos() {  
        return getNombre() + " " + getAltura();  
    }  
}
```

2. Java 9

Se incorporan los módulos.

3. Java 10

Se incorpora la palabra reservada `var`, esto ayuda a no tener que indicar el tipo de dato concreto en la declaración de un objeto.



Con la inferencia de tipos los nombres de las variables cobran mayor importancia dado que `var` elimina la posibilidad al lector del código adivinar la intención de una variable a partir del tipo.

Ejemplo:

// Java 9

```
Class.forName("org.postgresql.Driver");
```

```
Connection connection =
```

```
DriverManager.getConnection("jdbc:postgresql://localhost/database", "user",  
"password");
```

```
PreparedStatement statement = connection.prepareStatement("select * from  
user");
```

```
ResultSet resultSet = statement.executeQuery();
```

// Java 10

```
Class.forName("org.postgresql.Driver");
```

```
var connection =
```

```
DriverManager.getConnection("jdbc:postgresql://localhost/database", "user",  
"password");
```

```
var statement = connection.prepareStatement("select * from user");
```

```
var resultSet = statement.executeQuery();
```

4. Java 11

Ejecución desde archivo de código fuente único

Para ejecutar un programa Java es necesario compilarlo a bytecode y posteriormente ejecutarlo. Se necesitan dos pasos. Para facilitar la ejecución de los programas que se componen de un único archivo de código fuente se añade la posibilidad de lanzar un programa desde el archivo de código fuente. Esto es útil para programas pequeños o para los casos de aprendizaje del lenguaje.

Unicode 10

Soporte de Unicode 10.

Cliente HTTP

Incorporación del cliente HTTP con soporte para HTTP/2 en el propio JDK.

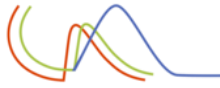
5. Java 12

Recolector de basura **Shenandoah**

Se incorpora un nuevo recolector de basura denominado Shenandoah.

Cambios en switch

Se elimina el uso de `break`.



6. Java 13

Uso triple comilla doble

Hasta ahora en Java, para definir una cadena de caracteres que tuviese varias líneas había que concatenarlas. El resultado es una cadena con problemas de legibilidad por los caracteres de escape que pueda incluir. La legibilidad empeora si la cadena de caracteres tiene elementos HTML, JSON, sentencias SQL o expresiones regulares.

Ejemplo:

```
String html = "<html>\n" +  
" <body>\n" +  
" <p class=\"text\">Hello, Escapes</p>\n" +  
" </body>\n" +  
"</html>\n";
```

Como solución, Java propone para este tipo de bloques de texto el uso de una triple comilla doble al comienzo y al final del bloque para encerrar el texto.

Ejemplo:

```
String html = """"  
<html>  
<body>  
<p class="text">Hello, Text Blocks</p>  
</body>  
</html>""";
```

Expresiones switch mejoradas

Incorporan bloques de código y la opción de retornar un valor.

7. Java 14

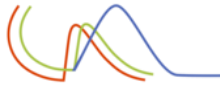
Records

Característica más destacada que permite reducir significativamente el código necesario para algunas clases.

Los registros son clases que no contienen más datos que los públicos declarados. Evitan muchas líneas de código para definir los constructores y los métodos getter.

Ejemplo:

```
final class Rectangle implements Shape {  
    final double length;  
    final double width;
```



```
public Rectangle(double length, double width) {  
    this.length = length;  
    this.width = width;  
    double length() {return length;}  
    double width() {return width;}  
}  
Record Rectangle (float length, float width){}  
  
}
```